Ornaments and Proof Transport applied to Numerical Representations

Samuel Klumpers
6057314

May 15, 2023

Abstract

Todo list

poke . . . 3
poke . . . 4
Make sure this does not literally repeat the introduction too often. . 5
Merge . . . 8
Express what this accomplishes, and why this is impressive compared to without univalence . . . 12
Generalizing the observation that lists look like unary naturals and Braun trees look like binary naturals. . . . 12
Taking the writing out of our hands, formalizing the "looks like relation". . . . 13
Adapt and split into background and actual work . . . 13
Is there a simple twist or other interesting example that we can run through instead, or would anything else be too abrupt without starting from this simple case? . . . 13
Put some minimal definitions here. . . . 16
Adapt this to the non-proposal form . . . 22
Merge . . . 26
This section is about as digestable as a brick. . . . 30

Contents

1 Introduction **2**
1.1 The Problem . . . 3
1.2 Contributions . . . 4

2 Background **5**
2.1 Agda . . . 5
2.2 Cubical Agda . . . 7
2.3 The Structure Identity Principle . . . 7

2.3.1 Unary numbers are binary numbers 9
2.3.2 Functions from specifications 10
2.3.3 The Structure Identity Principle 12
2.4 Numerical representations 12
2.5 Generic programming and ornaments 13

I Numerical representations and ornaments 13

3 Types from Specifications: Ornamentation and Calculation 13
3.1 From numbers to containers 13
3.2 Numerical representations as ornaments 15
3.3 Heterogeneization 17

II Enumeration 18

4 Enumeration 18
4.1 Basic strategy 19
4.2 Cardinalities 21
4.3 Indexed types 21

5 Related work 22
5.1 The Structure Identity Principle 22
5.2 Numerical Representations 22
5.3 Ornamentation 23
5.4 Generic constructions 23

III Appendix 25

A More equivalences for less effort 25
A.1 Well-founded monic algebras are initial 26
A.1.1 Datatypes as initial algebras 27
A.1.2 Accessibility 29
A.1.3 Example 30

1 Introduction

Program verification is an indispensible aspect of programming, whether you're coding up your own Asteroids or you're developing a linear algebra library, it would be a waste of time to hunt for bugs which could have been uncovered by random testing. When testing does not offer enough certainty or cannot handle the complexity of the input, we can instead use formal program verification: it would be embarassing if someone else suffers the consequences of a bug in your library, so you might prove your library or parts of it correct in a proof assistant like Coq or Agda. In a more extreme example, you might code directly into a proof assistant, specifying the behaviour of your program beforehand, having it checked while you're implementing it.

Yet, program verification, especially of the last kind, is a double-edged sword: while it becomes easier to write code without bugs, it becomes harder to write code in the first place. A proof assistant has to enforce total and terminating programs (at least by default), as incomplete or circular steps would undermine the correctness of a proof. Non-total operations are abundant in most languages, like getting the first element of a list; such operations would require the programmer to provide evidence that the operation can not fail at each usage. In this example the evidence can be encoded by modifying a list to remember its length, and generally we can create variations on datastructures for use in correct-by-construction programs.

This might prompt defining variations for each use case, and duplicating all operations on them, making little or no use of the fact that types like lists and vectors are strongly related. But this can be avoided, since a broad class of relations has been tamed by ornaments [McB14; KG16]. Informally, an ornament describes the pieces of information necessary to construct a new type from an existing type.

However, we do not have to stop at relating lists and vectors. Just like vectors can be described as lists with more information, lists can be described as natural numbers with more information [McB14]. This can be generalized to other datastructures, such as binary numbers and trees. The idea of instead constructing datastructures from number systems has been studied as numerical representations [Oka98; HS22]. This provides a way to talk about datastructures using their underlying numbers, and allows one to mechanically calculate datastructures and some of their properties from these numbers, albeit manually.

By calculating a datastructure, one hopes to gain an isomorphism between the datatype represented as a lookup function, and the concrete version of the datatype. As the representation and the concrete type are equivalent, one can reason about properties of the concrete side by looking at the representation, which is often simpler. In the usual context, one would still have to manually convert proofs back and forth. More conveniently, we would like to apply representation independence; similarly to how equality of indiscernables ensures that exchanging equal terms cannot change the behaviour of a program, the same should hold for isomorphic types. While such results usually only exists in the meta-theory, or can only be applied on concrete types by manually weaving conversions through proofs, structured equivalences [Ang+20] can internalize this, at the cost of using Cubical Agda.

