Ornaments and Proof Transport applied to Numerical Representations

Samuel Klumpers
6057314

August 21, 2023

Abstract

This thesis explains the concepts numerical representations and ornaments, and aims to combine these to simplify the presentation and verification of finger trees. We demonstrate the generalizability and easier verification of the resulting code. Further, we also investigate to which extent descriptions and ornaments, and generic programs built on top of these, remain effective in a setting without axiom K.

Contents

Todo list

1 Introduction

Programming is hard, but using the right tools can make it easier. Logically, much time and effort goes into creating such tools. Because it hard to memorize the documentation of a library, we have code suggestion; to read code more easily, we have code highlighting; to write tidy code, we have linters and formatters; to make sure code does what we hope it does, we use testing; to easily access the right tool for each of the above, we have IDEs.

In this thesis, we look at how we can make written code more easy to verify and to reuse, or even to generate from scratch. We hope that this lets us spend more time on writing code rather than tests, spend less time repeating similar work, and save time by writing more powerful code.

We use the language Agda [Tea23], of which the dependent types form the logic we use to specify and verify the code we write. In our approach, we describe a part of the language inside the language itself. This allows us to reason about the structure of other code using code itself. Such descriptions of code can then be interpreted to generate usable code. Using constructions known as ornaments [McB14; Sij16], we can also discuss how we can transform one piece of code into another by comparing the descriptions of the two pieces.

We will describe and then generate a class of container types (which are types that contain elements of other types) from number systems. The idea is that some container types "look like" a number system by squinting a bit. Consequently, types of that class of containers are known as numerical representations [Oka98]. This leads us to our research question:

Can numerical representations be described as ornaments on number systems, and how does this make generating them and verifying their properties easier?

Generating numerical representations is closely related to calculating datastructures [HS22]. As an example, one can calculate the definition of a random-access list by applying a chain of type isomorphisms to the representable container, which is defined by the lookup function from (Leibniz or bijective base-2) binary numbers. Likewise, ornaments and their applications to numerical representations have been studied before, describing binomial heaps as an ornament on (ordinary) binary numbers [KG16]. The underlying descriptions in this approach correspond roughly to the indexed polynomial endofunctors on the type of types. We also know that we can use the algebraic structure arising from ornaments to construct different, algebraic, ornaments [McB14]. In an example this is used to obtain a description of vectors with an ornament from lists.

We seek to expand upon these developments by generating the numerical representation from a number system, collecting the instances of calculated datastructures under one generic calculation. However, we cannot formulate this as an ornamental operation in most existing frameworks, which are based on indexed polynomial endofunctors. Namely, nested datatypes, such as the random-access list mentioned above, cannot be directly represented by such functors[1]. Furthermore, these calculations target indexed containers, while the algebras arising from ornaments suggest that we only have to make an ornament to the unindexed containers, which yields the indexed containers by the algebraic ornament construction.

Our contribution will be to rework part of the existing theory and techniques of descriptions and ornaments to comfortably fit a class of number systems and numerical representations into this theory, which then also encompasses nested datatypes. We will then use this to formalize the construction of numerical representations from their number systems as an ornament.

To make the research question formal, we first need to properly define the concepts of descriptions and ornaments.

Background

We extend upon and make heavy use of existing work for generic programming and ornaments, so let us take a closer look at the nuts and bolts to see what all the concepts are about.

[1]Maybe refer to the appendix, to which extent you can squash datatypes.

2 Agda

•Note to self: add more citations and then double check them below here •Also, type in type: we know that nothing breaks with universe polymorphism [EC22], but it does not make things easier to read. We formalize all of our work in the programming language Agda [Tea23]. While we will only occasionally reference Haskell, those more familiar with Haskell may look at Agda as an extension of Haskell (in a rough approximation). Agda is a total functional programming language with dependent types, where totality means that functions of a given type always terminate in a value of that type, ruling out non-terminating (and not obviously terminating) programs. Using the dependent types we can use Agda as a proof assistant, allowing us to state and prove theorems about our datastructures and programs.

In this section, we will explain and highlight some parts of Agda which will be necessary to discuss the work in later sections.

3 Data in Agda

At the level of ordinary (i.e., generalized algebraic) datatypes, Agda is close to Haskell. In both languages, one can define objects using data declarations, and interact with them using function declarations. For example,

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

The constructors of this type tell us we can make values of `Bool` in exactly two ways, `false` and `true`. We can then define (non-trivial) functions on `Bool` by pattern matching. As an example, we can define the conditional operator as

```
if_then_else_ : Bool → A → A → A
if false then t else e = e
if true  then t else e = t
```

When pattern matching, we must provide definitions for each possible case of the pattern matched type[2].

We can also define a type representing numbers as a datatype

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

such that `ℕ` always has a `zero` element, and for each element n the constructor `suc` expresses that there is also an element representing $n + 1$. In conclusion, `ℕ` represents the positive integers, henceforth naturals. Similarly, we can pattern match on `ℕ` to define the comparison operator

```
_<_ : (n m : ℕ) → Bool
n     < zero  = false
zero  < suc m = true
suc n < suc m = n < m
```

[2]This is enforced by the coverage checker.

Now one of the cases contains a recursive instance of `ℕ`, the coverage checker and termination checker ensure that we still define $n < m$ for all possible combinations of n and m. Essentially, these checks make sure that any valid definition by pattern matching corresponds to a valid proof by cases and induction[3].

