Ornaments and Proof Transport applied to Numerical Representations

Samuel Klumpers
6057314

June 20, 2023

Contents

Todo list

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 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].

Revisit this when further

1.2 Contributions

In this paper, we:

Revisit this when further

x Adapt ornaments to nested types.

x Allow ornaments to refer to sub-ornaments.

x Define a small universe of typical number systems.

x Give a generic derivation of numerical representations as ornaments from these number systems.

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.

Background

2 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 other languages like Haskell[1].

Syntactically Agda is reminiscent of Haskell. One difference 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
```

Another is that datatypes are always either given as generalized algebraic datatypes (GADTs) or record types. For example, the definition of booleans

[1] Or JavaScript, if you want.

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

can be written in Agda as

```
data Bool : Type where
  false : Bool
  true  : Bool
```

The unit type

```
data Unit = Unit
```

becomes[2]

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

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: the Curry-Howard isomorphism states that certain formulas correspond to certain types, and proofs of a formula correspond to terms of the corresponding type. The atomic formula true can be represented by `⊤`, so that `tt` always proves true. False can be represented by a datatype with no constructors

```
data ⊥ : Type where
```

since there is (hopefully) no way to make get a term of `⊥` without inconsistent assumptions. The logical implication $A \implies B$ corresponds to the type of functions $A \to B$: a proof of A can be converted to a proof of B. Using implication, we can define the negation $\neg A$ of a formula A as the type $A \to \bot$. Disjunction (logical or) is described by a sum type $A + B$:

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

if we have either A or B, we can prove $A + B$. Conjunction (logical and) is given as a product type:

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

we need both A and B to prove $A \times B$. Using the correspondence, we can reason in propositional logic by writing functional programs. As an example, consider the proof of the tautology

```
→-×-undistr : ((A → C) × (B → C)) → (A + B) → C
→-×-undistr (a→c , b→c) (inl a) = a→c a
→-×-undistr (a→c , b→c) (inr b) = b→c b
```

Compared to Haskell, Agda allows the type of the codomain of a function to vary with the applied value: given a function P from A into `Type`, a type family over A, we can form the dependent function type $(a : A) \to P\ a$. Applying a function $f : (a : A) \to P\ a$ to a value $a : A$ then will have type $f\ a : P\ a$.

[2]One can also write the unit type as a datatype with one constructor. However, in Agda, records (can) benefit from eta-expansion. In our case, all terms of `⊤` are definitionally equal.

Similarly, the type of a field in a record type can depend on values of earlier fields, e.g.,

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

The presence of these dependent types enriches the interpretation of logic into programs. To interpret first-order logic we need to describe formulas containing variables, which are called predicates. Predicates correspond to functions

```
P : A → Type
```

Using predicates, we can interpret quantifiers as the dependent types above. Universal quantification (for all) is a dependent function type

```
(a : A) → P a
```

since for each $a : A$, we have a proof of $P\ a$. Likewise, existential quantification (exists) is the dependent pair type ∃, since this gives an $a : A$ and a proof $P\ a$.

Indexed what, example fin

Predicates can also be expressed using indexed datatypes, in which the choice of constructor can influence the index. 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 of this type 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
```

Unlike most languages, Agda rules out non-terminating functions by restricting their definitions 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
```

This ensures that all our interpretations mentioned above remain consistent.

3 • Generic programming

To be able to reason about datatypes themselves, we first have to represent datatypes by another datatype. This can be done by defining a datatype of codes instructing how datatypes can be formed, together with a function assigning the meaning to this encoding, henceforth description and interpretation respectively. We will start from an encoding which captures only a small set of types, and work towards an encoding of parametrized indexed types.

We can describe the universe of finite types with the following description:

```
data Desc : Type where
```

```
  𝟘 𝟙     : Desc
  _⊕_ _⊗_ : Desc → Desc → Desc
```
Each of the constructors of this description represents a type-former. In this case, the universe only contains sums and products of the 0 and 1; the meaning of the type-formers comes from the interpretation:
```
μ : Desc → Type
μ 𝟘       = ⊥
μ 𝟙       = ⊤
μ (D ⊕ E) = μ D ⊎ μ E
μ (D ⊗ E) = μ D × μ E
```
Booleans live in this universe as
```
BoolD : Desc
BoolD = 𝟙 ⊕ 𝟙
```
but to encode a type like `ℕ` we need a different setup. Consider the definition
```
data ℕ : Type where
  zero : ℕ
  suc  : ℕ → ℕ
```
we can interpret this as the declaration `ℕ ≃ ⊤ ⊎ ℕ`, and formally `ℕ` is indeed the least fixpoint of this equation. In category theoretic terms we would say that `ℕ` is the initial algebra of its base functor `(⊤ ⊎_)`. Letting
```
⟦_⟧ : Desc → Type → Type
```
assign base functors to descriptions, we can take the fixpoint as
```
data μ (D : Desc) : Type where
  con : ⟦ D ⟧ (μ D) → μ D
```
We see that if `⟦ ND ⟧` is `(⊤ ⊎_)`, then `μ ND` satisfies the equation for `ℕ`.

We change the codes to
```
data Desc where
  𝟙 ρ     : Desc
  _⊕_ _⊗_ : Desc → Desc → Desc
```
and describe the base functors:
```
⟦ 𝟙     ⟧ X = ⊤
⟦ ρ     ⟧ X = X
⟦ D ⊕ E ⟧ X = (⟦ D ⟧ X) ⊎ (⟦ E ⟧ X)
⟦ D ⊗ E ⟧ X = (⟦ D ⟧ X) × (⟦ E ⟧ X)
```
Now, `𝟙` encodes the leaves of a datatype, and `ρ` encodes a recursive node. The operators `⊕` and `⊗` are changed to act pointwise. In this universe, we can define `ℕ` by
```
ℕD : Desc
ℕD = 𝟙 ⊕ ρ
```
To describe complex types more practically, we can merge `ρ` and `⊗`, and add a variant `σ` of `⊗`, which then represent adding a recursive and a non-recursive field respectively
```
data Desc : Type₁ where
  𝟙   : Desc
  ρ   : Desc → Desc
  σ   : (S : Type) → (S → Desc) → Desc
```

```
    _⊕_ : Desc → Desc → Desc
```

In σ, we ask for a function `S → Desc` rather than just a `Desc`, modelling a `Desc` with a bound variable of type `S`. The interpretation is similar, interpreting ρ and σ as a product and dependent product respectively.

In this universe we can describe types in which the fields be either `X`, the type itself, or another type `S`. For example, we can describe `List` as

```
ListD : Type → Desc
ListD A = 𝟙 ⊕ (σ A λ _ → ρ 𝟙)
```

using a type parameter from outside the description. We will soon see how we can internalize parameters, but since internalizing indices is easier, we will tackle indices first.

We should note that there are two strategies we can use to describe an indexed type. First, we can define a description of a type indexed by `I` to simply be a function `I → Desc`, yielding a universe of index-first types. Second, we can pull the index completely into `Desc`, and let 𝟙 declare the index at the leaf of a constructor, more closely resembling Agda's datatypes. Both have their advantages and disadvantages, mainly, index-first datatypes are more space efficient. We opt however for the second option, because as we will see later, this allows us to keep descriptions "relatively small" (i.e., something like foldable) and more flexible in their levels.