1.1 The Problem

The main question of this project is: *can we describe finger trees [HP06] in the frameworks of numerical representations and ornamentation [KG16], simplifying the verification of their properties as flexible two-sided arrays?* This question

poke

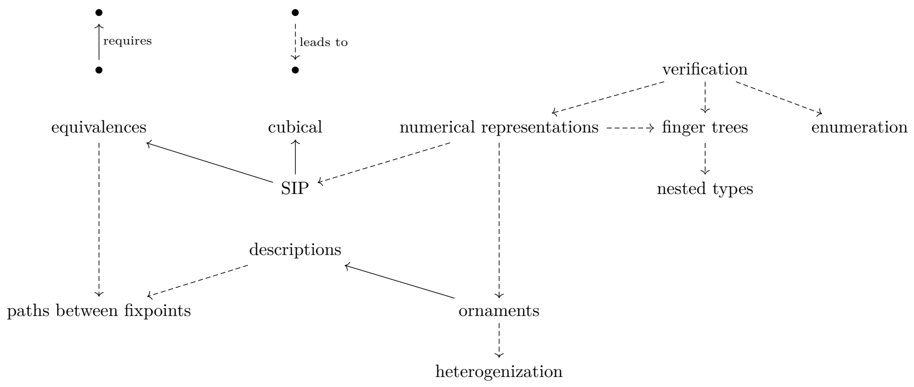

Figure 1: Temporary overview

generates a number of interesting subproblems, such as that the number system corresponding to finger trees has many representations for the same number, which we expect to describe using quotients [VMA19] and reason about using representation independence [Ang+20].

1.2 Contributions

In this paper, we

poke

- adapt ornaments to nested types
- allow ornaments to refer to sub-ornaments

x define a small universe of typical number systems

- give a generic derivation of numerical representations as ornaments from these number systems
- instantiate a Structure Identity Principle for these representations.

We follow this up by enumerating these, and more structures. We

x define hierarchies to enumerate terms by levels

x track the cardinalities of each level

- include parametrized datatypes into this setup
- modify this to include nested types
- adapt this approach to index-first datatypes
- iterate the accessible indices per level

Along the way, we also

x characterize identities of W-types

x express heterogeneous variants of datastructures as ornaments.

2 Background

2.1 Agda

We formalize our work in Agda [Tea23], a functional programming language with dependent types. Using dependent types we can use Agda as a proof assistant, allowing us to state and prove theorems about our datastructures and programs. These proofs can then be run as algorithms, or in some cases be extracted to a Haskell program[1].

Make sure this does not literally repeat the introduction too often.

Syntactically Agda is similar to Haskell, with a few notable differences. One is that Agda allows most characters and words in identifiers with only a small set of exceptions. For example, we can write

```
_🍧_🌶_ : Bool → A → A → A
false 🍧 t 🌶 e = e
true 🍧 t 🌶 e = t
```

The other is that datatypes are given either as generalized algebraic datatypes (GADTs) or record types in Haskell.

The type system of Agda is an extension of (intensional) Martin-Löf type theory (MLTT), a constructive type theory in which we can interpret intuitionistic logic. Compared to Haskell, which extends a polymorphic lambda calculus with inductive types, MLTT allows types of codomains of functions to vary with values in the domains: Whereas in Haskell only datatypes can map into types[2], in Agda we define functions into Type

$$\dots$$

given a function f from A into Type, we can then form the type

$$\dots$$

Likewise, the type of the second field of a pair type can vary with the value of the first. The presence of these types enriches the interpretation of logic into programs, known as the Curry-Howard isomorphism: propositions or logical formulas are related to types, such that a term of a type constitutes a proof of the related proposition.

To ensure that the logic interpreted by this isomorphism remains consistent, Agda rules out non-terminating functions by restricting their definitions

[1] Or JavaScript, if you want.

[2] Excluding extensions

to structural recursion. The termination checker (together with other restrictions which we will encounter in due time) prevents trivial proofs which would be tolerated in Haskell, like

```
undefined : ∀ {A : Type} → A
undefined = undefined
```

The propositional part of the Curry-Howard correspondence can then be formulated by the usual type formers. The atomic formulas, true and false, can be represented respectively as the empty record: there always is a proof tt of true

```
record ⊤ : Type where
  constructor tt
```

and the type with no constructors: there is no way to make a proof of false

```
record ⊥ : Type where
```

Implication $A \implies B$ corresponds to function types $A \to B$: a proof of A can be converted to a proof of B. Implication also gives an interpretation of negation as functions into false $A \to \bot$. Disjunction (logical or) is described by a sum type $A + B$: either of A or B can prove $A + B$

```
data _+_ A B : Type where
  inl : A → A + B
  inr : B → A + B
```

Conjunction (logical and) is given as a product type: having both A and B proves $A \times B$

```
record _×_ A B : Type where
  constructor _,_
  field
    fst : A
    snd : B
```