We could now directly define lists of natural numbers, but then we would have to also repeat this for lists of booleans, and all other types. Using parameters, we can define lists as a family of types instead:

```
data List (A : Type) : Type where
  []  : List A
  _∷_ : A → List A → List A
```

defining a type of lists for each type. Introducing a parameter in a datatype corresponds to abstracting the body of the type over a free variable, by applying this abstraction one gets a type for each value of the parameter. We can work with lists by inserting some elements, and extracting them later:

```
lookup? : List A → ℕ → Maybe A
lookup? []       n       = nothing
lookup? (x ∷ xs) zero    = just x
lookup? (x ∷ xs) (suc n) = lookup? xs n
```

However, the type of this function is a bit weaker than we might like, as we are relying on the type

```
data Maybe (A : Type) : Type where
  nothing : Maybe A
  just    : A → Maybe A
```

to handle the case where the index falls outside the list, and we cannot return an element. Note that checking whether

```
length : List A → ℕ
length []       = zero
length (x ∷ xs) = suc (length xs)
```

satisfies `n < length xs` beforehand amounts to the same, and only gives us `false` when we would otherwise get `nothing`.

If we know the length of the list, then we also know for which indices `lookup` will succeed, and for which it will not. To encode and use this information we will need indexed types. Like parameters, indices add a variable to the context of a datatype, but unlike parameters, indices can influence the availability of constructors. This difference can be used to constrain the possible values of a variable from the type level. As an example, we can index `Bool` over itself:

```
data HBool : Bool → Type where
  hfalse : HBool false
  htrue  : HBool true
```

This defines a singleton type, in which knowing the type of a variable of uniquely fixes its value: `HBool true` can only be `htrue`, while `HBool false` can only be `hfalse`.

[3] Also note that in this case we actually have a simultaneous induction on two arguments. This is tolerated when the arguments decrease lexicographically. In other words, a simultaneous induction is valid iff the clauses can be permuted to become a nested sequence of valid inductions.

Applying this idea to naturals and lists, we create the following pair of types:

```
data Fin : ℕ → Type where
  zero : Fin (suc n)
  suc  : Fin n → Fin (suc n)

data Vec (A : Type) : ℕ → Type where
  []  :             Vec A zero
  _∷_ : A → Vec A n → Vec A (suc n)
```

Here, `Fin zero` contains no elements, and the only vector in `Vec zero` is the empty vector `[]`, containing no elements. Likewise, `Fin (suc n)` contains one more element than `Fin n`, and a vector of type `Vec (suc n)` is necessarily a vector `Vec n` with one more element. From this we conclude that `Fin n` is exactly the type of suitable indices into a `Vec n`. Note that we can still interconvert between lists and vectors, in one direction dropping the index

```
toList : Vec A n → List A
toList []       = []
toList (x ∷ xs) = x ∷ toList xs
```

Dependent functions let us bind variables in type signatures, and use them in later types. With them, we can convert in the other direction, recomputing the index:

```
toVec : (xs : List A) → Vec A (length xs)
toVec []       = []
toVec (x ∷ xs) = x ∷ toVec xs
```

We can also define

```
lookup : ∀ {n} → Vec A n → Fin n → A
lookup (x ∷ xs) zero = x
lookup (x ∷ xs) (suc i) = lookup xs i
```

as a dependent function by letting the size of the index type vary with the length of the vector. The case in which we would return `nothing` for lists, is now actually impossible and can be discarded. Lookup always succeeds for vectors, demonstrating that vectors are correct-by-construction. However, this does not yet prove that `lookup` always returns the right element. We will need some more logic to verify this.

4 Proving in Agda

When defining vectors, we saw how knowing the index of a variable constrains the values it can assume. We can use this to define a datatype in which the index actually determines whether there is any value, or none at all:

```
data _≡_ (a : A) : A → Type where
  refl : a ≡ a
```

Here, we have `refl` of type `Eq a b` if and only if `a` is `b`, whence the `_≡_` symbol. We can use this type to describe the behaviour of functions like `lookup`: If we can insert elements into a vector

```
insert : ∀ {n} → Vec A n → Fin (suc n) → A → Vec A (suc n)
insert xs       zero    y = y ∷ xs
```

```
    insert (x ∷ xs) (suc i) y = x ∷ insert xs i y
```
we can express the correctness of `lookup` as
```
    lookup-insert-type : ∀ {n} → Vec A n → Fin (suc n) → A → Type
    lookup-insert-type xs i x = lookup (insert xs i x) i ≡ x
```
To prove this, we proceed as when defining any other function, by pattern matching as the equivalent of induction. By simultaneous induction on the index and vector of `lookup-insert-type` we get
```
    lookup-insert : ∀ {n} (xs : Vec A n) (i : Fin (suc n)) (y : A)
                  → lookup-insert-type xs i y
    lookup-insert []       zero    y = refl
    lookup-insert (x ∷ xs) zero    y = refl
    lookup-insert (x ∷ xs) (suc i) y = lookup-insert xs i y
```
Other than `_≡_`, we can encode many more logical operations into datatypes. For example, we can encode disjunctions (the logical or operation) as
```
    data _⊎_ A B : Type where
      inj₁ : A → A ⊎ B
      inj₂ : B → A ⊎ B
```
While we generally do not directly use these explicit encodings, we will appeal to the logical content of some types in later sections. The disjunction, together with some other types form a correspondence between types and logic, known as the Curry-Howard isomorphism.