```
data Desc (I : Type) : Type₁ where
  𝟙   : I → Desc I
  ρ   : I → Desc I → Desc I
  σ   : (S : Type) → (S → Desc I) → Desc I
  _⊕_ : Desc I → Desc I → Desc I
```

Now `𝟙 j` says that this branch constructs a term of `X j`, while `ρ i` asks for a recursive field `X i`. As `Desc I` describes a type indexed by `I`, which is a function `I → Type`, we also have to interpret `Desc I` as an indexed functor

```
⟦_⟧ : Desc I → (I → Type) → (I → Type)
⟦ 𝟙 j   ⟧ X i = i ≡ j
⟦ ρ j D ⟧ X i = X j × ⟦ D ⟧ X i
⟦ σ S D ⟧ X i = Σ[ s ∈ S ] ⟦ D s ⟧ X i
⟦ D ⊕ E ⟧ X i = ⟦ D ⟧ X i ⊎ ⟦ E ⟧ X i
```

Applying an interpretation to an index asks for the constructors at that index. We see that by interpreting `𝟙 j` as an equality, we ensure that asking for an index indeed gives that index.

In this universe we can describe vectors

```
VecD : Type → Desc ℕ
VecD A = (𝟙 zero) ⊕ (σ ℕ λ n → σ A λ _ → ρ n (𝟙 (suc n)))
```

making use of the variable binding in σ to state that if we get a vector of length `n`, then we can construct a vector of length `suc n`.

The observant reader might have noticed that we claim `I → Desc` does not give small descriptions, but still allow for `S → Desc`. We can fix this issue at the same time we implement parameters, keeping a form of variable binding. Implementing types with a single parameter can be done by interpreting to "endofunctors" `Type → I → Type`, adding another type-former accessing the parameter. To handle types with more parameters, which may depend on each other, we abstract descriptions over lists of parameters.

We will first need some structure expressing the kinds of parameters that we can have. We could try using List Type, but this rules out types like `Σ (A : Type) (B : A → Type)`. Instead, we use a telescope, a list of types which explicitly captures the dependencies. We define telescopes and their meaning by induction-recursion:

```
data Tel : Type₁
⟦_⟧tel : Tel → Type

data Tel where
  ∅   : Tel
  _▷_ : (Γ : Tel) (S : ⟦ Γ ⟧tel → Type) → Tel
```

A telescope can either be empty, or be formed from a telescope and a type in the context of that telescope.

Contexts are interpreted as To deal with variables, we will also need to be able to describe variable telescopes. This means that while the parameter telescope in a description stays constant, the variable telescope grows independently when we add more `σ`'s. We can represent this by parametrizing telescopes over a type

```
data Tel (P : Type) : Type₁
⟦_⟧tel : Tel P → P → Type
_⊢_ : Tel P → Type a → Type a
Γ ⊢ A = Σ _ ⟦ Γ ⟧tel → A

data Tel P where
  ∅   : Tel P
  _▷_ : (Γ : Tel P) (S : Γ ⊢ Type) → Tel P
```

We define a shorthand `Γ ⊢ A` for the type of `S`, representing a value of `A` in context `Γ`. By changing `⟦_⟧tel` to depend on a value of `P` as

```
⟦ ∅     ⟧tel p = ⊤
⟦ Γ ▷ S ⟧tel p = Σ[ x ∈ ⟦ Γ ⟧tel p ] S (p , x)
```

a telescope of the form

```
ExTel : Tel ⊤ → Type₁
ExTel Γ = Tel (⟦ Γ ⟧tel tt)
```

can access all values of `Γ`, and can be treated as an extension of `Γ`. To interpret them, we define

```
⟦_&_⟧tel : (Γ : Tel ⊤) (V : ExTel Γ) → Type
⟦ Γ & V ⟧tel = Σ (⟦ Γ ⟧tel tt) ⟦ V ⟧tel
```

To make use of this we also split `⊕` and `Desc`, making `Desc` a list of constructors, in line with actual Agda datatypes

```
data Con (Γ : Tel ⊤) (V : ExTel Γ) (I : Type) : Type₁
data Desc (Γ : Tel ⊤) (I : Type) : Type₁ where
  []  : Desc Γ I
  _∷_ : Con Γ ∅ I → Desc Γ I → Desc Γ I
```

A constructor then starts off with the empty variable context, which grows as fields are added

```
data Con Γ V I where
  𝟙 : V ⊢ I → Con Γ V I
  ρ : V ⊢ I → Con Γ V I → Con Γ V I
  σ : (S : V ⊢ Type) → Con Γ (V ▷ S) I → Con Γ V I
```

replacing I by V ⊢ I in 𝟙 and ρ allows the index of a constructor or argument to depend on the preceding fields, of which the values are made accessible by appending them to the context as V ▷ S in σ. Finally, we interpret this as

```
⟦_⟧C : Con Γ V I → (⟦ Γ & V ⟧tel → I → Type) → (⟦ Γ & V ⟧tel → I → Type)
⟦ 𝟙 j   ⟧C X pv i = i ≡ (j pv)
⟦ ρ j C ⟧C X pv i = X pv (j pv) × ⟦ C ⟧C X pv i
⟦ σ S C ⟧C X pv@(p , v) i = Σ[ s ∈ S pv ] ⟦ C ⟧C (X ∘ map₂ proj₁) (p , v , s) i

⟦_⟧D : Desc Γ I → (⟦ Γ ⟧tel tt → I → Type) → (⟦ Γ ⟧tel tt → I → Type)
⟦ []     ⟧D X p i = ⊥
⟦ C ∷ Cs ⟧D X p i = ⟦ C ⟧C (X ∘ proj₁) (p , tt) i ⊎ ⟦ Cs ⟧D X p i
```

with the fixpoint

```
data μ (D : Desc Γ I) (p : ⟦ Γ ⟧tel tt) (i : I) : Type where
  con : ⟦ D ⟧D (μ D) p i → μ D p i
```

Formalizing the “looks like relation”.

4 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.

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

4.1 Paths

4.2 Univalence

4.3 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 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 each other. We observe how the interpretations are 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 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.

4.3.1 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
```

[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 C.

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
```

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

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

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

4.3.2 Functions from specifications

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

This doesn't really belong here.

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

from `ℕ`to `Leibniz`to get

```
_+′_ : 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]

4.3.3 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,

[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. On the other hand, it should not be hard to write a macro to define a function rebuilding the case tree while replacing any `defined-by` by that function.

```
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

Part I

Descriptions and ornaments

If we are going to simplify working with complex containers by instantiating generic programs to them, we should first make sure that these types fit into the descriptions.

We construct descriptions for nested datatypes by extending the encoding of parametric and indexed datatypes from Section 3 with three features: information bundles, parameter transformation, and description composition. Also, to make sharing constructors easier, we introduce variable transformations. Transforming variables before they are passed to child descriptions allows both aggressively hiding variables and introducing values as if by let-constructs.

We base the encoding of off existing encodings [Sij16; EC22]. The descriptions take shape as sums of products, enforce indices at leaf nodes, and have explicit parameter and variable telescopes. Unlike some encodings [eff20; EC22], we do not allow higher-order inductive arguments.

We use type-in-type and with-K to simplify the presentation, noting that these can be eliminated respectively by moving to Typeω and by implementing interpretations as datatypes.

5 +The descriptions

We use telescopes identical to those in Section 3:

```
data Tel (P : Type) : Type
⟦_⟧tel : (Γ : Tel P) → P → Type

_⊢_ : (Γ : Tel P) → Type → Type
Γ ⊢ A = Σ _ ⟦ Γ ⟧tel → A

data Tel P where
  ∅ : Tel P
  _▷_ : (Γ : Tel P) (S : Γ ⊢ Type) → Tel P

⟦ ∅ ⟧tel p = ⊤
```

```
⟦ Γ ▷ S ⟧tel p = Σ (⟦ Γ ⟧tel p) (S ∘ (p ,_))

ExTel : Tel ⊤ → Type
ExTel Γ = Tel (⟦ Γ ⟧tel tt)
```

Recall that a Tel represents a sequence of types, which can depend on the external type P. This lets us represent a telescope succeeding another using ExTel. A term of the interpretation ⟦_⟧tel is then a sequence of terms of all the types in the telescope.

We use some shorthands

```
_▷′_ : (Γ : Tel P) (S : Type) → Tel P
Γ ▷′ S = Γ ▷ const S

_&_⊢_ : (Γ : Tel ⊤) → ExTel Γ → Type → Type
Γ & V ⊢ A = V ⊢ A

⟦_&_⟧tel : (Γ : Tel ⊤) (V : ExTel Γ) → Type
⟦ Γ & V ⟧tel = Σ (⟦ Γ ⟧tel tt) ⟦ V ⟧tel

Cxf : (Γ Δ : Tel ⊤) → Type
Cxf Γ Δ = ⟦ Γ ⟧tel tt → ⟦ Δ ⟧tel tt

Vxf : (Γ : Tel ⊤) (V W : ExTel Γ) → Type
Vxf Γ V W = ∀ {p} → ⟦ V ⟧tel p → ⟦ W ⟧tel p