Predicates, formulas containing variables, correspond to functions into the type of formulas

```
P : A → Type
```

allowing interpretations of higher-order logic. Quantifiers are interpreted via dependent types. Universal quantification (for all) is a dependent function type: for each $a : A$, give a proof of $P\ a$

```
(a : A) → P a
```

Likewise, existential quantification (exists) is a dependent pair type: there is an $a : A$ and a proof $P\ a$

```
record ∃ A (P : A → Type) : Type where
  constructor _,_
  field
    fst : A
    snd : P fst
```

Predicates can also be expressed using indexed datatypes, in which the choice of constructor can influence the index, whereas parameters must be constant over all constructors. Equality of elements of a type A can then be interpreted as the type

```
data Eq (a : A) : A → Type where
  refl : Eq a a
```

Closed terms of this type can only be constructed for definitionally equal elements, but crucially, variables can contain equalities between different elements. As the second argument is an index, pattern matching on refl unifies the elements, such that properties like substitution follow

```
subst : Eq a b ⟶ P a ⟶ P b
subst refl x = x
```

With this, we can do math. For example, we could define natural numbers as an inductive type

· · ·

and set out to prove the elementary properties of prime numbers. But to get the same results to binary numbers (without duplicating the proofs), we need a bit more. The usual notion of equalities of types are isomorphisms: two types A, B are isomorphic if there are functions $A \to B$ and $B \to A$, which are mutually inverse

· · ·

In ordinary Agda, we cannot directly apply these to transport along like we can for equalities, however.

2.2 Cubical Agda

Intuitively, one expects that like how isomorphic groups share the same group-theoretical properties, isomorphic types also share the same type-theoretical properties. Meta-theoretically, this is known as *representation independence*, and is evident. Inside (ordinary) Agda this is not so practical, as this independence only holds when applied to concrete types, and is then only realized by manually substituting along the isomorphism. On the other hand, in Cubical Agda, the Structure Identity Principle internalizes a kind of representation independence [Ang+20].

Cubical Agda modifies the type theory of Agda to a kind of homotopy type theory, looking at equalities as paths between terms rather than the equivalence relation generated by reflexivity. In cubical type theories, the role played by pattern matching on refl or by axiom J, in MLTT and "Book HoTT" respectively, is instead acted out by directly manipulating cubes[3]. In Cubical Agda, univalence is not an axiom but a theorem.

2.3 The Structure Identity Principle

To give an understanding of the basics of Cubical Agda [VMA19] and the Structure Identity Principle (SIP), we walk through the steps to transport proofs about addition on Peano naturals to Leibniz naturals. We give an overview of some features of Cubical Agda, such as that paths give the primitive notion of equality, until the simplified statement of univalence. We do note that Cubical

[3] Under the analogy where a term is a point, an equality between points is a line, a line between lines is a square.

Agda has two downsides relating to termination checking and universe levels, which we encounter in later sections.

Starting by defining the unary Peano naturals and the binary Leibniz naturals, we prove that they are isomorphic by interpreting them into eachother. We explain that these interpretations are easily seen to be mutual inverses by proving lemmas stating that both interpretations "respect the constructors" of the types. Next, we demonstrate how this isomorphism can be promoted into an equivalence or an equality, and remark that this is sufficient to transport intrinsic properties, such as having decidable equality, from one natural to the other.

Noting that transporting unary addition to binary addition is possible but not efficient, we define binary addition while ensuring that it corresponds to unary addition. We present a variant on refinement types as a syntax to recover definition from chains of equality reasoning, allowing one to rewrite definitions while preserving equalities.

We clarify that to transport proofs referring to addition from unary to binary naturals, we indeed require that these are meaningfully related. Then, we observe that in this instance, the pairs of "type and operation" are actually equated as magmas, and explain that this is an instance of the SIP.

Finally, we describe the use case of the SIP, how it generalizes our observation about magmas, and how it can calculate the minimal requirements to equate to implementations of an interface. This is demonstrated by transporting associativity from unary addition to binary addition, noting that this would save many lines of code provided there is much to be transported.

Merge

Let us quickly review some features of Cubical Agda [VMA19] that we will use in this section.

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.

Paths between types are incredibly useful, as they effectively let us directly transport properties between isomorphic structures. However, they do not come without downsides, such as that the negation of axiom K complicates both some termination checking and some universe levels.[4]

We will discuss how to deal with these issues in later sections, so let us not be distracted from what we *can* do with paths. For example, the 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

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

the given path.

Also, because we can now interpret paths in record and function types in a new way, 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, equivalences, the HoTT-compatible variant of bijections, have the univalence theorem

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

stating that “equivalent types are identified”, such that equivalences like $1 \to A \simeq A$ become paths $1 \to A \equiv A$, making it so that we can transport proofs along them. We will demonstrate this by a more practical example in the next section.