The other components of this isomorphism are as follows. Conjunction (logical and) can be interpreted by[4]
```
    record _×_ A B : Type where
      constructor _,_
      field
        fst : A
        snd : B
```
True and false are represented by
```
    record ⊤ : Type where
      constructor tt
```
and respectively
```
    data ⊥ : Type where
```
Interpreting logical implication as the function type, we also get negation in terms of implication and false
```
    ¬_ : Type → Type
    ¬ A = A → ⊥
```
Quantifiers require dependent types to implement. In
```
    record Σ A (P : A → Type) : Type where
      constructor _,_
      field
        fst : A
        snd : P fst
```

[4]A (inductive) record type is almost equivalent to a datatype with one constructor. One advantage of records is that, e.g., $(a, b) = (c, d) \iff a = c \wedge b = d$ holds automatically.

if `P` is a formula containing a variable of type `A`, the existential quantifier represents that there is an element `fst` of `A` for which `P a` is true. The universal quantifier

```
data ∀′ A (P : A → Type) : Type where
  all : (∀ a → P a) → ∀′ A P
```

represents that for each `a` of type `A` the formula `P a` is true.

More generally, we can read function types as chains of implications and universal quantifications. Furthermore, datatypes can be read as conjunctions of their constructors, each of which can be seen as a combination of conjunctions (unbound fields) and existentials (bound fields).

5 Numerical representations

In the previous section, we saw that `Vec` equipped with `lookup` and `insert` satisfies a useful container law. In fact, the definitions of `lookup` and `insert`, as well as the proof of `lookup-insert` are almost trivial. From this, we might deduce that `Vec` is quite a natural container for `ℕ`, which we can make formal by comparing `Vec` to the most natural, and trivial, container.

This trivial container, also known as representable, is obtained by reading `lookup` as a prescript:

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

One can see that for `Lookup`, we can define `lookup` as the identity. If a container law is consistent, then `Lookup` will also satisfy it, which we can use to simplify the verification of the same laws for `Vec`, or other containers: if a container is equivalent to `Lookup`, then it automatically is correct as a container.

To describe such equivalences, we import the notion of isomorphisms into types:

```
record Iso A B : Type where
  constructor iso
  field
    fun : A → B
    inv : B → A
    rightInv : ∀ b → fun (inv b) ≡ b
    leftInv  : ∀ a → inv (fun a) ≡ a
```

which expresses that `A` and `B` are isomorphic if we can go back and forth between them, and doing so brings you back where you started[5].

Rather than defining `Vec` "out of the blue" and proving that it is correct or isomorphic to `Lookup`, we can also turn the isomorphism on its head. Instead, we can start from the equation that `Vec` is equivalent to `Lookup`, and then calculate `Vec` as if solving that equation [HS22]. We can also derive `Fin`: the finite type `Fin n` should actually contain `n` elements, and thus be isomorphic to `Σ[ m ∈ ℕ ] m < n`.

[5]Which is equivalent to the other notion of equivalence: one has a conversion $f : A \to B$, and for each `b` in `B` there is exactly one `a` in `A` for which $f(a) = b$.

For the equational reasoning, we will use the "use-as-definition"[6] and equality reasoning notation[7]. As a bonus, this allows us to break up the construction of harder isomorphisms into multiple small and easy-to-read steps. The equalities we will need are

```
⊥-strict : (A → ⊥) → A ≡ ⊥
<-split  : ∀ n → (Σ[ m ∈ ℕ ] m < suc n) ≡ (⊤ ⊎ (Σ[ m ∈ ℕ ] m < n))
```

We then put these together, constructing Fin:

```
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)
  ≡⟨ <-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)
```

This definition of Fin can then be used to define Lookup. Using[8]

```
⊥→A≡⊤ : (⊥ → A) ≡ ⊤
⊤→A≡A : (⊤ → A) ≡ A
Π⊎≡⊎Π : (A ⊎ B → C) ≡ (A → C) × (B → C)
```

we derive the type of vectors:

```
Vec-def : ∀ A n → Def (Lookup A n)
Vec-def A zero =
     (⊥ → A)
  ≡⟨ ⊥→A≡⊤ ⟩
     ⊤ ■ use-as-def
Vec-def A (suc n) =
     ((⊤ ⊎ Fin n) → A)
  ≡⟨ Π⊎≡⊎Π ⟩
     (⊤ → A) × (Fin n → A)
  ≡⟨ cong₂ _×_
       ⊤→A≡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)
```

[6]Appendix
[7]Agda Cubical, but I think this is readable.
[8]Compare these to the algebra laws: $A^0 = 1$, $A^1 = A$ and $C^{A+B} = C^A C^B$.

6 Descriptions

In the previous sections we completed a quadruple of types (`ℕ`, `List`, `Vec`, `Fin`), even computing the latter two from `ℕ`. These types also have nice interactions (`length`, `toList`, `lookup`). This construction works because `ℕ`, `List`, `Vec`, and `Fin` all have the same shape, and one can expect it to work for similar quadruples. However, shape is not (yet) defined inside our language, so in this section, we will set out to explain the groundwork making this expectation sensible.

Before we can describe the shape of a datatype, we must describe what a datatype is. To this end, we define a type of descriptions. Comparing this to the terminology of Martin-Löf type theory, in which a type of types like `Type` is known as a universe, we can think of a type of descriptions as an internal universe. Each description, or code, should then represent a type via an interpretation $U \to \text{Type}$.

Let us walk through some concrete universe constructions. We will start from a rather simple universe, building our way up to more general types, until we reach a universe in which we can describe the (`ℕ`, `List`, `Vec`, `Fin`) quadruple, which, as a bonus, gives some insight into the meaning of datatypes.

