Ornaments and Proof Transport applied to Numerical Representations

Samuel Klumpers
6057314

October 25, 2023

Abstract

The dependently typed functional programming language Agda encourages defining custom datatypes to write correct-by-construction programs with. In some cases, even those datatypes can be made correct-by-construction, by manually distilling them from a mixture of requirements, as opposed to pulling them out of thin air. This is in particular the case for numerical representations, a class of datastructures inspired by number systems, containing structures such as linked lists and binary trees. However, constructing datatypes in this manner, and establishing the necessary relations between them can quickly become tedious and duplicative.

long

distracted

In the general case, employing datatype-generic programming can curtail code-duplication by allowing the definition of constructions that can be instantiated to a class of types. Furthermore, ornaments make it possible to succinctly describe relations between structurally similar types.

In this thesis, we apply generic programming and ornaments to numerical representations, giving a recipe to compute such a representation from a provided number system. For this, we describe a generic universe and a type of ornaments on it, allowing us to formulate the recipe as an ornament from a number system to the computed datatype.

Todo legend:

To do

There is something missing here

General and vague self-criticism, might ignore. Might do something if I have time on my hands.

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;

'Programming is hard' - citation needed? Misschien beter om de nadruk te leggen op iets als 'statically typed programming languages can rule 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 \cite{agda}, 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 \cite{algorn, sijsling}, we can also discuss how we can transform one piece of code into another by comparing the descriptions of the two pieces.

Maar misschien is het nog beter om een iets andere opening gambit te kiezen. Er is een 'folklore' relatie tussen getalsystemen en datastructuren – maar wat bedoel je hier precies mee? Kunnen we niet ornaments (en dependent types) gebruiken om deze relatie precies te maken? En wat levert dit inzicht ons op?

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

Ik zou proberen om weg te blijven

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. 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.

In de inleiding ook goed om te noemen dat deze universe constructies de manier zijn om (datatype) generic programming in Agda te doen.

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

Background

Start A

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

We will describe some common datatypes and how they can be used for programming, exploring how dependent types also let us use datatypes to prove properties of programs, or write programs that are correct-by-construction, leading us to discuss descriptions of datatypes and ornaments.

2 Agda

We formalize our work in the programming language Agda [Tea23]. While we will only occasionally reference Haskell, those more familiar with Haskell might understand (the reasonable part of) Agda as the subset of total Haskell programs [Coc+22].

Agda is a total functional programming language with dependent types. Here, 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 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 we use in the later sections. Many of the types we use in this section are also described and explained in most Agda tutorials ([Nor09], [WKS22], etc.), and can be imported from the standard library [The23].

Note that we use `--type-in-type` to keep the explanations more readable.

3 Data in Agda

At the level of 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, we can define the type of *booleans*:

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

The constructors of this type state that we can make values of `Bool` in exactly two ways: `false` and `true`. We can then define 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*, the coverage checker ensures we define the function on all cases of the type matched on, and thus the function is completely defined.

We can also define a type representing the natural numbers

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

Here, `ℕ` always has a `zero` element, and for each element n the constructor `suc` expresses that there is also an element representing $n + 1$. Hence, `ℕ` represents the *naturals* by encoding the existential axioms of the Peano axioms. By pattern matching and recursion on `ℕ`, we define the less-than operator:

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

One of the cases contains a recursive instance of `ℕ`, so termination checker also verifies that this recursion indeed terminates, ensuring that we still define `n <? m` for all possible combinations of `n` and `m`. In this case the recursion is valid, since both arguments decrease before the recursive call, meaning that at some point `n` or `m` hits `zero` and the recursion terminates.

Like in Haskell, we can *parametrize* a datatype over other types to make *polymorphic* type, which we can use to define lists of values for all types:

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

A list of `A` can either be empty `[]`, or contain an element of `A` and another list via `_∷_`. In other words, `List` is a type of *finite sequences* in `A` (in the sense of sequences as an abstract type [Oka98]).

Using polymorphic functions, we can manipulate and inspect lists by inserting or extracting elements. For example, we can define a function to look up the value at some position `n` in a list

```
lookup? : List A → ℕ → Maybe A
```

```
lookup? []       n       = nothing
lookup? (x ∷ xs) zero    = just x
lookup? (x ∷ xs) (suc n) = lookup? xs n
```

However, this function *partial*, 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 position falls outside the list and we cannot return an element. If we know the length of the list `xs`, then we also know for which positions `lookup` will succeed, and for which it will not. We define

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

so that we can test whether the position `n` lies inside the list by checking `n <? length xs`. If we declare `lookup` as a dependent function consuming a proof of `n <? length xs`, then `lookup` always succeeds. However, this actually only moves the burden of checking whether the output was `nothing` afterwards to proving that `n <? length xs` beforehand.

We can avoid both by defining an *indexed type* representing numbers below an upper bound

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

Like parameters, indices add a variable to the context of a datatype, but unlike parameters, indices can influence the availability of constructors. The type `Fin` is defined such that a variable of type `Fin n` represents a number less than `n`. Since both constructors `zero` and `suc` dictate that the index is the `suc` of some natural `n`, we see that `Fin zero` has no values. On the other hand, `suc` gives a value of `Fin (suc n)` for each value of `Fin n`, and `zero` gives exactly one additional value of `Fin (suc n)` for each `n`. By induction (externally), we find that `Fin n` has exactly `n` closed terms, each representing a number less than `n`.