2.3.1 Unary numbers are 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 is well-covered in most libraries. However, the definition is also impractically slow, since most arithmetic operations defined on ℕ have time complexity in the order of the value of the result.

As an alternative we can use binary numbers, for which for example addition has logarithmic time complexity. Standard libraries tend to contain few proofs about binary number properties, but this does not have to be a problem: 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
⟦_⟧ = toℕ
```

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 0 = 0b
fromℕ (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 → ⟦ bsuc x ⟧ ≡ suc ⟦ x ⟧
```

so toℕ respects successors. Similarly, by induction on ℕ we get

```
fromℕ-1+2· : ∀ x → fromℕ (1 + double x) ≡ (fromℕ x) 1b
```

and

```
fromℕ-2+2· : ∀ x → fromℕ (2 + double 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
  ret : retract fromℕ toℕ
```

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; as an example, we have not yet shown that Leibniz is discrete, i.e., has decidable equality. Using substitution, we can quickly derive this[5]

```
discreteL : Discrete Leibniz
discreteL = subst Discrete ℕ≡L discreteℕ
```

This can be generalized even further to transport proofs about operations from ℕ to Leibniz.

2.3.2 Functions from specifications

As an example, we will define addition of binary numbers. We could transport _+_ as a binary operation

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

from ℕto Leibnizto get

[5] Of course, this gives a rather inefficient equality test, but for the homotopical consequences this is not a problem.

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

But this is inefficient, incurring an $O(n+m)$ overhead when adding n and 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 a syntax for giving definitions by equational reasoning, inspired by the "use-as-definition" notation used by Hinze and Swierstra [HS22]: 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
```

we define

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

which extracts a definition as the right endpoint of a given path.

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 (⟦ x ⟧ + ⟦ y ⟧))
plus-def 0b y =
      fromℕ ⟦ y ⟧
  ≡⟨ ℕ↔L .rightInv y ⟩
      y ∎ use-as-def
plus-def (x 1b) (y 1b) =
      fromℕ ((1 + double ⟦ x ⟧) + (1 + double ⟦ y ⟧))
  ≡⟨ solved ⟩
      fromℕ (2 + (double (⟦ x ⟧ + ⟦ y ⟧)))
  ≡⟨ fromℕ-2+2· (⟦ x ⟧ + ⟦ y ⟧) ⟩
      fromℕ (⟦ x ⟧ + ⟦ y ⟧) 2b
  ≡⟨ cong _2b (by-definition (plus-def x y)) ⟩
      defined-by (plus-def x y) 2b ∎ use-as-def
-- ...
```

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

We remark that Def is close in concept to refinement types[6], but extracts the value from the proof, rather than requiring it before. [7]

2.3.3 The Structure Identity Principle

We point out that ℕ with N.+ and Leibniz with plus form magmas, that is, inhabitants of

```
Magma' : Type₁
Magma' = Σ[ X ∈ Type ] BinOp X
```

Using that a path in a dependent pair corresponds to a dependent pair of paths, we get a path from (ℕ, N.+) to (Leibniz, plus). This observation is further generalized by the Structure Identity Principle (SIP) as a form of representation independence [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
```

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
```

Express what this accomplishes, and why this is impressive compared to without univalence

2.4 Numerical representations

Generalizing the observation that lists look like unary naturals and Braun trees look like binary naturals.

[6] À la Data.Refinement.

[7] 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.

2.5 Generic programming and ornaments

Taking the writing out of our hands, formalizing the “looks like relation”.

Part I
Numerical representations and ornaments

Adapt and split into background and actual work

3 Types from Specifications: Ornamentation and Calculation

Suppose that we started writing and verifying some code using a vector-based implementation of the two-sided flexible array interface, but later decide to reimplement more efficiently using trees. It would be a shame to lay aside our vector lemmas, and rebuild the correctness proofs for trees from scratch. Instead, we note that both vectors and trees can be represented by their lookup function. In fact, we can ask for more, and rather than 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 by Okasaki [Oka98], and has some formalized examples due to Hinze and Swierstra [HS22] and Ko and Gibbons [KG16]. Numerical representations are our starting point for defining more complex datastructures based on simpler ones, so we demonstrate such a calculation.

3.1 From numbers to containers

We can compute the type of vectors starting from ℕ.

Is there a simple twist or other interesting example that we can run through instead, or would anything else be too abrupt without starting from this simple case?

[8] 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

[8] This is adapted (and fairly abridged) from Calculating Datastructures [HS22]

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

Since we defined Vec such that it agrees with Lookup, we can relate their implementations as well.