Vxf0 : (f : Cxf Γ Δ) (V : ExTel Γ) (W : ExTel Δ) → Type
Vxf0 f V W = ∀ {p} → ⟦ V ⟧tel p → ⟦ W ⟧tel (f p)
_⇉_ : (X Y : A → Type) → Type
X ⇉ Y = ∀ a → X a → Y a

_≡_ : (X Y : A → B → Type) → Type
X ≡ Y = ∀ a b → X a b → Y a b

liftM2 : (A → B → C) → (X → A) → (X → B) → X → C
liftM2 f g h x = f (g x) (h x)

! : {A : Type} → A → ⊤
! _ = tt
```

As we will see in Section 11, some generics require descriptions augmented with more information. For example, a number system needs to describe both a datatype and its interpretation into naturals. This can be incorporated into a description by allowing description formers to query specific pieces of information. We will control where and when which pieces get queried by parametrizing descriptions over information bundles

```
record Info : Type where
  field
    𝟙i : Type
    ρi : Type
```

```
σi : (S : Γ & V ⊢ Type) → Type
δi : Tel ⊤ → Type → Type
```

Here a bundle declares for example that `𝟙i` is the type of information has to be provided at a `𝟙` former. Remark that in `σi`, the bundle can ask for something depending on the type of the field. In `δi`, the bundle can ask something regarding the parameters and indices (e.g., it can force only unindexed subdescriptions.).

Example 5.1. For example, we can encode a class of number systems using the information

```
Number : Info
Number .𝟙i = ℕ
Number .ρi = ℕ
Number .σi S = ∀ p → S p → ℕ
Number .δi Γ J = Γ ≡ ∅ × J ≡ ⊤ × ℕ
```

(refer to Section 11). If we then define the unit type, when viewed as a `Number` Unit we have to provide the information that the only value of the unit type evaluates to 1.

We can recover the conventional descriptions by providing the plain bundle:

```
Plain : Info
Plain .𝟙i = ⊤
Plain .ρi = ⊤
Plain .σi _ = ⊤
Plain .δi _ _ = ⊤
```

We define the “down-casting” of information as

```
record InfoF (L R : Info) : Type where
  field
    𝟙f : L .𝟙i → R .𝟙i
    ρf : L .ρi → R .ρi
    σf : {V : ExTel Γ} (S : V ⊢ Type) → L .σi S → R .σi S
    δf : ∀ Γ A → L .δi Γ A → R .δi Γ A
```

allowing us to reuse more specific descriptions in less specific ones, so that e.g., a number system can be used in a plain datatype.

We can now define the descriptions, which should represent a mapping between parametrized indexed functors

```
PIType : Tel ⊤ → Type → Type
PIType Γ J = ⟦ Γ ⟧tel tt → J → Type
```

Recall that a description

```
data DescI If Γ J where
  []  : DescI If Γ J
  _∷_ : ConI If Γ J ∅ → DescI If Γ J → DescI If Γ J
```

is simply a list of constructor descriptions

```
data ConI (If : Info) (Γ : Tel ⊤) (J : Type) (V : ExTel Γ) : Type where
```

The interpretations `⟦_⟧` of the formers can be found below.

Leaves are formed by

```
𝟙 : {if : If .𝟙i} (j : Γ & V ⊢ J) → ConI If Γ J V
```

Here `if` queries information according to `If`, and `j` computes the index of the leaf from the parameters and variables.

A recursive field is formed by

```
ρ : {if : If .ρi}
    (j : Γ & V ⊢ J) (g : Cxf Γ Γ) (C : ConI If Γ J V)
  → ConI If Γ J V
```

where `j` now determines the index of the recursive field. The function `g` represents a parameter transform: the parameters of the recursive field can now changed at each recursive level, allowing us to describe nested datatypes. The remainder of the fields are described by `C`. Note that a recursive field is intentionally not brought into scope: making use of it requires induction-recursion anyway!

A non-recursive field is formed similarly to a recursive field

```
σ : (S : V ⊢ Type) {if : If .σi S}
    (h : Vxf Γ (V ▷ S) W) (C : ConI If Γ J W)
  → ConI If Γ J V
```

The type of the field is given by `S`, which may depend on the values of the preceding fields. We bring the field into scope, so we continue the description in an extended context. However, we allow the remainder of the description to provide a conversion from `V ▷ S` into `W` to select a new context. This makes it possible to hide fields which are unused in the remainder.

Almost analogously, we make composition of descriptions internal by a variant of σ

```
δ : {if : If .δi Δ K} {iff : InfoF If′ If}
    (j : Γ & V ⊢ K) (g : Γ & V ⊢ ⟦ Δ ⟧tel tt) (R : DescI If′ Δ K)
    (h : Vxf Γ (V ▷ liftM2 (μ R) g j) W) (C : ConI If Γ J W)
  → ConI If Γ J V
```

This takes a description `R`, and acts like the σ of `μ R`, only with more ceremony. This will allow us to form descriptions by composing other descriptions, avoiding multiplying the number of constructors of composite datatypes.

Similar to ρ, the functions `j` and `g` control indices and parameters, only now of the applied description. As we allow the description `R` of the field to have a different kind of information bundle `If′`, we must ask that we can down-cast it into `If` via `iff`.

Descriptions and constructor descriptions can then be interpreted to appropriate kind of functor, constructor descriptions also taking variables

```
⟦_⟧ : {t : Tag Γ} → UnTag Γ J t → PIType Γ J → UnFun Γ J t
⟦_⟧ {t = CT V} (𝟙 j)         X pv i
  = i ≡ j pv

⟦_⟧ {t = CT V} (ρ j f D)     X pv@(p , v) i
  = X (f p) (j pv) × ⟦ D ⟧ X pv i

⟦_⟧ {t = CT V} (σ S h D)     X pv@(p , v) i
  = Σ[ s ∈ S pv ] ⟦ D ⟧ X (p , h (v , s)) i
```

```
⟦_⟧ {t = CT V} (δ j g R h D) X pv@(p , v) i
    = Σ[ s ∈ μ R (g pv) (j pv) ] ⟦ D ⟧ X (p , h (v , s)) i

⟦_⟧ {t = DT}   []          X p i
    = ⊥

⟦_⟧ {t = DT}   (C ∷ D)      X p i
    = (⟦ C ⟧ X (p , tt) i) ⊎ (⟦ D ⟧ X p i)
```

We see that a leaf becomes a constraint between expected index and the actual index. A recursive field passes down a transformation of the current parameters and the expected index computed from the variables, before interpreting the remainder of the description. Likewise, a non-recursive field adds a field with type depending on variables, but also adds this field to the variables, which are then transformed and passed on to the remainder. The composite field is analogous, only adding a field from a description rather than a type. Finally, the list of constructor descriptions are interpreted as alternatives.

The fixpoint can then be taken over the interpretation of a description

```
data μ D p where
  con : ∀ {i} → ⟦ D ⟧ (μ D) p i → μ D p i
```

giving the datatype represented by the description.

We can then give a generic fold for the represented datatypes which descends the description, mapping itself over all recursive fields before applying the folding function.

Remark 5.1. The situation of `fold` is very common when dealing with different kinds of recursive interpretations: functions from the fixpoint are generally defined from functions out of the interpretation, generalizing over the inner description while pattern matching on the outer description.

Note that the fold requires a rather general function, limiting its usefulness: because of the parameter transformations, we cannot instantiate the fold to a single parameter. Defining, e.g., the vector sum, would require us to inspect the description, and ask that a vector of naturals can be converted into a vector of naturals, which is trivial in this case.

Sigma plus/minus

Let's look at some examples. We can encode the naturals as a type parametrized by ∅ and indexed by ⊤

```
NatD : Desc ∅ ⊤
NatD = 𝟙 _
     ∷ ρ0 _ (𝟙 _)
     ∷ []
```

Lists can be encoded similarly, but this time using the telescope

```
ListTel : Tel ⊤
ListTel = ∅ ▷ const Type
```

declaring that lists have a single type parameter. Compared to the naturals, the description now also asks for a field in the second case

```
ListD : Desc ListTel ⊤
ListD = 𝟙 _
```

```
      ∷ σ- (par top) (ρ0 _ (𝟙 _))
      ∷ []
```

Since the type parameter is at the top of the parameter telescope, the type of the field is given as `par top`.

Vectors are described using the same structure, but have indices in ℕ.