To complement `Fin`, we define another indexed type representing lists of a known length, also known as vectors:

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

The `[]` constructor of this type produces the only term of type `Vec A zero`. The `_∷_` constructor ensures that a `Vec A (suc n)` always consists of an element of `A` and a `Vec A n`. By induction, we find that a `Vec A n` contains exactly `n` elements of `A`. Thus, we conclude that `Fin n` is exactly the type of positions in a `Vec A n`. In comparison to `List`, we can say that `Vec` is a type of arrays (in the sense of arrays as the abstract type of sequences of a fixed length). Furthermore, knowing the index of a term `xs` of type `Vec A n` uniquely determines the the constructor it was formed by. Namely, if `n` is `zero`, then `xs` is `[]`, and if `n` is `suc` of `m`, then `xs` is formed by `_∷_`.

Using this, we define a variant of `lookup` for `Fin` and `Vec`, taking a vector of length `n` and a position below `n`:

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

The case in which we would return `nothing` for lists, which is when `xs` is `[]`, is omitted. This happens because `x` of type `Fin n` is either `zero` or `suc i`, and both cases imply that `n` is `suc m` for some `m`. As we saw above, a `Vec A (suc m)` is always formed by `_∷_`, making the case in which `xs` is `[]` impossible. Consequently, lookup always succeeds for vectors, however, this does not yet prove that `lookup` necessarily returns the right element, we will need some more logic to verify this.

4 Proving in Agda

To describe equality of terms we define a new type

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

If we have a value `x` of `a ≡ b`, then, as the only constructor of `_≡_` is `refl`, we must have that `a` is equal to `b`. We can use this type to describe the behaviour of functions like `lookup`: If we insert elements into a vector with

```
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
```

stating that we expect to find an element where we insert it.

To prove the statement, we proceed as when defining any other function. By simultaneous induction on the position and vector, we prove

```
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
```

In the first two cases, where we `lookup` the first position, `insert xs zero y` simplifies to `y ∷ xs`, so the lookup immediately returns `y` as wanted. In the last case, we have to prove that `lookup` is correct for `x ∷ xs`, so we use that the `lookup` ignores the term `x` and we appeal to the correctness of `lookup` on the smaller list `xs` to complete the proof.

Like `_≡_`, we can encode many other logical operations into datatypes, which establishes a correspondence between types and formulas, known as the Curry-Howard isomorphism. For example, we can encode disjunctions (the logical 'or' operation) as

```
data _⊎_ A B : Type where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

The other components of the isomorphism are as follows. Conjunction (logical 'and') can be represented by[1]

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

True and false are respectively represented by

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

so that always `tt : ⊤`, and

```
data ⊥ : Type where
```

The body of `⊥` is not accidentally left out: because `⊥` has no constructors, there is no proof of false[2].

Because we identify function types with logical implications, we can also define the negation of a formula `A` as "`A` implies false":

```
¬_ : Type → Type
¬ A = A → ⊥
```

The logical quantifiers $\forall$ and $\exists$ act on formulas with a free variable in a specific domain of discourse. We represent closed formulas by types, so we can represent a formula with a free variable of type `A` by a function values of `A` to types `A → Type`, also known as a predicate. The universal quantifier $\forall a P(a)$ is true when for all a the formula $P(a)$ is true, so we represent the universal quantification of a predicate `P` as a dependent function type `(a : A) → P a`, producing for each `a` of type `A` a proof of `P a`. The existential quantifier $\exists a P(a)$ is true when there is some a such that $P(a)$ is true, so we represent the existential quantification as

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

so that we have `Σ A P` iff we have an element `fst` of `A` and a proof `snd` of `P a`. To avoid the need for lambda abstractions in existentials, we define the syntax

```
syntax Σ-syntax A (λ x → P) = Σ[ x ∈ A ] P
```

letting us write `Σ[ a ∈ A ] P a` for $\exists a P(a)$.

5 Descriptions

In the previous sections we completed a quadruple of types (`ℕ`, `List`, `Vec`, `Fin`), which have nice interactions (`length`, `lookup`). Similar to the type of `length : List A → ℕ`, we can define

[1] We use a record here, rather than a datatype with a constructor `A → B → A × B`. The advantage of using a record is that this directly gives us projections like `fst : A × B → A`, and lets us use eta equality, making $(a, b) = (c, d) \iff a = c \wedge b = d$ holds automatically.

[2] If we did not use `--type-in-type`, and even in that case I can only hope.

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

converting vectors back to lists. In the other direction, we can also promote a list to a vector by recomputing its index:

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

We claim that is not a coincidence, but rather happens because `ℕ`, `List`, and `Vec` have the same "shape".

But what is the shape of a datatype? In this section, we will explain a framework of datatype descriptions and ornaments, allowing us to describe the shapes of datatypes and use these for generic programming [Nor09; AMM07; eff20; EC22]. Recall that while polymorphism allows us to write one program for many types at once, those programs act parametrically [Rey83; Wad89]: polymorphic functions must work for all types, thus they cannot inspect values of their type argument. Generic programs, by design, do use the structure of a datatype, allowing for more complex functions that do inspect values[3].

Using datatype descriptions we can then relate `ℕ`, `List` and `Vec`, explaining how `length` and `toList` are instances of a generic construction. Let us walk through some ways of defining descriptions. We will start from simpler descriptions, building our way up to more general types, until we reach a framework in which we can describe `ℕ`, `List`, `Vec` and `Fin`.

5.1 Finite types

A datatype description, which are datatypes of which each value again represents a datatype, consist of two components. Namely, a type of descriptions `U`, also referred to as codes, and an interpretation `U → Type`, decoding descriptions to the represented types. In the terminology of Martin-Löf type theory (MLTT)[Cha+10], where types of types like `Type` are called universes, we can think of a type of descriptions as an internal universe.

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
```