The implementation of arrays as functions is 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[9]

```
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
```

Now, rather than rederiving the laws for vectors, the equality allows us to transport them from Lookup to Vec.[10]

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

Reflecting on this derivation for ℕ, we could perform the same computation for Leibniz to get Braun trees. However, 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 [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

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

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

need to be introduced or dropped to go from one description to the other. *Ornamental descriptions*, which act as one-sided ornaments, let us describe new datatypes by recording the modifications to an existing description.

Put some minimal definitions here.

Looking back at Vec, ornaments let us show that express that Vec can be formed by introducing indices and adding a fields holding an elements to ℕ.However, deriving List from ℕ generalizes to Leibniz with less notational overhead, so we tackle that case first. We use the following description of ℕ

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

Here, σ adds a field to the description, upon which the rest of the description can 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. Using the least fixpoint and description extraction, we can then define List from this ornamental description. 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! Thus, 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.

3.3 Heterogeneization

The situation in which one wants to collect a variety of types is not uncommon, and is typically handled by tuples. However, if e.g., you are making a game in Haskell, you might feel the need to maintain a list of “Drawables”, which may be of different types. Such a list would have to be a kind of “heterogeneous list”. In Haskell, this can be resolved by using an existentially quantified list, which, informally speaking, can contain any type implementing a given constraint, but can only be inspected as if it contains the intersection of all types implementing this constraint.

This ports directly to Agda, but becomes cumbersome quickly, and 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. This approach and mainly its specialization to binary trees is investigated by Swierstra [Swi20].

We will demonstrate that we can express this “lift a type over itself” operation 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 E : RDesc ⊤ ℓ-zero) (x : Ḟ (λ _ → D) (μ (λ _ → E) Type) Type tt)
        → ROrnDesc (μ (λ _ → E) Type tt) ! D
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 identical, 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 (ṿ []) })

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 , _ , _)
```

Part II
Enumeration

4 Enumeration

Property based testing frameworks often rely on random generation of values, consider for example the Arbitrary class of Quickcheck [CH00]. How these values are best generated depends on the property being tested; if we are testing an implementation of insertSorted, we should probably generate sorted lists [**rest**]! Some frameworks like Quickcheck do provide deriving mechanisms for Arbitrary instances, but this relinquishes most control over the distribution. This leaves manually re-implementing Arbitrary as necessary as the only option for a user who wants to test properties with more sophisticated preconditions.

A more controllable alternative to random generation is the complete enumeration of all values. Provided that such an enumeration supports efficient (and fair) indexing, one can adjust a random distribution of values by controlling the sampling from enumerations. There is rich theory of enumeration, and this problem has also been tackled numerous times in the context of functional programming. Some approaches focus on the efficient indexing of enumerations [DJW12], others focus on generating indexed types as a means of enumerating values with invariants [RS22].

We will describe a framework generalizing these approaches, which will support:

1. unique and complete enumeration
2. indexing by (exact) recursive depth
3. fast skipping through the enumeration
4. indexed, nested, and mutually recursive types

We will follow an approach similar to the list-to-list approach [RS22], but rather than expressing enumerations as a step-function, computing the next generation of values from a list of predecessors, we will keep track of the entire depth indexed hierarchies.

4.1 Basic strategy

We define a hierarchy of elements as

```
Hierarchy : Type → Type
Hierarchy A = ℕ → List A
```

When applied to a number n, a hierarchy should then return the list of elements of exactly depth n. To iteratively approximate hierarchies, we define a hierarchy-builder type

```
Builder : (A B : Type) → Type
Builder A B = Hierarchy A → Hierarchy B
```

Hierarchy-builders should be able to take a partially defined hierarchy, and return a hierarchy which is defined at one higher level.

We implement some basic hierarchy building blocks, such as the one-element builder

```
pure : B → Builder A B
pure x _ zero = [ x ]
pure x _ (suc n) = []
```

which represents nullary constructors, and the shift builder

```
rec : Builder A A
rec B zero = []
rec B (suc n) = B n
```

which represents recursive fields.

To interpret sum types, we use an interleaving operation. Consider that for the disjoint sum, the elements at level n must be formed from elements which are also at level n, regardless whether they are from the left summand or the right.