6.1 Finite types

As a start, we define a basic universe with two codes 0 and 1, respectively representing the types `⊥` and `⊤`, and the requirement that the universe is closed under sums and products:

```
data U-fin : Type where
  𝟘 𝟙     : U-fin
  _⊕_ _⊗_ : U-fin → U-fin → U-fin
```

which we interpret as:

```
⟦_⟧fin : U-fin → Type
⟦ 𝟘 ⟧fin = ⊥
⟦ 𝟙 ⟧fin = ⊤
⟦ D ⊕ E ⟧fin = ⟦ D ⟧fin ⊎ ⟦ E ⟧fin
⟦ D ⊗ E ⟧fin = ⟦ D ⟧fin × ⟦ E ⟧fin
```

In this universe, we can encode the type of booleans simply as

```
BoolD : U-fin
BoolD = 𝟙 ⊕ 𝟙
```

The types `𝟘` and `𝟙` are finite, and sums and products of finite types are also finite, which is why we call `U-fin` the universe of finite types. Consequently, the type of naturals `ℕ` cannot fit in `U-fin`.

6.2 Polynomial functors

To accommodate `ℕ`, we need to be able to express recursive types. By adding a code `ρ` to `U-fin` representing recursive type occurrences, we can express those types:

```
data U-rec : Type where
  𝟙 ρ     : U-rec
  _⊕_ _⊗_ : U-rec → U-rec → U-rec
```

However, the interpretation cannot be defined like in the previous example: when interpreting (𝟙 ⊕ ρ), we need to know that the whole type was 𝟙 ⊕ ρ while processing ρ. Thus, we split the interpretation into types in two phases. First, we interpret the descriptions into polynomial functors

```
⟦_⟧rec : U-rec → Type → Type
⟦ 𝟙     ⟧rec X = ⊤
⟦ ρ     ⟧rec X = X
⟦ D ⊕ E ⟧rec X = (⟦ D ⟧rec X) ⊎ (⟦ E ⟧rec X)
⟦ D ⊗ E ⟧rec X = (⟦ D ⟧rec X) × (⟦ E ⟧rec X)
```

By viewing such a functor as a type with a free type variable, the functor can model a recursive type by setting the variable to the type itself:

```
data μ-rec (D : U-rec) : Type where
  con : ⟦ D ⟧rec (μ-rec D) → μ-rec D
```

Recall the definition of ℕ, which can be read as the declaration that ℕ is a fixpoint: `ℕ ≡ F ℕ` for `F X = ⊤ ⊎ X`. This makes representing ℕ as simple as:

```
ℕD : U-rec
ℕD = 𝟙 ⊕ ρ
```

6.3 Sums of products

A downside of `U-rho` is that the definitions of types do not mirror their equivalent definitions in user-written Agda. We can define a similar universe using that polynomials can always be canonically written as sums of products. For this, we split the descriptions into a stage in which we can form sums, on top of a stage where we can form products.

```
data Con-sop : Type
data U-sop : Type where
  [] : U-sop
  _∷_ : Con-sop → U-sop → U-sop
```

When doing this, we can also let the left-hand side of a product be any type, allowing us to represent ordinary fields:

```
data Con-sop where
  𝟙 : Con-sop
  ρ : Con-sop → Con-sop
  σ : (S : Type) → (S → Con-sop) → Con-sop
```

The interpretation of this universe, while analogous to the one in the previous section, is also split into two parts:

```
⟦_⟧U-sop : U-sop → Type → Type
⟦_⟧C-sop : Con-sop → Type → Type

⟦ []  ⟧U-sop X = ⊥
⟦ C ∷ D ⟧U-sop X = ⟦ C ⟧C-sop X × ⟦ D ⟧U-sop X
```

```
  ⟦ 𝟙   ⟧C-sop X = ⊤
  ⟦ ρ C ⟧C-sop X = X × ⟦ C ⟧C-sop X
  ⟦ σ S f ⟧C-sop X = Σ[ s ∈ S ] ⟦ f s ⟧C-sop X
```

In this universe, we can define the type of lists as a description quantified over a type:

```
  ListD′ : Type → U-sop
  ListD′ A = 𝟙
           ∷ (σ A λ _ → ρ 𝟙)
           ∷ []
```

Using this universe requires us to split functions on descriptions into multiple parts, but makes interconversion between representations and concrete types straightforward.

6.4 Parametrized types

The encoding of fields in `U-sop` makes the descriptions large. In contrast to the universes before, this makes `U-sop` not foldable. By introducing parameters in the right way[9], we can restore the foldability of the universe as a bonus.

In the last section, we saw that we could define the parametrized type `List` by quantifying over a type. However, in some cases, we will want to be able to inspect or modify the parameters belonging to a type[11]. To represent the parameters of a type, we will need a new gadget. The parameters could, of course, be represented like `List Type`, but this excludes types like the existential quantifier `Σ_`. In a general parametrized type, parameters can refer to bound values of preceding parameters. The parameters of a type are thus a sequence of types depending on each other, which we call telescopes. We define telescopes using induction-recursion:

```
  data Tel′ : Type
  ⟦_⟧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. Here, we used the meaning of a telescope `⟦_⟧tel` to define types in the context of a telescope. This meaning represents the valid assignment of values to parameters:

```
  ⟦ ∅     ⟧tel′ = ⊤
  ⟦ Γ ▷ S ⟧tel′ = Σ ⟦ Γ ⟧tel′ S
```

interpreting a telescope into the dependent product of all the parameter types.

This definition of telescopes would let us write down the type of `Σ`:

[9]The right way for us, that is. If a foldable universe means nothing to you, there are simpler encodings for parameters and indices, which are recorded in the appendix[10].

[11]For example, deriving Traversable for parametrized types as functions would not be possible (without macros), as one could not decide whether the signature of a type in a field is compatible.