The meaning of the codes in this universe is then assigned by the interpretation

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

[3] Think of JSON encoding types with encodable fields [VL14], or deriving functor instances for a broad class of types [Mag+10].

which indeed sends `𝟘` to `⊥`, `𝟙` to `⊤`, sums to sums and products to products[4].

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`.

5.2 Recursive types

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`. As a consequence, we have to split the interpretation 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)
```

Then, 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:

```
NatD : U-rec
NatD = 𝟙 ⊕ ρ
```

5.3 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
```

[4]One might recognize that `⟦_⟧fin` is a morphism between the rings (`U-fin`, `⊕`, `⊗`) and (`Type`, `⊎`, `×`). Similarly, `Fin` also gives a ring morphism from `ℕ` with `+` and `×` to `Type`, and in fact `⟦_⟧fin` factors through `Fin` via the map sending the expressions in `U-fin` to their value in `ℕ`.

```
    [] : 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.

5.4 Parametrized types

The encoding of fields in `U-sop` makes the descriptions large in the following sense: by letting `S` in `σ` be an infinite type, we can get a description referencing infinitely many other descriptions. As a consequence, we cannot inspect an arbitrary description in its entirety. We will introduce parameters in such a way that we recover the finiteness of descriptions 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. To represent the parameters of a type, we will need a new gadget.

In a naive attempt, we can represent the parameters of a type as `List Type`. However, this cannot represent many useful types, of which the parameters depend on each other. For example, in the existential quantifier `Σ_`, the type `A → Type` of second parameter `B` references back to the first parameter `A`.

In a general parametrized type, parameters can refer to the values of all preceding parameters. The parameters of a type are thus a sequence of types depending on each other, which we call telescopes [EC22; Sij16; Bru91] (also known as contexts in MLTT). 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`:

```
Σ-Tel : Tel′
Σ-Tel = ∅ ▷ const Type ▷ (λ A → A → Type) ∘ snd
```

but is not sufficient to define `Σ`, 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 [EC22], we can represent bound arguments using almost the same setup:

```
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)
```

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)
```

representing a telescope of variables over the fixed parameter-telescope `Γ`, which can be extended independently of `Γ`. Extensions can be interpreted by interpreting the variable part given the interpretation of the parameter part:

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

In the descriptions directly relay the parameter telescope to the constructors, resetting the variable telescope to `∅` for each constructor:

```
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
```

Of the constructors we only modify the `σ` to request a type `S` in the context of `V`, and to extend the context for the subsequent fields by `S`: Replacing the function `S → U-sop` by `Con-par (V ▷ S)` allows us to bind the value of `S` while avoiding the higher order argument. We define a helper

Split or explain all the crazy helpers

```
map-var : ∀ {A B C} → (∀ {a} → B a → C a) → Σ A B → Σ A C
map-var f (a , b) = (a , f b)

-- hi please explain me
Cxf : (Δ Γ : Tel P) → Type
Cxf Δ Γ = ∀ {p} → ⟦ Δ ⟧tel p → ⟦ Γ ⟧tel p

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

var→par : {g : Cxf Δ Γ} → Vxf g W V → ⟦ Δ & W ⟧tel → ⟦ Γ & V ⟧tel
var→par v (d , w) = _ , v w

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

and interpret this universe as follows:

```
⟦_⟧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 pv = ⊤
⟦ ρ C   ⟧C-par X pv = X pv × ⟦ C ⟧C-par X pv
⟦ σ S C ⟧C-par X pv@(p , v)
  = Σ[ s ∈ S pv ] ⟦ C ⟧C-par (X ∘ map-var fst) (p , v , s)
```

In particular, we provide `X` the parameters and variables in the `σ` case, and extend context by `s` before passing to the rest of the interpretation.

In this universe, we can describe lists using a one-type telescope:

```
ListD : U-par (∅ ▷ const Type)
ListD = 𝟙
      ∷ σ (λ ((_ , A) , _) → A) (ρ 𝟙)
      ∷ []
```

This description declares that `List` has two constructors, one with no fields, corresponding to `[]`, and the second with one field and a recursive field, representing `_∷_`. In the second constructor, we used pattern lambdas to deconstruct the telescope[5] and extract the type `A`. Using the variable bound in `σ`, we can also define the existential quantifier:

```
SigmaD : U-par (∅ ▷ const Type ▷ λ { (_ , _ , A) → A → Type })
SigmaD = σ (λ (((_ , A) , _) , _) → A   )
       ( σ (λ ((_ , B) , (_ , a)) → B a )
         𝟙)
       ∷ []
```

having one constructor with two fields. Here, the first field of type `A` adds a value `a` to the variable telescope, which we recover in the second field by pattern matching, before passing it to `B`.

5.5 Indexed types

Lastly, we can integrate indexed types [DS06] into the universe by abstracting over indices

Unshake mutuals

```
data Con-ix (Γ : Tel τ) (V : ExTel Γ) (I : Type) : Type
data U-ix (Γ : Tel τ) (I : Type) : Type where
  []  : U-ix Γ I
  _∷_ : Con-ix Γ ∅ I → U-ix Γ I → U-ix Γ I
```

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:

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

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 [McB14]:

```
⟦_⟧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 a recursive field, the expected index can be chosen based on parameters and variables.