```
_⟨|⟩_ : Builder A B → Builder A C → Builder A (B ⊎ C)
(B₁ ⟨|⟩ B₂) V n = interleave (mapL inl (B₁ V n)) (mapL inr (B₂ V n))
```

For product types, the elements at level n are those which contain at least one component at level n, so we have to sum all possible combinations of products

```
pair : Builder A B → Builder A C → Builder A (B × C)
pair B₁ B₂ V n =
    (downFrom (suc n) >>= λ i → (prod (B₁ V n) (B₂ V i)))
  ++ (downFrom n >>= λ i → (prod (B₁ V i) (B₂ V n)))
```

We claim that this is sufficient to enumerate the following simple universe of types

```
data Desc : Set where
  one : Desc
  var : Desc
  _⊗_ : (D E : Desc) → Desc
  _⊕_ : (D E : Desc) → Desc

⟦_⟧ : Desc → Set → Set
⟦ one ⟧ X = ⊤
⟦ var ⟧ X = X
⟦ D ⊗ E ⟧ X = ⟦ D ⟧ X × ⟦ E ⟧ X
⟦ D ⊕ E ⟧ X = ⟦ D ⟧ X ⊎ ⟦ E ⟧ X

data μ (D : Desc) : Set where
  con : ⟦ D ⟧ (μ D) → μ D
```

In the same vein as other generic constructions, we can define a generic builder by cases over the interpretetation

```
builder : ∀ {D} D′ → Builder (μ D) (⟦ D′ ⟧ (μ D))
builder one = pure tt
builder var  = rec
builder (D ⊗ E) = pair (builder D) (builder E)
builder (D ⊕ E) = builder D ⟨|⟩ builder E
```

By applying constructors, we can wrap this up into an endomorphism at a fixpoint

```
gbuilder : ∀ D → Builder (μ D) (μ D)
gbuilder D V = mapH con (builder D V)
```

Finally, we observe that applying this builder $n+1$ times to the empty hierarchy is sufficient to approximate the hierarchy up to level n

```
iterate : ℕ → (A → A) → A → A
iterate zero f x = x
iterate (suc n) f x = f (iterate n f x)

build : Builder A A → Hierarchy A
build B n = iterate (suc n) B (const []) n

hierarchy : ∀ D → Hierarchy (μ D)
hierarchy D = build (gbuilder D)
```

which gives us the generic hierarchy

We can for example apply this to generate binary trees of given depths

```
TreeD : Desc
TreeD = one ⊕ (var ⊗ var)

TreeH = hierarchy TreeD
```

which returns the following trees of level 2

```
node (node leaf leaf) (node leaf leaf)
∷ node (node leaf leaf) leaf
```

```
  ∷ node leaf (node leaf leaf)
  ∷ []
```

However, it would be even cooler if

1. An enumeration could tell us how many elements there are of some depth
2. An enumeration was a map from constructor to subsequent enumerations
3. The possible indices get computed as we go down.

The first is essential for sampling. The second would give the user total control over the shapes of their generated values. And the third is particularly crucial when the set of possible indices is small.

4.2 Cardinalities

Simplifying our earlier approach a bit, we can tinker

```
Hierarchy : Type → Type
Hierarchy A = ℕ → ℕ × List A
```

to track the sizes. For example, our interleaving operation becomes

```
_⟨|⟩_ : Hierarchy A → Hierarchy B → Hierarchy (A ⊎ B)
(V₁ ⟨|⟩ V₂) n with V₁ n | V₂ n
... | c₁ , xs | c₂ , ys = c₁ + c₂ , interleave (mapL inl xs) (mapL inr ys)
```

We can write down a generic hierarchy

```
{-# TERMINATING #-}
ghierarchy : ∀ D {E} → Hierarchy (⟦ D ⟧ (μ E))
ghierarchy one = pure tt
ghierarchy var zero = 0 , []
ghierarchy var (suc n) = mapH con (ghierarchy _) n
ghierarchy (D ⊗ E) = ghierarchy D ⊛ ghierarchy E
ghierarchy (D ⊕ E) = ghierarchy D ⟨|⟩ ghierarchy E
-- note that the termination checker also does not like this case,
-- so inline it if you want to get rid of the pragma
```

Then we can count