```
Σ-Tel : Tel′
Σ-Tel = ∅ ▷ const Type ▷ (λ A → A → Type) ∘ snd
```
but is not sufficient to give its definition, as we need to be able to bind a value `a` of `A` and reference it in the field `P a`. By quantifying telescopes over a type, we can represent bound arguments using almost the same setup [EC22]:
```
data Tel (P : Type) : Type
⟦_⟧tel : Tel P → P → Type
```
A `Tel P` then represents a telescope for each value of `P`, which we can view as a telescope in the context of `P`. For readability, we redefine values in the context of a telescope as:
```
_⊢_ : Tel P → Type → Type
Γ ⊢ A = Σ _ ⟦ Γ ⟧tel → A
```
so we can define telescopes and their interpretations as:
```
data Tel P where
  ∅   : Tel P
  _▷_ : (Γ : Tel P) (S : Γ ⊢ Type) → Tel P

⟦ ∅     ⟧tel p = ⊤
⟦ Γ ▷ S ⟧tel p = Σ[ x ∈ ⟦ Γ ⟧tel p ] S (p , x)
```
Of course, by setting `P` = `⊤`, we recover the previous definition of parameter-telescopes. We can then define an extension of a telescope as a telescope in the context of a parameter telescope:
```
ExTel : Tel ⊤ → Type
ExTel Γ = Tel (⟦ Γ ⟧tel tt)
```
An `ExTel Γ` represents a telescope of variables over the fixed parameter-telescope `Γ`, and can be extended independently of the parameters. Extensions can be interpreted as follows: where we interpret the parameter part first, and then interpret the variables for the given interpretation of the parameters. We modify the codes of `U-sop` to make use of the new telescopes
```
data Con-par (Γ : Tel ⊤) (V : ExTel Γ) : Type
data U-par (Γ : Tel ⊤) : Type where
  []  : U-par Γ
  _∷_ : Con-par Γ ∅ → U-par Γ → U-par Γ

data Con-par Γ V where
  𝟙 : Con-par Γ V
  ρ : Con-par Γ V → Con-par Γ V
  σ : (S : V ⊢ Type) → Con-par Γ (V ▷ S) → Con-par Γ V
```
The function `S → U-sop` is now replaced by `U-par (V ▷ S)`, preserving the dependence in the variable-telescope, while avoiding the higher order argument. The interpretation is then given as:
```
⟦_⟧U-par : U-par Γ → (⟦ Γ ⟧tel tt → Type) → ⟦ Γ ⟧tel tt → Type
⟦_⟧C-par : Con-par Γ V → (⟦ Γ & V ⟧tel → Type) → ⟦ Γ & V ⟧tel → Type

⟦ []    ⟧U-par X p = ⊥
⟦ C ∷ D ⟧U-par X p = ⟦ C ⟧C-par (X ∘ fst) (p , tt) × ⟦ D ⟧U-par X p
⟦ 𝟙     ⟧C-par X p = ⊤
⟦ ρ C   ⟧C-par X p = X p × ⟦ C ⟧C-par X p
⟦ σ S C ⟧C-par X pv@(p , v)
  = Σ[ s ∈ S pv ] ⟦ C ⟧C-par (λ { (p , v , _) → X (p , v) }) (p , v , s)
```
We can describe lists using a one-type telescope:
```
ListD : U-par (∅ ▷ const Type)
ListD = 𝟙
      ∷ σ (snd ∘ fst) (ρ 𝟙)
      ∷ []
```
Using the variable bound in σ, we can also define the existential quantifier:
```
SigmaD : U-par (∅ ▷ const Type ▷ (λ A → A → Type) ∘ snd ∘ snd)
SigmaD = σ (snd ∘ fst ∘ fst)
       ( σ (λ { ((p , B) , _ , a) → B a }) 𝟙)
       ∷ []
```

6.5 Indexed types

Lastly, we can integrate indexed types into the universe by abstracting over indices. Recall that in native Agda datatypes, a choice of constructor can fix the indices of the recursive fields and the resultant type, so we encode[12]:
```
data Con-ix (Γ : Tel ⊤) (V : ExTel Γ) (I : Type) : Type
data U-ix (Γ : Tel ⊤) (I : Type) : Type where
  []  : U-ix Γ I
  _∷_ : Con-ix Γ ∅ I → U-ix Γ I → U-ix Γ I
```
The in most cases, the index is simply threaded through the interpretation, allowing for a choice in the relevant codes. If we are constructing a term of some indexed type, then the previous choices of constructors and arguments build up the actual index of this term. This actual index must then match the index we expected in the declaration of this term. This means that in the case of a leaf, we have to replace the unit type with the necessary equality between the expected and actual indices:
```
⟦_⟧C : Con-ix Γ V I → (⟦ Γ ⟧tel tt → I → Type) → (⟦ Γ & V ⟧tel → I → Type)
⟦ 𝟙 j   ⟧C X pv i = i ≡ (j pv)
⟦ ρ j C ⟧C X pv@(p , v) i = X p (j pv) × ⟦ C ⟧C X pv i
⟦ σ S C ⟧C X pv@(p , v) i = Σ[ s ∈ S pv ] ⟦ C ⟧C X (p , v , s) i

⟦_⟧D : U-ix 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 (p , tt) i ⊎ ⟦ Cs ⟧D X p i
```
In this universe, we can define finite types and vectors as:
```
FinD : U-ix ∅ ℕ
FinD = σ (const ℕ) (𝟙 (suc ∘ snd ∘ snd))
     ∷ σ (const ℕ) (ρ (snd ∘ snd) (𝟙 (suc ∘ snd ∘ snd)))
     ∷ []
```
and

[12]We could also use a telescope for indices, but we do not.