[5] Due to a quirk in the interpretation of telescopes, the `∅` part always contributes a value `tt` we explicitly ignore, which also explicitly needs to be provided when passing parameters and variables.

In this universe, we can define finite types and vectors as:

```
FinD : U-ix ∅ ℕ
FinD = σ (const ℕ)
       ( 𝟙 (λ (_ , (_ , n)) → suc n))
       ∷ σ (const ℕ)
       ( ρ (λ (_ , (_ , n)) → n)
       ( 𝟙 (λ (_ , (_ , n)) → suc n)))
       ∷ []
```

and

```
VecD : U-ix (∅ ▷ const Type) ℕ
VecD = 𝟙 (const zero)
       ∷ σ (const ℕ)
       ( σ (λ { ((_ , A) , _) → A } )
       ( ρ (λ { (_ , ((_ , n) , _)) → n } )
       ( 𝟙 (λ { (_ , ((_ , n) , _)) → suc n } ))))
       ∷ []
```

These are equivalent, but since we do not model implicit fields, they are slightly different in use compared to `Fin` and `Vec`. In the first constructor of `VecD` we report an actual index of `zero`. In the second, we have a field `ℕ` to bring the index `n` into scope, which is used to request a recursive field with index `n`, and report the actual index of `suc n`.

Let us also show how the definitions of naturals and lists from earlier sections can be replicated in `U-ix`

```
! : A → ⊤
! x = tt

NatD : U-ix ∅ ⊤
NatD = 𝟙 !
       ∷ ρ !
       ( 𝟙 !)
       ∷ []

ListD : U-ix (∅ ▷ const Type) ⊤
ListD = 𝟙 !
        ∷ σ (λ { ((_ , A) , _) → A })
        ( ρ !
        ( 𝟙 ! ))
        ∷ []
```

Writing the descriptions `NatD`, `ListD` and `VecD` next to each other makes it easy to see the similarities: `ListD` is the same as `NatD` with a type parameter and one more `σ`. Likewise, `VecD` is the same as `ListD`, but now indexing over `ℕ` and with yet one more `σ` of `ℕ`. This kind of analysis is the focus of Section 6.

5.5.1 Generic Programming

As a bonus, we can also use `U-ix` for generic programming. For example, by a long construction which can be found in Appendix E, 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
```

Let us describe how fold works intuitively. We can interpret a term of ⟦ D ⟧D X as a term of μ-ix D, where the recursive positions hold values of X rather than values of μ-ix D. Then fold states that a function collapsing such terms into values of X extends to a function collapsing μ-ix D into X, recursively collapsing applications of con from the bottom up.

As a more concrete example, when instantiating fold to ListD, the type ⟦ ListD ⟧D X reduces (up to equivalence) to ⊤ ⊎ (A × X A) → X A, and fold becomes

```
foldr : {X : Type → Type}
      → (∀ A → ⊤ ⊎ (A × X A) → X A)
      → ∀ B → List B → X B
```

which, much like the familiar foldr operation lets us consume a List A to produce a value X A, provided a value X A in the empty case, and a means to convert a pair (A, X A) to X A.

Do note that this version takes a polymorphic function as an argument, as opposed to the usual fold which has the quantifiers on the outside:

```
foldr′ : ∀ A B → (⊤ ⊎ (A × B) → B) → List A → B
```

Like a couple of constructions we will encounter in later sections, we can recover the usual fold into a type C by generalizing C to some kind of maps into C. For example, by letting X be continuation-passing computations into ℕ, we can recover

```
sum′ : ∀ A → List A → (A → ℕ) → ℕ
sum′ = foldr {X = λ A → (A → ℕ) → ℕ} go
  where
  go : ∀ A → ⊤ ⊎ (A × ((A → ℕ) → ℕ)) → (A → ℕ) → ℕ
  go A (inj₁ tt)       f = zero
  go A (inj₂ (x , xs)) f = f x + xs f

sum : List ℕ → ℕ
sum xs = sum′ ℕ xs id
```

6 Ornaments

In this section we will introduce a simplified definition of ornaments, which we will use to compare descriptions. 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 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.

Observations like these can be generalized using ornaments [McB14; KG16; Sij16], which define a binary relation describing which datatypes can be formed by "decorating" others. Conceptually, a type can be decorated by adding or modifying fields, extending its parameters, or refining its indices.

Essential to the concept of ornaments is the ability to convert back, forgetting the extra structure. After all, if there is an ornament from A to B, then B is A with extra fields and parameters, and more specific indices. In that case, we should also be able to discard those extra fields, parameters, and more specific indices, obtaining a conversion from B to A. If A is a U-ix Γ I and B is a U-ix Δ J, then a conversion from B to A presupposes a function re-par : Cxf Δ Γ for re-parametrization, and a function re-index : J → I for re-indexing.

In the same way that descriptions in U-ix are lists of constructor descriptions, ornaments are lists of constructor ornaments. We define the type of ornaments reparametrizing with re-par and reindexing with re-index as a type indexed over U-ix:

```
data Orn (re-par : Cxf Δ Γ) (re-index : J → I) :
     U-ix Γ I → U-ix Δ J → Type where
  []  : Orn re-par re-index [] []
  _∷_ : ConOrn re-par id re-index CD CE
      → Orn re-par re-index D E
      → Orn re-par re-index (CD ∷ D) (CE ∷ E)
```

The conversion between types induced by an ornament is then embodied by the forgetful map

```
bimap : {A B C D E : Type}
      → (A → B → C) → (D → A) → (E → B)
      → D → E → C
bimap f g h d e = f (g d) (h e)
ornForget : ∀ {re-par re-index} → Orn re-par re-index D E
          → μ-ix E ⇉ bimap (μ-ix D) re-par re-index
```

which will revert the modifications made by the constructor ornaments, and restores the original indices and parameters.

The allowed modifications are controlled by the definition of constructor ornaments ConOrn. We must keep in mind that each constructor of ConOrn also has to be reverted by ornForget, accordingly, some modifications have preconditions, which are in this case always pointwise equalities: Since constructors exist in the context of variables, we let constructor ornaments transform variables with re-var, in addition to parameters and indices.

The first three constructors of ConOrn represent the operations which copy the corresponding constructors of Con-ix[6]. The Δσ constructors allows one to add fields which are not present on the original datatype.

```
data ConOrn (re-par : Cxf Δ Γ) (re-var : Vxf re-par W V) (re-index : J → I) :
```

[6] Viewing ConOrn as a binary relation on Con-ix, these represent the preservation of ConOrn by 𝟙, ρ, and σ, up to parameters, variables, and indices.

```
            Con-ix Γ V I → Con-ix Δ W J → Type where
  𝟙 : ∀ {i j}
    → re-index ∘ j ∼ i ∘ var→par re-var
    → ConOrn re-par re-var re-index (𝟙 i) (𝟙 j)

  ρ : ∀ {i j CD CE}
    → re-index ∘ j ∼ i ∘ var→par re-var
    → ConOrn re-par re-var re-index CD CE
    → ConOrn re-par re-var re-index (ρ i CD) (ρ j CE)

  σ : ∀ {S CD CE}
    → ConOrn re-par (Vxf-▷ re-var S) re-index CD CE
    → ConOrn re-par re-var re-index (σ S CD) (σ (S ∘ var→par re-var) CE)

  Δσ : ∀ {S CD CE}
     → ConOrn re-par (re-var ∘ fst) re-index CD CE
     → ConOrn re-par re-var re-index CD (σ S CE)
```

The commuting square `re-index ∘ j ∼ i ∘ var→par re-var` in the first two constructors ensures that the indices on both sides are indeed related, up to `re-index` and `re-var`.

Now, we can show that lists are indeed naturals decorated with fields:

```
NatD-ListD : Orn ! id NatD ListD
NatD-ListD = 𝟙 (const refl)
           ∷ Δσ {S = λ { ((_ , A), _) → A }}
           ( ρ (const refl)
           ( 𝟙 (const refl)))
           ∷ []
```

This ornament preserves most structure of `ℕ`, only adding a field using `Δσ`[7]. As `ℕ` has no parameters or indices, `List` has more specific parameters, namely a single type parameter. Consequently, all commuting squares factor through the unit type and can be satisfied with `λ _ → refl`.

We can also ornament lists to get vectors by reindexing them over `ℕ`

```
ListD-VecD : Orn id ! ListD VecD
ListD-VecD = 𝟙 (const refl)
           ∷ Δσ {S = λ _ → ℕ}
           ( σ
           ( ρ {j = λ { (_ , (_ , n) , _) → n }}      (const refl)
           ( 𝟙 {j = λ { (_ , (_ , n) , _) → suc n }} (const refl))))
           ∷ []
```

We bind a new field of `ℕ` with `Δσ`, extracting it in `𝟙` and `ρ` to declare that the constructor corresponding to `_∷_` takes a vector of length `n` and returns a vector of length `suc n`.

The conversions from lists to naturals, and from vectors to lists are given by `ornForget`. We define `ornForget` as a `fold` over an algebra that erases a single layer of decorations

[7] Note that `S` is an implicit argument: Agda would happily infer it from `ListD` had we omitted it

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

Recursively applying this algebra, which reinterprets values of `E` as values of `D`, lets us take apart a value in the fixpoint `μ-ix E` and rebuild it to a value of `μ-ix D`. This algebra

```
ornAlg : ∀ {D : U-ix Γ I} {E : U-ix Δ J} {re-par re-index}
       → Orn re-par re-index D E
       → ⟦ E ⟧D (bimap (μ-ix D) re-par re-index)
         ⇒ bimap (μ-ix D) re-par re-index
ornAlg O p j x = con (ornErase O p j x)
```

is a special case of the erasing function, which undecorates interpretations of arbitrary types `X`:

```
ornErase : ∀ {re-par re-index} {X}
         → Orn re-par re-index D E
         →  ⟦ E ⟧D (bimap X re-par re-index)
            ⇒ bimap (⟦ D ⟧D X) re-par re-index
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 : ∀ {re-par re-index} {W V} {X} {re-var : Vxf re-par W V}
                {CD : Con-ix Γ V I} {CE : Con-ix Δ W J}
            → ConOrn re-par re-var re-index CD CE
            → ⟦ CE ⟧C (bimap X re-par re-index)
              ⇒ bimap (⟦ CD ⟧C X) (var→par re-var) re-index
conOrnErase {re-index = i} (𝟙 sq) p j x  = trans (cong i x) (sq p)
conOrnErase {X = X} (ρ sq CD) p j (x , y) = subst (X _) (sq p) x , conOrnErase CD p j y
conOrnErase (σ CD) (p , w) j (s , x)      = s , conOrnErase CD (p , w , s) j x
conOrnErase (Δσ CD) (p , w) j (s , x)     = conOrnErase CD (p , w , s) j x
```

Reading off the ornament, we see which bits of `CE` are new and which are copied from `CD`, and consequently which parts of a term `x` under an interpretation of `CE` need to be forgotten, and which needs to be copied or translated. Specifically, the first three cases of `conOrnErase` correspond to the structure-preserving ornaments, and merely translate equivalent structures from `CE` to `CD`.

For example, in the first case the ornament `𝟙 sq` copies leaves, telling us that `CD` is `𝟙 i'` and `CE` is `𝟙 j'`. The interpretation `⟦ 𝟙 j' ⟧C _ p j` of a leaf `𝟙 j'` at parameters `p` and index `j` is simply the equality of expected and actual indices `j ≡ (j' p)`. The term `x` of `j ≡ (j' p)`, then only has to be converted to the corresponding proof of equality on the `CD` side: `re-index j ≡ (i' (var→par re-var p))`. This is precisely accomplished by applying `re-index` to both sides and composing with the square `sq` at `p`.

Likewise, in the case of `ρ` we only have to show that `x` can be converted from one `ρ` to the other `ρ` by translating its parameters, and in the `σ` case the field is directly copied. The only other ornament `Δσ` adding fields, is easily undone by removing those fields.

Thus, `ornForget` establishes that `E` in an ornament `Orn g i D E` is an adorned version of `D` by associating to each value of `E` its an underlying value in `D`. Additionally, `ornForget` makes it simple to relate functions between related types.

For example, instantiating ornForget for NatD-ListD yields length. Hence, the statement that length sends concatenation _++_ to addition _+_, i.e. length (xs ++ ys) ≡ length xs + length ys, is equivalent to the statement that _++_ and _+_ are related or that _++_ is a lifting of _+_ [DM14].

Ik hoop dat dit minder wazig is en de mental typechecking load wat reduceert.

7 Ornamental Descriptions

By defining the ornaments NatD-ListD and ListD-VecD we could show that lists are numbers with fields and vectors are lists with fixed lengths. Even though we had to give ListD before we could define NatD-ListD, the value of NatD-ListD actually forces the right-hand side to be ListD.

End A

This means we can also use an ornament to represent a description as a patch on top of another description, if we leave out the right-hand side of the ornament. Ornamental descriptions are precisely defined as ornaments without the right-hand side, and effectively bundle a description and an ornament to it. Their definition is analogous to that of ornaments, making the arguments which would only appear in the new description explicit:

```
data OrnDesc (Δ : Tel τ) (J : Type)
    (re-par : Cxf Δ Γ) (re-index : J → I)
    : U-ix Γ I → Type where
  []  : OrnDesc Δ J re-par re-index []
  _∷_ : ConOrnDesc Δ ∅ J re-par ! re-index CD
     → OrnDesc Δ J re-par re-index D
     → OrnDesc Δ J re-par re-index (CD ∷ D)
data ConOrnDesc (Δ : Tel τ) (W : ExTel Δ) (J : Type)
                (re-par : Cxf Δ Γ) (re-var : Vxf re-par W V) (re-index : J → I)
                : Con-ix Γ V I → Type where
  𝟙 : ∀ {i} (j : W ⊢ J)
    → re-index ∘ j ~ i ∘ var→par re-var
    → ConOrnDesc Δ W J re-par re-var re-index (𝟙 i)

  ρ : ∀ {i} {CD} (j : W ⊢ J)
    → re-index ∘ j ~ i ∘ var→par re-var
    → ConOrnDesc Δ W J re-par re-var re-index CD
    → ConOrnDesc Δ W J re-par re-var re-index (ρ i CD)

  σ : ∀ (S : V ⊢ Type) {CD}
    → ConOrnDesc Δ (W ▷ S ∘ var→par re-var) J re-par (Vxf-▷ re-var S) re-index CD
    → ConOrnDesc Δ W J re-par re-var re-index (σ S CD)

  Δσ : ∀ (S : W ⊢ Type) {CD}
     → ConOrnDesc Δ (W ▷ S) J re-par (re-var ∘ fst) re-index CD
     → ConOrnDesc Δ W J re-par re-var re-index CD
```

Using OrnDesc we can describe lists as the patch on NatD which inserts a σ in the constructor corresponding to suc:

Explain the lambda