```
numTrees : ℕ → ℕ
numTrees n = fst (TreeH n)
```

and see that there are 210065930571 trees of level 6, wow! It still takes a bit of time to walk across all branches and products in the description, because there is no memoization at all, but it's a lot better than counting the trees after generating them. Also indexing will be slow, even knowing this information, because we're working with plain lists. Things would probably already get a lot better if we worked with trees that know the sizes of their children.

4.3 Indexed types

Ideally, we get a meaningful list or enumeration of indices at the end: the non-empty ones. However, we do not (yet) require the index type to be enumerable.

The index-first presentation of indexed datatypes, while efficient and succinct, does not seem suitable for this. After all, the descriptions for such a presentation live in the function space from the index to the base descriptions. We would rather want to start "recklessly applying" constructors and seeing what kinds of indices that leaves us with.

This example explains why it's also pretty hopeless for Sijsling's descriptions: We would need a notion of "forward indexed type" in which the indices in the arguments must be strictly less crazy than those in the resulting type.

Anyway, we restrict our attention to indexed types that work, that is, we can decide whether an index fits. In the previous example, the constructor would instead compute whether n is $n' + 2$, and return n' if it is. This completely breaks any attempt at counting the enumeration.

In comparison, the index-first presentation tells us nothing about which indices are reachable, but probably does better with counting. I suppose you could combine them at the cost of a lot, and first run the forward idea on only the indices, and then see how much each index has, or something.

5 Related work

Adapt this to the non-proposal form

5.1 The Structure Identity Principle

If we write a program, and replace an expression by an equal one, then we can prove that the behaviour of the program can not change. Likewise, if we replace one implementation of an interface with another, in such a way that the correspondence respects all operations in the interface, then the implementations should be equal when viewed through the interface. Observations like these are instances of "representation indepencence", but even in languages with an internal notation of type equality, the applicability is usually exclusive to the metatheory.

In our case, moving from Agda's "usual type theory" to Cubical Agda, a cubical homotopy type theory, *univalence* [VMA19] lets us internalize a kind of representation independence known as the Structure Identity Principle [Ang+20], and even generalize it from equivalences to quasi-equivalence relations. We will also be able to apply univalence to get a true "equational reasoning" for types when we are looking at numerical representations.

Still, representation independence in non-homotopical settings may be internalized in some cases [Kap23], and remains of interest in the context of generic constructions that conflict with cubical.

5.2 Numerical Representations

Rather than equating implementations after the fact, we can also "compute" datastructures by imposing equations. In the case of container types, one may observe similarities to number systems [Oka98] and call such containers numerical representations. One can then use these representations to prototype new datastructures that automatically inherit properties and equalities from their underlying number systems [HS22].

From another perspective, numerical representations run by using representability as a kind of “strictification” of types, suggesting that we may be able to generalize the approach of numerical representations, using that any (non-indexed) infinitary inductive-recursive type supports a lookup operation [DS16].

5.3 Ornamentation

While we can derive datastructures from number systems by going through their index types [HS22], we may also interpret numerical representations more literally as intstructions to rewrite a number system to a container type. We can record this transformation internally using ornaments, which can then be used to derive an indexed version of the container [McB14], or can be modified further to naturally integrate other constraints, e.g., ordering, into the resulting structure [KG16]. Furthermore, we can also use the forgetful functions induced by ornaments to generate specifications for functions defined on the ornamented types [DM14].

5.4 Generic constructions

Being able to define a datatype and reflect its structure in the same language opens doors to many more interesting constructions [EC22]; a lot of “recipes” we recognize, such as defining the eliminators for a given datatype, can be formalized and automated using reflection and macros. We expect that other type transformations can also be interpreted as ornaments, like the extraction of heterogeneous binary trees from level-polymorphic binary trees [Swi20].

References

[Ang+20] Carlo Angiuli et al. *Internalizing Representation Independence with Univalence*. 2020. arXiv: `2009.05547 [cs.PL]`.

[CH00] Koen Claessen and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *SIGPLAN Not.* 35.9 (Sept. 2000), pp. 268–279. ISSN: 0362-1340. DOI: `10.1145/357766.351266`. URL: `https://doi.org/10.1145/357766.351266`.

[DJW12] Jonas Duregård, Patrik Jansson, and Meng Wang. “Feat: Functional Enumeration of Algebraic Types”. In: *SIGPLAN Not.* 47.12 (Sept. 2012), pp. 61–72. ISSN: 0362-1340. DOI: `10.1145/2430532.2364515`. URL: `https://doi.org/10.1145/2430532.2364515`.

[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`.

[DS16] Larry Diehl and Tim Sheard. “Generic Lookup and Update for Infinitary Inductive-Recursive Types”. In: *Proceedings of the 1st International Workshop on Type-Driven Development*. TyDe 2016. Nara, Japan: Association for Computing Machinery, 2016, pp. 1–12. ISBN: 9781450344357. DOI: `10.1145/2976022.2976031`. URL: `https://doi.org/10.1145/2976022.2976031`.

[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`.

[HP06] Ralf Hinze and Ross Paterson. “Finger trees: a simple general-purpose data structure”. In: *Journal of Functional Programming* 16.2 (2006), pp. 197–217. DOI: `10.1017/S0956796805005769`.

[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.

[Kap23] Kevin Kappelmann. *Transport via Partial Galois Connections and Equivalences*. 2023. arXiv: `2303.05244 [cs.PL]`.

[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.

[RS22] Cas van der Rest and Wouter Swierstra. “A Completely Unique Account of Enumeration”. In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: `10.1145/3547636`. URL: `https://doi.org/10.1145/3547636`.