```
VecD : U-ix (∅ ▷ const Type) ℕ
VecD = 𝟙 (const zero)
     ∷ σ (const ℕ)
     ( σ (snd ∘ fst)
     ( ρ (snd ∘ fst ∘ snd)
     ( 𝟙 (suc ∘ snd ∘ fst ∘ snd))))
     ∷ []
```

We can now compare the structures in the quadruple (`ℕ`, `List`, `Fin`, `Vec`) by looking at their descriptions.

As a bonus, we can also use `U-ix` for generic programming. For example, by a long construction which can be found in the appendix[13], we can define the generic fold operation:

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

fold : ∀ {D : U-ix Γ I} {X}
     → ⟦ D ⟧D X ⇉ X → μ-ix D ⇉ X
```

maybe explain why this does not give you foldr or sum for lists

7 Ornaments

In this section we will introduce the concept of an ornament, used to compare descriptions, and give a simplified definition. Since we settled on `U-ix` as our universe (for now), we redefine and rename a couple of things for readability and reusability:

```
Desc = U-ix
Con  = Con-ix
μ    = μ-ix

! : A → ⊤
! x = tt

ℕD : Desc ∅ ⊤
ℕD = 𝟙 !
   ∷ ρ ! (𝟙 !)
   ∷ []

ListD : Desc (∅ ▷ const Type) ⊤
ListD = 𝟙 !
      ∷ σ (snd ∘ fst) (ρ ! (𝟙 !))
      ∷ []
```

Purely looking at their descriptions, `ℕ` and `List` are rather similar, except that `List` has a parameter and an extra field `ℕ` does not have. We could say that we can form the type of lists by starting from `ℕ` and adding this parameter and field, while keeping everything else the same. In the other direction, we see that

[13] the appendix

each list corresponds to a natural by stripping this information. Likewise, the type of vectors is almost identical to `List`, can be formed from it by adding indices, and each vector corresponds to a list by dropping the indices.

These and similar observations can be generalized using ornaments [McB14], which define a language or binary relation, describing which datatypes can be formed by decorating others. Conceptually, an ornament from a type `A` to a type `B` represents that `B` can be formed from `A` by adding information or making the indices more specific. Consequently, for each ornament from `A` to `B`, we expect to get a function from `B` to `A` erasing this information and reverting to less specific indices.

If the indices `J` (analogously, parameters `Δ`) of `B` are more specific than the indices `I` (and parameters `Γ`) of `A`, we require functions from `J` to `I` and from `Δ` to `Γ`. Our ornaments

```
Cxf : (Δ Γ : Tel P) → Type
Cxf Δ Γ = ∀ {p} → ⟦ Δ ⟧tel p → ⟦ Γ ⟧tel p

data Orn (g : Cxf Δ Γ) (i : J → I) :
        Desc Γ I → Desc Δ J → Type
```

should then come equipped with a function:

```
ornForget : ∀ {g i} → Orn g i D E
          → ∀ p j → μ E p j → μ D (g p) (i j)
```

where we define `Cxf` as the type of functions between (the interpretations of) `Δ` and `Γ`.

Since we are working with sums of products descriptions, we can decide that ornaments cannot change the number or order of constructors, and the actual work happens in the constructor ornaments:

```
Cxf′ : Cxf Δ Γ → (W : ExTel Δ) (V : ExTel Γ) → Type
Cxf′ g W V = ∀ {d} → ⟦ W ⟧tel d → ⟦ V ⟧tel (g d)

data ConOrn (g : Cxf Δ Γ) (v : Cxf′ g W V) (i : J → I) :
              Con Γ V I → Con Δ W J → Type
```

and we define ornaments as lists of ornaments for all constructors:

```
data Orn g i where
  []  : Orn g i [] []
  _∷_ : ConOrn g id i CD CE → Orn g i D E
      → Orn g i (CD ∷ D) (CE ∷ E)
```

(Similarly, we use `Cxf′` as the type of functions between variables, respecting `g`.)

To (readably) write down `ConOrn`, we use a couple of helpers to manipulate telescopes:

```
over : {g : Cxf Δ Γ} → Cxf′ g W V → ⟦ Δ & W ⟧tel → ⟦ Γ & V ⟧tel
over v (d , w) = _ , v w

Cxf′-▷ : {g : Cxf Δ Γ} (v : Cxf′ g W V) (S : V ⊢ Type)
       → Cxf′ g (W ▷ (S ∘ over v)) (V ▷ S)
Cxf′-▷ v S (p , w) = v p , w
```

```
_⊨_ : (V : Tel P) → V ⊢ Type → Type
V ⊨ S = ∀ p → S p

⊨-▷ : ∀ {S} → V ⊨ S → ∀ {p} → ⟦ V ⟧tel p → ⟦ V ▷ S ⟧tel p
⊨-▷ s v = v , s (_ , v)
```

These mostly interconvert some values between similar telescopes. But notably, if `S` is of type `V ⊢ Type`, then `S` is a type in the context of `V`, while `V ⊨ S` is the type of values of `S` in the context of `V`.

Now we can define `ConOrn`. Of course, we expect that adding nothing gives the identity ornament, which is encoded in the first three constructors of `ConOrn`.

```
data ConOrn {W = W} {V = V} g v i where
  𝟙 : ∀ {i′ j′} → (∀ w → i (j′ w) ≡ i′ (over v w))
    → ConOrn g v i (𝟙 i′) (𝟙 j′)

  ρ : ∀ {i′ j′ CD CE} → ConOrn g v i CD CE
    → (∀ w → i (j′ w) ≡ i′ (over v w))
    → ConOrn g v i (ρ i′ CD) (ρ j′ CE)

  σ : ∀ {S} {CD CE} → ConOrn g (Cxf′-▷ v S) i CD CE
    → ConOrn g v i (σ S CD) (σ (S ∘ over v) CE)

  Δσ : ∀ {S} {CD CE} → ConOrn g (v ∘ fst) i CD CE
     → ConOrn g v i CD (σ S CE)

  ∇σ : ∀ {S} {CD CE}
     → (s : V ⊨ S)
     → ConOrn g (⊨-▷ s ∘ v) i CD CE
     → ConOrn g v i (σ S CD) CE
```

However, since the parameters, indices, and variables need not be identical on both sides (in particular, the variables can diverge even more depending on the preceding ornament), we have to ask that for `𝟙` and `ρ`, these are related by a structure respecting conversion, or more graphically, a commuting square[14]. In fact, we will soon see that these pieces of information are exactly what we need to complete `ornForget`.