```
NatOD : OrnDesc (∅ ▷ const Type) ⊤ ! ! NatD
NatOD = 𝟙 (λ _ → tt) (λ a → refl)
      ∷ Δσ (λ { ((_ , A) , _) → A })
      ( ρ (λ _ → tt) (λ a → refl)
      ( 𝟙 (λ _ → tt) (λ a → refl)) )
      ∷ []
```

To extract ListD from NatOD, we can use the projection applying the patch in an ornamental description:

```
toDesc : {D : U-ix Γ I} → OrnDesc Δ J re-par re-index D
        → U-ix Δ J
toDesc [] = []
toDesc (COD ∷ OD) = toCon COD ∷ toDesc OD

toCon : ∀ {CD : Con-ix Γ V I} {re-par} {W} {re-var : Vxf re-par W V}
      → ConOrnDesc Δ W J re-par re-var re-index CD
      → Con-ix Δ W J
toCon (𝟙 j j~i)               = 𝟙 j
toCon (ρ j j~i COD)           = ρ j (toCon COD)
toCon {re-var = v} (σ S COD) = σ (S ∘ var→par v) (toCon COD)
toCon (Δσ S COD)              = σ S (toCon COD)
```

The other projection reconstructs the ornament NatD-ListD from NatOD:

```
toOrn : {D : U-ix Γ I}
        (OD : OrnDesc Δ J re-par re-index D)
      → Orn re-par re-index D (toDesc OD)
toOrn [] = []
toOrn (COD ∷ OD) = toConOrn COD ∷ toOrn OD

toConOrn : ∀ {CD : Con-ix Γ V I} {re-par} {W} {re-var : Vxf re-par W V}
         → (COD : ConOrnDesc Δ W J re-par re-var re-index CD)
         → ConOrn re-par re-var re-index CD (toCon COD)
toConOrn (𝟙 j j~i)     = 𝟙 j~i
toConOrn (ρ j j~i COD) = ρ j~i (toConOrn COD)
toConOrn (σ S COD)     = σ     (toConOrn COD)
toConOrn (Δσ S COD)    = Δσ    (toConOrn COD)
```

As a consequence, OrnDesc enjoys the features of both Desc and Orn, such as interpretation into a datatype by μ and the conversion to the underlying type by ornForget, by factoring through these projections.

In later sections, we will routinely use OrnDesc to view triples like (NatD, ListD, VecD) as a base type equipped with two patches in sequence.

Part I

Descriptions

If we are going to simplify working with complex sequence types 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 Subsection 5.5 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 other 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, as described in Appendix B.

8 Numerical Representations

Before we dive into descriptions, let us revisit the situation of `ℕ`, `List` and `Vec`. If it was not a coincidence that there are ornaments from `ℕ` to `List` and from `List` to `Vec`, then we can expect to find ornaments beforehand, instead of as a consequence of the definitions of `List` and `Vec`.

e

Rather than finding the properties of `Vec` that were already there, let us view `Vec` as a consequence of the definition of `ℕ` and `lookup`. From `ℕ`, we obtain a trivial type of arrays by reading `lookup` as a prescript:

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

For this definition, the lookup function is simply the identity function on `Lookup`. As this is the prototypical array corresponding to natural numbers, any other array type we define should satisfy all the same properties and laws `Lookup` does, and should in fact be equivalent.

Ik zou uitkijken met 'trivial type of arrays' – wij zijn in ons paper duidelijk dat het over 'flexible arrays' gaat. Misschien uitleggen dat het niet over de 'standaard' arrays gaan, maar een datastructuur met lookups.

We remark that without further assumptions, we cannot use the equality type `≡` for this notion of sameness of types: repeating the definition of a type gives two distinct types with no equality between them. Instead, we import another notion of sameness, known as isomorphisms:

'repeating the definition of a type' - volgens mij heet dit onderscheid

```
record _≃_ 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
```

An `Iso` from `A` to `B` is a map from `A` to `B` with a (two-sided) inverse[8]. In terms of elements, this means that elements of `A` and `B` are in one-to-one correspondence.

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

Now, rather than defining `Vec` “out of the blue” and proving that it is correct or isomorphic to `Lookup`, we can also turn the `Iso` on its head: Starting from the equation that `Vec` is equivalent to `Lookup`, we derive a definition of `Vec` as if solving that equation [HS22]. As a warm-up, we can also derive `Fin` from the fact that `Fin n` should contain `n` elements, and thus be isomorphic to `Σ[ m ∈ ℕ ] m < n`.

To express such a definition by isomorphism, we define:

```
Def : Type → Type
Def A = Σ' Type λ B → A ≃ B

defined-by    : {A : Type} → Def A       → Type
by-definition : {A : Type} → (d : Def A) → A ≃ (defined-by d)
```

using

```
record Σ' (A : Type) (B : A → Type) : Type where
  constructor _use-as-def
  field
    {fst} : A
    snd : B fst
```

The type `Def A` is deceptively simple, after all, there is (up to isomorphism) only one unique term in it! However, when using `Def`initions, the implicit `Σ'` extracts the right-hand side of a proof of an isomorphism, allowing us to reinterpret a proof as a definition.

To keep the resulting `Iso`s readable, we construct them as chains of smaller `Iso`s using a variant of “equational reasoning” [The23; WKS22], which lets us compose `Iso`s while displaying the intermediate steps. In the calculation of `Fin`, we will use the following lemmas

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

In the terminology of Section 4, `⊥-strict` states that “if A is false, then A *is* false”, if we allow reading isomorphisms as “*is*”, while `<-split` states that the set of numbers below $n + 1$ is 1 greater than the set of numbers below n.

Using these, we can calculate[9]