[Swi20] WOUTER Swierstra. “Heterogeneous binary random-access lists”. In: *Journal of Functional Programming* 30 (2020), e10. DOI: `10.1017/S0956796820000064`.

[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`.

Part III
Appendix

A More equivalences for less effort

Noting that constructing equivalences directly or from isomorphisms as in Subsection 2.3 can quickly become challenging when one of the sides is complicated, we work out a different approach making use of the initial semantics of W-types instead. We claim that the functions in the isomorphism of Subsection 2.3 were partially forced, but this fact was not made use of.

First, we explain that if we assume that one of the two sides of the equivalence is a fixpoint or initial algebra of a polynomial functor (that is, the μ of a Desc′), this simplifies giving an equivalence to showing that the other side is also initial.

We describe how we altered the original ornaments [KG16] to ensure that μ remains initial for its base functor in Cubical Agda, explaining why this fails otherwise, and how defining base functors as datatypes avoids this issue.

In a subsection focussing on the categorical point of view, we show how we can describe initial algebras (and truncate the appropriate parts) in such a way that the construction both applies to general types (rather than only sets), and still produces an equivalence at the end. We explain how this definition, like the usual definition, makes sure that a pair of initial objects always induces a pair of conversion functions, which automatically become inverses. Finally, we explain that we can escape our earlier truncation by appealing to the fact that "being an equivalence" is a proposition.

Next, we describe some theory, using which other types can be shown to be initial for a given algebra. This is compared to the construction in Subsection 2.3, observing that intuitively, initiality follows because the interpretation of the zero constructor is forced by the square defining algebra maps, and the other values are forced by repeatedly applying similar squares. This is clarified as an instance of recursion over a polynomial functor.

To characterize when this recursion is allowed, we define accessibility with respect to polynomial functors as a mutually recursive datatype as follows. This datatype is constructed using the fibers of the algebra map, defining accessibility of elements of these fibers by cases over the description of the algebra. Then we remark that this construction is an atypical instance of well-founded recursion, and define a type as well-founded for an algebra when all its elements are accessible.

We interpret well-foundedness as an upper bound on the size of a type, leading us to claim that injectivity of the algebra map gives a lower bound, which is sufficient to induce the isomorphism. We sketch the proof of the theorem, relating part of this construction to similar concepts in the formalization of well-founded recursion in the Standard Library. In particular, we prove an irrelevance and an unfolding lemma, which lets us show that the map into any other algebra induced by recursion is indeed an algebra map. By showing that it is also unique, we conclude initiality, and get the isomorphism as a corrolary.

The theorem is applied and demonstrated to the example of binary naturals. We remark that the construction of well-foundedness looks similar to view-patterns. After this, we conclude that this example takes more lines that the direct deriviation in Subsection 2.3, but we argue that most of this code can likely be automated.

Merge

Using Subsection 2.3 we can relate functionally equivalent structures, and using Section 3 we can relate structurally similar structures. However, both have downsides; the former requires us to construct isomorphisms, and the latter wraps all components behind a layer of constructors. In this section will alleviate these problems through generics and by alternative descriptions of equivalences.

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 Subsection 2.3, we see that when one establishes an equivalence, most of the time is spent working out a series of lemmas that prove the conversion functions are to be mutual inverses. We note that the functions themselves were, in fact, 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′.

A.1 Well-founded monic algebras are initial

Unfortunately, the machinery developed by Ko and Gibbons [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 the original setup, the recursion of mapFold is a structural descent in ⟦ D' ⟧ (μ D). Because ⟦_⟧ is a

[11] Of course, indexed datatypes are indexed W-types, mutually recursive datatypes are represented yet differently...

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-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.[13]

Recall that the Base functors of descriptions are special polynomial functors, and the fixpoint of a base functor is its initial algebra. 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 $(\mu F, \mathrm{con})$, then $X \cong \mu F$ would follow immediately, so it is equivalent to ask for the algebras to be isomorphic instead.

A.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]

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

[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.

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 a 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
```

Because being an equivalence is a property, we can eliminate from the truncations to get the wanted result.

A.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 A.1. The fixpoint μD is not in general 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 Subsection 2.3, 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.

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 get out of reach of an algebra. *Now* having $FX \cong X$ also gives us a lower bound, but note 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 A.1. If there is a mono $f : FX \to X$ and X is well-founded for f, then X is an initial F-algebra.

Proof sketch of Claim A.1. 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"[15], so we use a similar strategy. 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
```

□

A.1.3 Example

Let us redo the proof in Subsection 2.3, 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 view-patterns (the main difference being that we look 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)
```

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

```
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 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
```

In this case, we needed more lines of code to prove the same statement, however, the process of writing became more forced, and might be more amenable to automation.