```
VecD : Desc ListTel ℕ
VecD = 𝟙 (const 0)
     ∷ σ- (par top) (σ+ (const ℕ) (ρ0 top (𝟙 (suc ∘ top))))
     ∷ []
```

In the first case, the index is fixed at 0. The second case declares that to construct a vector of length `suc ∘ top`, the recursive field must have length `top`. Note that unlike index-first types, we cannot know the expected index from inside the description, so much like native indexed types, we must add a field choosing an index.

Recall the type of finger trees. Using parameter transformations and composition, we can give a description of full-fledges finger trees! First, we describe the digits

```
DigitD : Desc (∅ ▷ const Type) ⊤
DigitD = σ- (par top) (𝟙 _)
       ∷ σ- (par top) (σ- (par top) (𝟙 _))
       ∷ σ- (par top) (σ- (par top) (σ- (par top) (𝟙 _)))
       ∷ []
```

and define the nodes[8]

```
data Node (A : Type) : Type where
  two   : A → A     → Node A
  three : A → A → A → Node A
```

We encode finger trees as

```
FingerD : Desc (∅ ▷ const Type) ⊤
FingerD = 𝟙 _
        ∷ σ- (par top) (𝟙 _)
        ∷ δ- _ (par ((tt ,_) ∘ top)) DigitD
        ( ρ _ (λ { (_ , A) → (_ , Node A) })
        ( δ- _ (par ((tt ,_) ∘ top)) DigitD (𝟙 _)))
        ∷ []
```

In the third case, we have digits which are passed the parameters on both sides in composite fields, and a recursive field in the middle. The recursive field has a parameter transformation, turning the type parameter `A` into a `Node A` in the recursive child.

6 +The ornaments

Now that we have descriptions, we can start relating them. We construct ornaments as a binary relation on descriptions

[8]We could give the nodes as a description, but in this case we only use them in the recursive fields, so we would take the fixpoint without looking at their description anyway.

```
data Orn {If} {If′} (f : Cxf Δ Γ) (e : K → J)
         : DescI If Γ J → DescI If′ Δ K → Type
```

which encode proofs that "`E` is a more informative variant of `D`". This means that we can always convert the parameters and indices of `E` into those of `D`, as witnessed by `f` and `e`. Furthermore, we also have to be able to convert[9] values of `E` to `D`

```
ornForget : {f : Cxf Δ Γ} {e : K → J} {D : DescI If Γ J} {E : DescI If′ Δ K}
          → Orn f e D E → ∀ p {i} → μ E p i → μ D (f p) (e i)
```

We will walk through the constructor ornaments

```
data ConOrn {If} {If′} {c : Cxf Δ Γ} (f : VxfO c W V) (e : K → J)
            : ConI If Γ J V → ConI If′ Δ K W → Type where
```

again, an ornament between datatypes is just a list of ornaments between their constructors

```
data Orn f e where
  []  : Orn f e [] []
  _∷_ : ∀ {D E D' E'}
      → ConOrn {c = f} id e D' E'
      → Orn f e D E
      → Orn f e (D' ∷ D) (E' ∷ E)
```

Note that all ornaments completely ignore information bundles! They cannot affect the existence of `ornForget` after all.

Copying parts from one description to another, up to parameter and index refinement, corresponds to reflexivity. Preservation of leaves follows the rule

```
𝟙 : ∀ {k j}
  → (∀ p → e (k p) ≡ j (over f p))
  → ∀ {if if′}
  → ConOrn f e (𝟙 {if = if} j) (𝟙 {if = if′} k)
```

We can see that this commuting square (`e (k p) ≡ j (over f p)`) is necessary: take a value of `E` at `p, i`, where `i` is given as `k p`. Then `ornForget` has to convert this to a value of `D` at `f p , e i`, but since `e i` must match `j (f p)`, this is only possible if `e (k p) = j (f p)`.

Preserving a recursive field similarly requires a square of indices and conversions to commute

```
ρ : ∀ {k j g h D E}
  → ConOrn f e D E
  → (∀ p → g (c p) ≡ c (h p))
  → (∀ p → e (k p) ≡ j (over f p))
  → ∀ {if if′}
  → ConOrn f e (ρ {if = if} j g D) (ρ {if = if′} k h E)
```

additionally requiring the recursive parameters to commute with the conversion.

Preservation of non-recursive fields and description fields is analogous

```
σ : ∀ {S} {V'} {W'} {D : ConI If Γ J V'} {E : ConI If′ Δ K W'}
    {g : Vxf Γ (V ▷ S) _} {h : Vxf Δ (W ▷ (S ∘ over f)) _}
```

Does adding the derivations for the squares everywhere make this section clearer?

[9] I would love to require this conversion to be epi, but we add a field of the empty type.

```
  → (f' : VxfO c W' V')
  → ConOrn f' e D E
  → (∀ {p'} (p : ⟦ W ▷ (S ∘ over f) ⟧tel p') → g (VxfO-▷ f S p) ≡ f' (h p))
  → ∀ {if if′}
  → ConOrn f e (σ S {if = if} g D) (σ (S ∘ over f) {if = if′} h E)

δ : ∀ {R : DescI If″ Θ L} {V'} {W'}
      {D : ConI If Γ J V'} {E : ConI If′ Δ K W'}
      {j : Γ & V ⊢ L} {k} {g : Vxf Γ _ _} {h : Vxf Δ _ _} {f' : VxfO c _ _}
  → ConOrn f' e D E
  → ( ∀ {p'} (p : ⟦ W ▷ liftM2 (μ R) (k ∘ over f) (j ∘ over f) ⟧tel p')
    → g (VxfO-▷ f (liftM2 (μ R) k j) p) ≡ f' (h p))
  → ∀ {if if′}
  → ∀ {iff iff′}
  → ConOrn f e (δ {if = if} {iff = iff} j k R g D)
               (δ {if = if′} {iff = iff′} (j ∘ over f) (k ∘ over f) R h E)
```

differing only in that non-recursive fields appears transformed on the right hand, while description fields have their conversions modified instead. For this rule, we need that the variable transformations fit into a commuting square with the parameter conversions. The condition on indices for descriptions, which is a commuting triangle, is encoded in the return type[10].

Ornaments would not be very interesting if they only related identical structures. The left-hand side can also have more fields than the right-hand side, in which case `ornForget` will simply drop the fields which have no counterpart on the right-hand side. As a consequence, the description extending rules have fewer conditions than the description preserving rules:

```
Δρ : ∀ {D : ConI If Γ J V} {E} {k} {h}
   → ConOrn f e D E
   → ∀ {if}
   → ConOrn f e D (ρ {if = if} k h E)
```

Note that this extension[11] with a recursive field has no conditions.

Extending by a non-recursive field or a description field again only requires the variable transform to interact well with the parameter conversion

```
Δσ : ∀ {W'} {S} {D : ConI If Γ J V} {E : ConI If′ Δ K W'}
   → (f' : VxfO c _ _) → {h : Vxf Δ _ _}
   → ConOrn f' e D E
   → (∀ {p'} (p : ⟦ W ▷ S ⟧tel p') → f (p .proj₁) ≡ f' (h p))
   → ∀ {if′}
   → ConOrn f e D (σ S {if = if′} h E)

Δδ : ∀ {W'} {R : DescI If″ Θ L} {D : ConI If Γ J V} {E : ConI If′ Δ K W'}
       {f' : VxfO c _ _} {m} {k} {h : Vxf Δ _ _}
   → ConOrn f' e D E
```

[10]Should this become a problem like with ρ, modifying the rule to require a triangle is trivial.

[11]Kind of breaking the "ornaments relate types with similar recursive structure" interpretation.