The other two constructors, `Δ` and `∇`, state that we can add fields, and remove fields if we provide a default value, respectively. Again, the constructor `Δ` which adds a field depending on the variables requires some manner of commuting square.

We can now formulate the formation of `List` from `ℕ` as an ornament:

```
ℕD-ListD : Orn ! id ℕD ListD
ℕD-ListD = (𝟙 (const refl))
         ∷ (Δσ (ρ (𝟙 (const refl)) (const refl)))
         ∷ []
```

Using that `ℕ` has no parameters or indices, we see that `List` has more specific parameters, namely a single type parameter, and also no indices. Because of

[14]While for `σ`, we bake the relatedness of the fields in by letting the resulting descriptions only differ by the conversion `v`.

this, all commuting squares factor through the unit type and are hence (fortunately) trivial. This ornament preserves most structure of `ℕ`, only adding a field of the type parameter of `List` using `Δ`.

We can also ornament `List` to become `Vec`, for which the index is more informative, but the ornament does equally little:

```
ListD-VecD : Orn id ! ListD VecD
ListD-VecD = (𝟙 (const refl))
           ∷ (Δσ (σ (ρ (𝟙 (const refl)) (const refl))))
           ∷ []
```

Now the commuting square for the indices is equally trivial, but while the square for the parameters is still trivial, it is now an identity square, rather than a constant one.

We deferred the definition of `ornForget`, so let us give it now. The process is split into two steps: first, we define a function to strip off a single layer of ornamentation:

```
ornErase : ∀ {X} {g i} → Orn g i D E
         → ∀ p j → ⟦ E ⟧D (λ p j → X (g p) (i j)) p j
                 → ⟦ D ⟧D X (g p) (i j)

conOrnErase : ∀ {g i} {W V} {X} {v : Cxf′ g W V}
                {CD : Con Γ V I} {CE : Con Δ W J}
            → ConOrn g v i CD CE
            → ∀ p j → ⟦ CE ⟧C (λ p j → X (g p) (i j)) p j
                    → ⟦ CD ⟧C X (over v p) (i j)
```

which uses the commutativity squares we required earlier to revert some values (and parameters, indices, and variables) to the unornamented type. For example, in the case of the `𝟙` preserving ornament[15]:

```
ornErase (CD ∷ D) p j (inj₁ x) = inj₁ (conOrnErase CD (p , tt) j x)
ornErase (CD ∷ D) p j (inj₂ x) = inj₂ (ornErase D p j x)

conOrnErase {i = i} (𝟙 sq) p j x = trans (cong i x) (sq p)
```

This function defines an algebra for the functor associated to a description `E`:

```
ornAlg : ∀ {D : Desc Γ I} {E : Desc Δ J} {g} {i}
       → Orn g i D E
       → ⟦ E ⟧D (λ p j → μ D (g p) (i j)) ⇉ λ p j → μ D (g p) (i j)
ornAlg O p j x = con (ornErase O p j x)
```

We can now make good use of the generic fold we defined in Subsection 6.5!

```
ornForget O = fold (ornAlg O)
```

The function `ornForget` also makes it easy to generalize relations of functions between similar types. For example, if we instantiate `ornForget` for `ℕD-ListD`, then the statement that list concatenation preserves length can equivalently be expressed as the commutation of concatenation and `ornForget`.

[15]The rest, appendix.

Part I
Descriptions and ornaments

•Somewhat final version above, draft/notes/rough comments/outline below. •Redo (or check) the Agda snippets below here.

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 **??** 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.

8 The descriptions

```
:warning: let's blast out the descriptions

:warning: recap: what do we want? (leaf, field, recursive
    , composite)

:warning: how do we cram number systems in? bundles

:warning: go (list everything)
```

We use telescopes identical to those in **??**:

```
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 14, 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 8.1. 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 14). 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 8.1. 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[16]
```
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.

9 The ornaments

we could ditch removal of fields: we don't use it.

[16]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.

downside: ornament over ornament is the same as field removal for deltas

:warning: match everything, add/remove field, add/remove recursive field, add/remove description field, ornament over ornament

Put something that isn't yet in ?? here.

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

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

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)) _}
  → (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[17].

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[18] 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 _ _}
```

[17]Should this become a problem like with ρ, modifying the rule to require a triangle is trivial.
[18]Kind of breaking the "ornaments relate types with similar recursive structure" interpretation.