```
  → (∀ {p'} (p : ⟦ W ▷ liftM2 (μ R) m k ⟧tel p') → f (p .proj₁) ≡ f' (h p))
  → ∀ {if′ iff′}
  → ConOrn f e D (δ {if = if′} {iff = iff′} k m R h E)
```

In the other direction, the left-hand side can also omit a field which appears on the right-hand side, provided we can produce a default value

```
∇σ : ∀ {S} {V'} {D : ConI If Γ J V'} {E : ConI If′ Δ K W} {g : Vxf Γ _ _}
  → (s : V ⊧ S)
  → ConOrn ((g ∘ ⊧-▷ s) ∘ f) e D E
  → ∀ {if}
  → ConOrn f e (σ S {if = if} g D) E

∇δ : ∀ {R : DescI If″ Θ L} {V'} {D : ConI If Γ J V'} {E : ConI If′ Δ K W}
     {m} {k} {g : Vxf Γ _ _}
  → (s : V ⊧ liftM2 (μ R) m k)
  → ConOrn ((g ∘ ⊧-▷ s) ∘ f) e D E
  → ∀ {if iff}
  → ConOrn f e (δ {if = if} {iff = iff} k m R g D) E
```

These rules let us describe the basic set of ornaments between datatypes.

Intuitively we also expect a conversion to exist when two constructors have description fields which are not equal, but are only related by an ornament. Such a composition of ornaments takes two ornaments, one between the field, and one between the outer descriptions. This composition rule reads:

```
∙δ : ∀ {Θ Λ M L W' V'} {D : ConI If Γ J V'} {E : ConI If′ Δ K W'}
     {R : DescI If″ Θ L} {R' : DescI If‴ Λ M} {c' : Cxf Λ Θ} {e' : M → L}
     {f'' : VxfO c W' V'} {fΘ : V ⊢ ⟦ Θ ⟧tel tt} {fΛ : W ⊢ ⟦ Λ ⟧tel tt}
     {l : V ⊢ L} {m : W ⊢ M} {g : Vxf _ (V ▷ _) V'} {h : Vxf _ (W ▷ _) W'}
  → ConOrn f'' e D E
  → (O : Orn c' e' R R')
  → (p₁ : ∀ q w → c' (fΛ (q , w)) ≡ fΘ (c q , f w))
  → (p₂ : ∀ q w → e' (m (q , w)) ≡ l (c q , f w))
  → ( ∀ {p'} (p : ⟦ W ▷ liftM2 (μ R') fΛ m ⟧tel p') → f'' (h p)
       ≡ g (VxfO-▷-map f (liftM2 (μ R) fΘ l) (liftM2 (μ R') fΛ m)
           (λ q w x → subst2 (μ R) (p₁ _ _) (p₂ _ _)
                  (ornForget O (fΛ (q , w)) x)) p))
  → ∀ {if if′}
  → ∀ {iff iff′}
  → ConOrn f e (δ {if = if}  {iff = iff}  l fΘ R  g D)
               (δ {if = if′} {iff = iff′} m fΛ R' h E)
```

The implicits kind of get out of control here, but the rule is also unreadable without them. I might hide the rule altogether and only run an example with it.

We first require two commuting squares, one relating the parameters of the fields to the inner and outer parameter conversions, and one relating the indices of the fields to the inner index conversion and the outer parameter conversion. Then, the last square has a rather complicated equation, which merely states that the variable transforms for the remainder respect the outer parameter conversion.

We will construct `ornForget` as a `fold`. Using

```
pre₂ : (A → B → C) → (X → A) → (Y → B) → X → Y → C
```

```
pre₂ f a b x y = f (a x) (b y)

erase : ∀ {D : DescI If Γ J} {E : DescI If′ Δ K} {f} {e} {X : PIType Γ J}
      → Orn f e D E → ∀ p k → ⟦ E ⟧ (pre₂ X f e) p k → ⟦ D ⟧ X (f p) (e k)
```
we can define the algebra which forgets the added structure of the outer layer
```
ornAlg : ∀ {D : DescI If Γ J} {E : DescI If′ Δ K} {f} {e}
       → Orn f e D E
       → ⟦ E ⟧ (λ p k → μ D (f p) (e k)) ⇉ λ p k → μ D (f p) (e k)
ornAlg O p k x = con (erase O p k x)
```
Folding over this algebra gives the wanted function
```
ornForget O p = fold (ornAlg O) p _
```
Now we can show that the descriptions we gave in Section 5 are related. The ornament between naturals and lists is
```
NatD-ListD : Orn ! id NatD ListD
NatD-ListD = 𝟙 (const refl)
           ∷ Δσ _
           ( ρ (𝟙 (const refl)) (const refl) (const refl))
           (const refl)
           ∷ []
```
We use `!` to convert parameters, naturals have no parameters, so we can map every parameter of lists to the empty sequence. The index conversion is `id`, since neither type has an index. All structure is preserved; we just have to note that lists have an added field using `Δσ`, and all commutativity squares are trivial, since naturals have neither parameters nor indices.

We can also relate lists and vectors
```
ListD-VecD : Orn id ! ListD VecD
ListD-VecD = 𝟙 (const refl)
           ∷ σ id
           ( Δσ _
           ( ρ (𝟙 (const refl)) (λ p → refl) (const refl))
           λ p → refl)
           (const refl)
           ∷ []
```
Now the parameter conversion is the identity, since both have a single type parameter. The index conversion is `!`, since lists have no indices. Again, most structure is preserved, we only note that vectors have an added field carrying the length.

Instantiating `ornForget` to these ornaments, we now get the functions `length` and `toList` for free!

7 +Ornamental descriptions

A description can say "this is how you make this datatype", an ornament can say "this is how you go between these types". However, an ornament needs its left-hand side to be predefined before it can express the relation, while we might also interpret an ornament as a set of instructions to translate one description into another. A slight variation on ornaments can make this kind of usage possible: ornamental descriptions.

An ornamental description drops the left-hand side when compared to an ornament, and interprets the remaining right-hand side as the starting point of the new datatype:

```
data ConOrnDesc {If} (If′ : Info) {Γ} {Δ} {c : Cxf Δ Γ}
                {W} {V} {K} {J} (f : VxfO c W V) (e : K → J)
                : ConI If Γ J V → Type
```

The definition of ornamental descriptions can be derived in a straightforward manner from ornaments, removing all mentions of the LHS and making all fields which then no longer appear in the indices explicit[12]. We will show the leaf-preserving rule as an example, the others are derived analogously:

```
𝟙 : ∀ {j} (k : Δ & W ⊢ K)
  → (∀ p → e (k p) ≡ j (over f p))
  → ∀ {if} {if′ : If′ .𝟙i}
  → ConOrnDesc If′ f e (𝟙 {if = if} j)
```

As we can see, the only change we need to make is that `k` becomes explicit and fully annotated.

Almost by construction, we have that an ornamental description can be decomposed into a description of the new datatype

```
toDesc : {f : Cxf Δ Γ} {e : K → J} {D : DescI If Γ J}
       → OrnDesc If′ Δ f K e D → DescI If′ Δ K

toCon  : {c : Cxf Δ Γ} {f : VxfO c W V} {e : K → J} {D : ConI If Γ J V}
       → ConOrnDesc If′ f e D → ConI If′ Δ K W
```

and an ornament between the starting description and this new description

```
toOrn : {f : Cxf Δ Γ} {e : K → J} {D : DescI If Γ J}
        (OD : OrnDesc If′ Δ f K e D) → Orn f e D (toDesc OD)

toConOrn : {c : Cxf Δ Γ} {f : VxfO c W V} {e : K → J} {D : ConI If Γ J V}
           (OD : ConOrnDesc If′ f e D) → ConOrn f e D (toCon OD)
```

8 Temporary: future work

Remark 8.1. Note that this allows us to express datatypes like finger trees, but not rose trees. Such datatypes would need a way to place a functor "around the ρ", which then also requires a description of strictly positive functors. In our setup, this could only be encoded by separating general descriptions from strictly positive descriptions. The non-recursive fields of these strictly positive descriptions then need to be restricted to only allow compositions of strictly positive context functions.

Remark 8.2. Variable transforms are not essential in these descriptions, but there are a couple of reasons for keeping them. In particular, they make it

[12] One might expect to need less equalities, alas, this is difficult because of Remark 8.4.

possible to reuse a description in multiple contexts, and save us from writing complex expressions in the indices of our ornaments. On the other hand, the transforms still make defining ornaments harder (the majority of the commuting squares are from variables). Isolating them into a single constructor of `Desc`, call it `v`, seems like a good middle ground, but raises some odd questions, like "why is there no ornament between `v (g ∘ f) C` and `v g (v f C)`". (Furthermore, this also does not simplify the indices of ornaments).

Remark 8.3. Rather, ornaments themselves could act as information bundles. If there was a description for `Desc`, that is. Such a scheme of levitation would make it easier to switch between being able to actively manipulate information, and not having to interact with it at all. However, the complexity of our descriptions makes this a non-trivial task; since our `Desc` is given by mutual recursion and induction-recursion, the descriptions, and the ornaments, would have to be amended to encode both forms of recursion as well.

Remark 8.4. Rather than having the user provide two indices and show that the square commutes, we can ask for a "lift" k

$$\begin{tikzcd}
\bullet \arrow[r, "e"] & \bullet \\
\bullet \arrow[u, "j"] \arrow[r, "f"'] & \bullet \arrow[u, "i"'] \arrow[ul, "k"]
\end{tikzcd}$$

and derive the indices as $i = ek, j = kf$. However, this is more restrictive, unless f is a split epi, as only then pairs i, j and arrows k are in bijection. In addition, this makes ornaments harder to work with, because we have to hit the indices definitionally, whereas asking for the square to commute gives us some leeway (i.e., the lift would require the user to transport the ornament).

Part II

Numerical representations

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.

9 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?

[13] 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
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⊎≃ ⟩
```

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

```
    (⊤ → 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[14]

```
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`.[15]

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

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

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

10 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 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.

11 +Generic numerical representations

We will demonstrate how we can use ornamental descriptions to generically construct datastructures. The claim is that calculating a datastructure is actually an ornamental operation, so we might call our approach "calculating ornaments".

We first define the kind of information constituting a type of "natural numbers"
```
Number : Info
Number .𝟙i = ℕ
Number .ρi = ℕ
Number .σi S = ∀ p → S p → ℕ
Number .δi Γ J = Γ ≡ ∅ × J ≡ ⊤ × ℕ
```
which gets its semantics from the conversion to ℕ
```
toℕ : {D : DescI Number Γ ⊤} → ∀ {p} → μ D p tt → ℕ
```
This conversion is defined by generalizing over the inner information bundle and folding using
```
toℕ-desc : (D : DescI If Γ ⊤) → ∀ {a b} → ⟦ D ⟧ (λ _ _ → ℕ) a b → ℕ
toℕ-con : (C : ConI If Γ ⊤ V) → ∀ {a b} → ⟦ C ⟧ (λ _ _ → ℕ) a b → ℕ

toℕ-desc (C ∷ D) (inj₁ x) = toℕ-con C x
toℕ-desc (C ∷ D) (inj₂ y) = toℕ-desc D y

toℕ-con (𝟙 {if = k} j) refl
        = ϕ .𝟙f k

toℕ-con (ρ {if = k} j g C)                (n , x)
        = ϕ .ρf k * n + toℕ-con C x

toℕ-con (σ S {if = S→ℕ} h C)              (s , x)
        = ϕ .σf _ S→ℕ _ s + toℕ-con C x

toℕ-con (δ {if = if} {iff = iff} j g R h C) (r , x)
        with ϕ .δf _ _ if
```

```
...      | refl , refl , k
         = k * toℕ-lift R (ϕ ∘InfoF iff) r + toℕ-con C x
```
Hence, a number can have a list of alternatives, which can be one of

- a leaf with a fixed value k
- a recursive field n and remainder x, which get a value of $kn+x$ for a fixed k
- a non-recursive field, which can add an arbitrary value to the remainder
- a field containing another number r, and a remainder x, which again get a value of $kr+x$ for a fixed k.

This restricts the numbers to the class of numbers which are interpreted by linear functions, which certainly does not include all interesting number systems, but does include almost all systems that have associated containers[16]. Note that an arbitrary number system of this kind is not necessarily isomorphic to ℕ, as the system can still be incomplete (i.e., it cannot express some numbers) or redundant (it has multiple representations of some numbers).

Recall the calculation of vectors from ℕ in Section 9. In this universe, we can encode ℕ and its interpretation as

```
NatND : DescI Number ∅ ⊤
NatND = 𝟙 {if = 0} _
      ∷ ρ0 {if = 1} _ (𝟙 {if = 1} _)
      ∷ []
```
In such a calculation, all we really needed was a translation between the type of numbers, and a type of shapes. This encoding precisely captures all information we need to form such a type of shapes.

The essence of the calculation of arrays is that given a number system, we can calculate a datastructure which still has the same shape, and has the correct number of elements. We can generalize the calculation to all number systems while proving that the shape is preserved by presenting the datastructure by an ornamental description.

We could directly compute indexed array, using the index for the proof of representability, and from it the correctness of numbers of elements. However, we give the unindexed array first: we can get the indexed variant for free [McB14]! no, rewrite this

Conjecture 11.1. *We claim then that the description given by*

```
TrieO : (D : DescI Number ∅ ⊤) → OrnDesc Plain (∅ ▷ const Type) ! ⊤ ! D
```
and the number of elements coincides with the underlying number, as given by `ornForget`*.* Currently, without proof

The hard work of `TrieO` is done by

```
TrieO-con : ∀ {V} {W : ExTel (∅ ▷ const Type)} {f : VxfO ! W V}
            (C : ConI If ∅ ⊤ V) → InfoF If Number
            → ConOrnDesc Plain {W = W} {K = ⊤} f ! C
```

[16] Notably, polynomials still calculate datastructures, interpreting multiplication as precomposition.

Let us walk through the definition of TrieO-Con. Suppose we encounter a leaf of value k

```
TrieO-con {f = f} (𝟙 {if = k} j) ϕ =
  Δσ (λ { ((_ , A) , _) → Vec A (ϕ .𝟙f k)}) f proj₁
  (𝟙 ! (const refl))
  (λ p → refl)
```

then, the trie simply preserves the leaf, and adds a field with a vector of k elements. Trivially the number of elements and the underlying number coincide.

When we encounter a recursive field

```
TrieO-con {f = f} (ρ {if = k} j g C) ϕ =
  ρ ! (λ { (_ , A) → _ , Vec A (ϕ .ρf k) })
  (TrieO-con C ϕ)
  (λ p → refl) λ p → refl
```

we first preserve this field. The formula used is almost identical to the one in the case of a leaf, but because it is in a recursive parameter, it instead acts to multiply the parameter A by k. Using that the number of elements and the underlying number of the recursive field correspond, let this be r, we see that we get r times A^k. Then, we translate the remainder. It follows that we have kr elements from the recursive field, and by the correctness of the remainder, the total number of elements in ρ also corresponds to the underlying number.

The case for a non-recursive field is similar

```
TrieO-con {f = f} (σ S {if = if} h C) ϕ =
  σ S id (h ∘ VxfO-▷ f S)
  (Δσ (λ { ((_ , A) , _ , s) → Vec A (ϕ .σf _ if _ s) }) (h ∘ _) id
  (TrieO-con C ϕ)
  λ p → refl) (λ p → refl)
```

except we preserve the field directly, and add a field containing its value number of elements. Translating the remainder, the number of elements and the underlying number of a σ coincide.

Consider the case of a description field[17]

```
TrieO-con {f = f} (δ {if = if} {iff = iff} j g R h C) ϕ with ϕ .δf _ _ if
... | refl , refl , k =
  •δ
    {f'' = λ { (w , x) → h (f w , ornForget
          (toOrn (TrieO-desc R (ϕ ∘InfoF iff))) _ x) }}
    (λ { ((_ , A) , _) → _ , Vec A k }) !
  (TrieO-con C ϕ)
  (TrieO-desc R (ϕ ∘InfoF iff)) id
  (λ _ _ → refl) (λ _ _ → refl) λ p → refl
```

We essentially rerun the recipe of ρ, multiplying the elements of the field by k, but now pass it to the description `R`. Again, correctness of δ follows directly from the correctness of `R` and the remainder.