```
    → ConOrn f' e D E
    → (∀ {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:

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.

```
  •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)
```

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 _
```
NatD was removed here

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!

10 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[19]. We will show the

[19]One might expect to need less equalities, alas, this is difficult because of Remark 18.4.

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)
```

Part II

Numerical representations

```
outline:
instead of positing a datastructure and then proving it
   correct, constructing datastructures methodically can
   give some insights into their correctness
we can run this manually, but we can also automate this
   by giving it as an ornamental description}

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 \AgdaFunction{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 \cite{ purelyfunctional}, and has some formalized examples due to Hinze and Swierstra \cite{calcdata} and Ko and Gibbons \cite{progorn}. Numerical representations are our starting point for defining more complex datastructures based on simpler ones, so we demonstrate such a calculation.

3 Calculating datastructures using Ornaments

In this part we return to the matter numerical representations. With 2.3 in mind, we can rephrase part our original question to ask

> Can numerical representations be described as ornaments on their number systems?

Let us look at a numerical representation presented as ornament in action.

[actually much like calculating datastructures in concept, but very different in practice]: #

11 From numbers to containers

Let us put the calculation from 2.2 in the framework of ornaments. Recall the Peano naturals \bN and their description

:memo: NatD

which we will use as reference number system. The associated container then has constructors

:memo: ListD

:warning: calculate

But before we can answer the question in full generality, we need ask ourselves what kind of ornaments and number systems this question applies to. Construction similar to the one in this section (e.g., the construction from binary numbers \to binary trees \to

```
    indexed binary trees) will generally fit in any setup
    of descriptions and ornaments supporting (recursive)
    fields and indices. However, there are also other less
     regular datatypes which we can still recognize as
    numerical in nature.
```

12 Finger trees

```
:warning: finger trees are pretty cool for a lot of
    reasons, but for us it is sufficient to remark that
    they are also the simplest way to get fast cons on
    both sides

:warning: what are they though

:warning: they do not fit at all

:warning: so we need more stuff

:warning: nested datatypes

:warning: composites

:warning: number systems
```

13 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 → ẏ []
  ; true → ẏ [ tt ] }
```
Here, σ adds a field to the description, upon which the rest of the description can vary, and ẏ 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 → ẏ _
  ; true → Δ A (λ _ → ẏ (ok _ , _)) }
```
Here, the σ and ẏ 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           → ẏ []
  ; (suc zero)  → ẏ [ tt ]
  ; (suc (suc zero)) → ẏ [ 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           → ẏ _
  ; (suc zero)  → Δ (power n Two A) λ _ → ẏ (ok (suc n) , _)
  ; (suc (suc zero)) → Δ (power (suc n) Two A) λ _ → ẏ (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.

14 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[20]. 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).

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

Recall the calculation of vectors from ℕ in Section 11. In this universe, we can encode ℕ and its interpretation as

```
NatND : DescI Number ∅ τ
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 14.1. *We claim then that the description given by*

```
TrieO : (D : DescI Number ∅ τ) → OrnDesc Plain (∅ ▷ const Type) ! τ ! 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 ∅ τ V) → InfoF If Number
          → ConOrnDesc Plain {W = W} {K = τ} f ! C
```

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[21]

```
  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 14.1.

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.

The reader of [McB14] might question why we are doing the work of `TrieOIx` ourselves, rather than making use of `ornAlg`.

Remark 14.1. Problems

1. an algebra for D gives no algebras for the deltas in D,
2. no delta means constructors explode.

Solution 1: define other infrastructure for algebras that do have algebras for deltas

Solution 2:

1. kill deltas by an intermediate form which explodes,
2. give up on constructor lists and have big sigma.

Solution 3:

[21]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.

1. do deltas via mutual recursion,
2. mutual recursion already has good algebras.

Remark 14.2. Also, variables are annoying

- centralize variable transforms into one place in deltas
- what are ornaments now
- package descriptions and their thinnings
- something happens

Remark 14.3. Not all functions $\mu D \to X$ come from algebras $[D]X \to X$. Consider the function pred : $N \to N$, sending suc$n$ to n and 0 to 0.

$$\begin{tikzcd}
1+N \arrow[r, "1+\text{pred}"] \arrow[d, "\text{suc}"'] \arrow[dr, "\langle 0,\text{id}\rangle"] & 1+N \arrow[d, dashed, "?"] \\
N \arrow[r, "\text{pred}"'] & N
\end{tikzcd}
\qquad
\begin{tikzcd}
N \arrow[r, "\text{pred}"] \arrow[dr, "\text{id}"'] & N \arrow[d, dotted, "!!!"] \\
 & N
\end{tikzcd}$$

We see that pred cannot come from a map $1 + N \to N$, as that map would be a retraction, while pred is not mono.

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

Part III
Related work

15 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.

15.1 Merge me

15.1.1 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].

15.1.2 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].

15.2 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[22]. We represent indexed types by parametrizing over a type I. Since we are aiming for nested types, external polymorphism[23] 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:

...

[22]For better or worse, an ornament could refer to a different ornament for a recursive field.
[23]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[24].

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.

16 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.

[24] 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.

17 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.

Part IV
Discussion

:warning: the trie ornament is hard to prove about

:warning: nesting rather than branching

:warning: folds don't give folds over parts

:warning: composites complicate the indexed variant

:warning: the index type becomes more awkward than with big sigmas

18 Temporary: future work (Part I)

Remark 18.1. 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 18.2. Variable transforms are not essential in these descriptions, but there are a couple of reasons for keeping them. In particular, they make it 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 18.3. 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 18.4. 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).

Remark 18.5. Comparing SOP and computational sigmas. In particular, `s N ( n → v (replicate n tt))` is not in SOP without full nesting. SOP is good for generics in both directions (the conversion in both ways keeps the datatype like it is supposed to). On the other hand, computational sigmas make writing and proving about `Path` a lot easier.

19 Temporary: future work (Part II)

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?

References

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

[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`.

[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`.

[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`.

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

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

[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

:warning: The other way of parameters—indices

A Finger trees

B Heterogenization

C More equivalences for less effort

Noting that constructing equivalences directly or from isomorphisms as in **??** 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 **??** 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 **??**, 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 **??**, but we argue that most of this code can likely be automated.

Merge