Example 11.1.

[17] Excuse the formula of `f''`, it needs to be there for the ornament to work, but doesn't have much to do with the numbers.

This "proves" our construction correct, but let us compare it to an existing numerical representation: We see that applying `TrieO` to `NatND` gives us a description which corresponds almost directly to `ListD`, only replacing all fields with vectors of length 1.

We would like to algOrn, but we can't.

Remark 11.1.

Explain why no algOrn

12 Temporary: future work

Some goals: 1. Ix and paths. 2. Ix n -> A iso IxTrieO n A. 3. something about the correctness of TrieO

This implementation of TrieO always computes the random-access variant of the datastructure. Can we implement a variant which computes the "Braun tree" variant of the datastructure?

Index types are a simple ornament over number types: paths. This is not quite like [DS16].

Is Ix x -> A initial for the algebra of the algebraic ornament induced by TrieO? (This is [HS22]).

While evidently Ix x != Fin (toN x) for arbitrary number systems, does the expected iso Ix x -> A = Trie A x yield Traversable, for free?

Part III
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 [Res19]! 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.

13 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.

14 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.

15 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.

Part IV
Related work

16 Descriptions and ornaments

We compare our implementation to a selection of previous work, considering the following features

	Haskell	[JG07]	[Cha+10]	[McB14]	[KG16]
Fixpoint	yes*	yes	no	yes?	yes
Index	—	—	first**	equality	first
Poly	yes	1	external	external	external
Levels	—	—	no	no	no
Sums	list	—	large	large	large
IndArg	any	any	$\cdots \to X\ i$	$X\ i$	$X\ i$
Compose	yes	yes	no	no	no
Extension	—	—	no	—	—
Ignore	—	—	—	—	—
Set	—	—	—	—	—

	[Sij16]	[eff20]	[EC22]	Shallow	Deep (old)
Fixpoint	yes	yes	no	yes	yes
Index	equality	equality	equality	equality	
Poly	telescope	external	telescope	telescope	
Levels	no***	cumulative	Typeω	Type-in-Type	
Sums	list	large	list	list	
IndArg	$X\ pv\ i$	$\dots \to X\ v\ i$	$\dots \to X\ pv\ i$	$X(fpv)i$	?1
Compose	no	yes?2	no	yes	
Extension	—	yes	yes	no	
Ignore	no	?	?	transform	
Set	no	no	no	no	yes

- IndArg: the allowed shapes of inductive arguments. Note that none other than Haskell, higher-order functors, and potentially ?1, allow full nested types!
- Compose: can a description refer to another description?
- Extension: do inductive arguments and end nodes, and sums and products coincide through a top-level extension?
- Ignore: can subsequent constructor descriptions ignore values of previous ones? (Either this, or thinnings, are essential to make composites work)
- Set: are sets internalized in this description?

* These descriptions are "coinductive" in that they can contain themselves, so the "fixpoint" is more like a deep interpretation.

** This has no fixpoint, and the generalization over the index is external.

*** But you could bump the parameter telescope to Typeω and lose nothing.

*4 A variant keeps track of the highest level in the index.

?1 Deeply encoding all involved functors would remove the need for positivity annotations for full nested types like in other implementations.

?2 The "simplicity" of this implementation, where data and constructor descriptions coincide, automatically allows composite descriptions.

We take away some interesting points from this:

- Levels are important, because index-first descriptions are incompatible with "data-cumulativity" when not emulating it using equalities! (This results in datatypes being forced to have fields of a fixed level).
- Coinductive descriptions can generate inductive types!
- Typeω descriptions can generate types of any level!

- Large sums do not reflect Agda (a datatype instantiated from a derived description looks nothing like the original type)! On the other hand, they make lists unnecessary, and simplify the definition of ornaments as well.
- We can group/collapse multiple signatures into one using tags, this might be nice for defining generic functions in a more collected way.
- Everything becomes completely unreadable without opacity.

16.1 Merge me

16.1.1 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 instructions 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].

16.1.2 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].

16.2 Takeways

At the very least, descriptions will need sums, products, and recursive positions as well. While we could use coinductive descriptions, bringing normal and recursive fields to the same level, we avoid this as it also makes ornaments a bit more wild[18]. We represent indexed types by parametrizing over a type I. Since we are aiming for nested types, external polymorphism[19] does not suffice: we need to let descriptions control their contexts.

We describe parameters by defining descriptions relative to a context. Here, a context is a telescope of types, where each type can depend on all preceding types:

...

[18] For better or worse, an ornament could refer to a different ornament for a recursive field.
[19] E.g., for each type A a description of lists of A à la [KG16]

Much like the work Escot and Cockx [EC22] we shove everything into `Typeω`, but we do not (yet) allow parameters to depend on previous values, or indices on parameters[20].

We use equalities to enforce indices, simply because index-first types are not honest about being finite, and consequently mess up our levels. For an index type and a context a description represents a list of constructors:

...

These represent lists of alternative constructors, which each represent a list of fields:

...

We separate mere fields from "known" fields, which are given by descriptions rather than arbitrary types. Note that we do not split off fields to another description, as subsequent fields should be able to depend on previous fields

....

We parametrize over the levels, because unlike practical generic, we stay at one level.

Q: what happens when you precompose a datatype with a function? E.g. (List . f) A = List (f A)

Q: practgen is cool, compact, and probably necessary to have all datatypes. Note that in comparison, most other implementations (like Sijsling) do not allow functions as inductive arguments. Reasonably so.

Q: I should probably update my Agda and make use of the new opaque features to make things readable when refining

Adapt this to the non-proposal form.

17 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 independence", 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, *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.

[20] I do not know yet what that would mean for ornaments.

Still, representation independence in may be internalized outside the homotopical setting in some cases [Kap23], and remains of interest in the context of generic constructions that conflict with cubical type theory.

18 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.

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`.

[Cha+10] James Chapman et al. "The Gentle Art of Levitation". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Baltimore, Maryland, USA: Association for Computing Machinery, 2010, pp. 3–14. ISBN: 9781605587943. DOI: `10.1145/1863543.1863547`. URL: `https://doi.org/10.1145/1863543.1863547`.

[Cla20] Koen Claessen. "Finger Trees Explained Anew, and Slightly Simplified (Functional Pearl)". In: *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. Haskell 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 31–38. ISBN: 9781450380508. DOI: `10.1145/3406088.3409026`. URL: `https://doi.org/10.1145/3406088.3409026`.

[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.

[eff20] effectfully. *Generic*. 2020. URL: https://github.com/effectfully/Generic.

[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.

[JG07] Patricia Johann and Neil Ghani. "Initial Algebra Semantics Is Enough!" In: *Typed Lambda Calculi and Applications*. Ed. by Simona Ronchi Della Rocca. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 207–222. ISBN: 978-3-540-73228-0.

[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.

[KT95] Haim Kaplan and Robert E. Tarjan. "Persistent Lists with Catenation via Recursive Slow-Down". In: *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*. STOC '95. Las Vegas, Nevada, USA: Association for Computing Machinery, 1995, pp. 93–102. ISBN: 0897917189. DOI: 10.1145/225058.225090. URL: https://doi.org/10.1145/225058.225090.

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

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

[Res19] Cas van der Rest. "Generating Constrained Test Data using Datatype Generic Programming". In: *Master's thesis* (2019).

[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`.

[Sij16] Yorick Sijsling. “Generic programming with ornaments and dependent types”. In: *Master’s thesis* (2016).

[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 V
Appendix

A Finger trees

We know that some datastructures can be presented as non-redundant numerical representations, for example lists by unary numbers, random access lists by binary numbers [HS22], and, skew binary heaps by skew binary numbers [KG16]. So far, some of these examples do support amortized constant time `cons`, but they have at best logarithmic time `snoc`. This is reflected by their number systems, for which either the natural successor operation is logarithmic time, or is constant time, but can only act at the front. Instead, we will look at more redundant number systems, and refine these step-by-step to produce structures similar to finger trees. This gives us datastructures with fast access to both ends, and some of their properties for free.

A.1 Binary finger trees

If a datastructure has a numerical representation, we see that the operations on the datastructure must be coherent with the number system. Hence, if we want to have constant time `cons` and `snoc`, we must first have constant time `suc` anc `cus`. By starting from a symmetric number system, we can ensure good performance for both.

Note that such a system is necessarily redundant: if `suc` and `cus` both are amortized constant time, there must be cases where neither recurses (otherwise, there is a value and a sequence of sucs and cuss which cannot be amortized constant). On the other hand, both must clearly yield different values!

Symmetric unary numbers could be represented by a pair of Peano naturals, but would lead to a linear time lookup. By using a binary backbone for the numbers, we can get good suc and lookup

```
data Digit : Set where
  𝟙 𝟚 : Digit

data Bin : Set where
  𝟘 𝟙 : Bin
  _⟨_⟩_ : Digit → Bin → Digit → Bin
```

However, this shape is still not ideal. We can see that for values like

```
bad-1 : Bin
bad-1 = 𝟚 ⟨ 𝟚 ⟨ 𝟚 ⟨ 𝟙 ⟩ 𝟙 ⟩ 𝟙 ⟩ 𝟙
```

applying suc would give

```
bad-2 : Bin
bad-2 = 𝟙 ⟨ 𝟙 ⟨ 𝟙 ⟨ 𝟙 ⟨ 𝟘 ⟩ 𝟙 ⟩ 𝟙 ⟩ 𝟙 ⟩ 𝟙
```

Applying pred would take us back, so composing the two always takes logarithmic time [Cla20]. To avoid this, we can give the numbers bigger digits (the system merely goes from redundant to slightly more redundant)

```
data Digit : Set where
  𝟙 𝟚 𝟛 : Digit

data Bin : Set where
  𝟘 𝟙 : Bin
  _⟨_⟩_ : Digit → Bin → Digit → Bin
```

Now applying suc to the pathological case

```
good-1 : Bin
good-1 = 𝟛 ⟨ 𝟛 ⟨ 𝟛 ⟨ 𝟙 ⟩ 𝟙 ⟩ 𝟙 ⟩ 𝟙
```

produces

```
good-2 : Bin
good-2 = 𝟚 ⟨ 𝟚 ⟨ 𝟚 ⟨ 𝟙 ⟨ 𝟘 ⟩ 𝟙 ⟩ 𝟙 ⟩ 𝟙 ⟩ 𝟙
```

instead, for which both suc and pred are constant time [21]. We interpret this number system as

```
⟦_⟧D : Digit → ℕ
⟦ 𝟙 ⟧D = 1
⟦ 𝟚 ⟧D = 2
⟦ 𝟛 ⟧D = 3

⟦_⟧B : Bin → ℕ
⟦ 𝟘 ⟧B = 0
⟦ 𝟙 ⟧B = 1
⟦ l ⟨ m ⟩ r ⟧B = ⟦ l ⟧D + 2 * ⟦ m ⟧B + ⟦ r ⟧D
```

[21] More formally, we can use recursive slowdown [Oka98; KT95] to show that any sequence of operations amortizes to constant time.

To extract the datastructure, we must find a suitable index type for these numbers. Since the numbers are redundant, we can also get trees of different shapes with the same size, each having a different and incompatible index type. However, the trees of a fixed shape are represented by functions, and the isomorphisms will still hold.

The computation of the index type from the interpretation of the numbers is straightforward. We first compute the indices for digits, which yields the indices for the numbers

```
data IxD : Digit → Set where
  𝟙-1     : IxD 𝟙
  𝟚-1 𝟚-2 : IxD 𝟚
  𝟛-1 𝟛-2 𝟛-3 : IxD 𝟛

data IxB : Bin → Set where
  𝟙-1 : IxB 𝟙
  ⟨⟩-l : IxD d → IxB (d ⟨ n ⟩ e)
  ⟨⟩-m1 : IxB n → IxB (d ⟨ n ⟩ e)
  ⟨⟩-m2 : IxB n → IxB (d ⟨ n ⟩ e)
  ⟨⟩-r : IxD e → IxB (d ⟨ n ⟩ e)
```

This represents simple finger trees as

```
Array : Set → Bin → Set
Array A b = IxB b → A
```

To define the basic array operations like `cons` on these functions as datastructures, we again construct a `Fin`-like view for the indices. For this we produce values corresponding to zero

```
izero : ∀ {n} → IxB (succ n)
```

and induce the successor on the indices using

```
isucc : IxB n → IxB (succ n)
```

The view is similarly defined by

```
data IxV : IxB (succ n) → Set where
  as-izero : IxV {n} izero
  as-isucc : (i : IxB n) → IxV (isucc i)

iview : {n : Bin} → (i : IxB (succ n)) → IxV i
```

letting us define

```
head : Array A (succ n) → A
head {n = 𝟘} f = f 𝟙-1
head {n = 𝟙} f = f (⟨⟩-l 𝟙-1)
head {n = 𝟙 ⟨ m ⟩ r} f = f (⟨⟩-l 𝟚-1)
head {n = 𝟚 ⟨ m ⟩ r} f = f (⟨⟩-l 𝟛-1)
head {n = 𝟛 ⟨ m ⟩ r} f = f (⟨⟩-l 𝟚-1)

cons : A → Array A n → Array A (succ n)
cons {n = n} x f i with iview i
... | as-izero = x
... | as-isucc i = f i
```

We can again trieify this to get a concrete datastructure[22]

```
data Finger (A : Set) : Digit → Set where
  𝟙 : A → Finger A 𝟙
  𝟚 : A → A → Finger A 𝟚
  𝟛 : A → A → A → Finger A 𝟛

data Array′ (A : Set) : Bin → Set where
  𝟘 : Array′ A 𝟘
  𝟙 : A → Array′ A 𝟙
  _⟨_⟩_ : Finger A d → Array′ A n → Array′ A n → Finger A e → Array′ A (d ⟨ n ⟩ e)
```

Consequently, the concrete version will now obey all the relations the representable arrays obey as well. For example, for representable arrays we can easily see

```
(x : A) (xs : Array A n) → head (cons x xs) ≡ x
```

hence, the concrete arrays obey this as well.

On the other hand, as

```
∀ n → succ (cuss n) ≡ cuss (succ n)
```

does not generally hold for symmetric binary, `cons` will not interchange with `snoc` for finger trees either[23]; it seems that binary finger trees are not a very nice array type. Likewise, indexing into the finger trees is impractical, as changing shapes would require inefficient re-indexing.

A.2 Restoring efficient lookup

Can we restore lookup? We can probably do something similar to the original finger trees, and maintain the sizes internally (hopelessly breaking the isomorphism[24]). Then we could state that a fingertree of a given size is just a finger tree of a shape paired with a proof that this shape has the right size.

B Heterogenization

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

[22] I'll probably not do this manually, because it is theoretically analogous to the other trees, but hellish in practice

[23] For starters, the types are different

[24] Or would it stay intact, since the shape determines the size anyway?

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 heterogenized 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 , _ , _)
```

C More equivalences for less effort

Noting that constructing equivalences directly or from isomorphisms as in Subsection 4.3 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 4.3 were partially forced, but this fact was unused there.

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 4.3, 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 corollary.

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 derivation in Subsection 4.3, but we argue that most of this code can likely be automated.

Merge

Using Subsection 4.3 we can relate functionally equivalent structures, and using Section 9 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 4.3, 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[25] 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′`.

C.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
  ү : (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 type computing function and not a datatype, this descent becomes invalid[26],

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

[26] Refer to the without K page.

and `mapFold` fails the termination check. We resolve this by defining `Base` as a datatype

```
data Base (X : Set₁) : Desc′ → Set₁ where
  in-γ : ∀ {n} → Vec X n → Base X (γ 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.[27]

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.

C.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.[28]

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

[27] 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.

[28] 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.

C.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 C.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 4.3, 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 `ɣ 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-ɣ : All (Acc D f) x → Acc' D f (ɣ n) (in-ɣ 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 C.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 C.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"[29], 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
```

□

C.1.3 Example

Let us redo the proof in Subsection 4.3, 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)
```

[29] 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.