```
Fin-def : ∀ n → Def (Σ[ m ∈ ℕ ] m < n)
Fin-def zero    =  Σ[ m ∈ ℕ ] (m < zero)
                ≃⟨ ⊥-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
```

This gives a different (but equivalent) definition of `Fin` compared to `FinD`: the description `FinD` describes `Fin` as an inductive family, whereas `Fin-def` gives the same definition as a type-computing function [KG16].

[9] Here we make non-essential use of `cong` for type families. In the derivation of `Vec` we use function extensionality, which has to be postulated or brought in via Cubical Agda.

This Def then extracts to a definition of Fin

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

To derive Vec, we will use the isomorphisms

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

which one can compare to the familiar exponential laws. These compose to calculate

```
Vec-def : ∀ A n → Def (Lookup A n)
Vec-def A zero = (Fin zero → A) ≃⟨⟩
                 (⊥ → A)        ≃⟨ ⊥→A≃⊤ ⟩
                 ⊤              ≃-∎ use-as-def

Vec-def A (suc n) = (Fin (suc n) → A) ≃⟨⟩
                    (⊤ ⊎ Fin n → A) ≃⟨ ⊎→≃→× ⟩
                    (⊤ → A) × (Fin n → A) ≃⟨ cong (_× (Fin n → A)) ⊤→A≃A ⟩
                    A × (Fin n → A) ≃⟨ cong (A ×_) (by-definition (Vec-def A n)) ⟩
                    A × (defined-by (Vec-def A n)) ≃-∎ use-as-def
```

which yields us a definition of vectors

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

Vec-Lookup : ∀ A n → Lookup A n ≃ Vec A n
Vec-Lookup A n = by-definition (Vec-def A n)
```

and the Iso to Lookup in one go.

This explains how we can compute a type of lists or arrays (a numerical representation, here, Vec) from a number system (ℕ).

9 Augmented Descriptions

De definitie van number system (record Info) is nieuw en hangt vrij nauw samen met de universe die je gebruikt. Dat is (deels) te begrijpen — je wilt datastructuren beschrijven als ornaments van numbers. Maar nu voelt de presentatie enigsinds backwards — je definieert de Info type die precies past op je descriptions, ipv uit te leggen wat een number system is — een soort specificatie opstellen die los staat van de implementatie adhv descriptions. Hoe

zou je Peano/binary numbers/skew binary numbers/enz beschrijven
hiermee? En waarom is dit de juiste abstractie voor number systems?
(En kun je die vraag beantwoorden zonder te refereren aan
descriptions)

To describe more general numerical representations, we must first describe more general number systems. We do so very loosely, however, allowing for tree-like number systems so long as the values of nodes are linear combinations of the values of subnodes. This generalizes positional number systems such as ℕ and binary numbers, and allows for more exotic number systems, but for example does not include ℕ × ℕ with the Cantor pairing function as a number system.

By requiring that nodes are interpreted as linear combinations of subnodes, we can implement a universe of number systems as a special case of earlier universes by baking the relevant multipliers into the type-formers. Descriptions in the universe of number systems can then both be interpreted to datatypes, and can evaluate their values to ℕ using the multipliers in their structure.

For there to be an ornament between a number system and its numerical representation, the descriptions of both need to live in the same universe. Hence, we will generalize the type of descriptions over information such as multipliers later, rather than defining a new universe of number systems here. The information needed to describe a number system can be separated between the type-formers. Namely, a leaf 𝟙 requires a constant in ℕ, a recursive field ρ requires a multiplier in ℕ, while a field σ will need a function to convert values to ℕ.

To facilitate marking type-formers with specific bits of information, we define

```
record Info : Type where
  field
    𝟙i : Type
    ρi : Type
    σi : (S : Γ & V ⊢ Type) → Type
    δi : Tel τ → Type → Type
```

Compare this with the usual metadata in generics like in Haskell, but then a bit more wild. Also think of annotations on fingertrees.

to record the type of information corresponding to each type-former. We can summarize the information which makes a description into a number system as the following Info:

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

which will then ensure that each 𝟙 and ρ both are assigned a number ℕ, and each σ is assigned a function that converts values of the type of its field to ℕ.

On the other hand, we can also declare that a description needs no further information by:

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

By making the fields of information implicit in the type of descriptions, we can ensure that descriptions from U-ix can be imported into the generalized universe without change.

In the descriptions, the δ type-former, which we will discuss in closer detail in the next section, represents the inclusion of one description in a larger description. When we include another description, this description will also be equipped with extra information, which we allow to be different from the kind of information in the description it is included in. When this happens, we ask that the information on both sides is related by a transformation:

```
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
```

which makes it possible to downcast (or upcast) between different types of information. This, for example, allows the inclusion of a number system DescI Number into an ordinary datatype Desc without rewriting the former as a Desc first.

10 The Universe

We also need to take care that the numerical representations we will construct indeed fit in our universe. The final universe U-ix of Subsection 5.5, while already quite general, still excludes many interesting datastructures. In particular, the encoding of parameters forces recursive type occurrences to have the same applied parameters, ruling out nested datatypes such as (binary) random-access lists [HS22; Oka98]:

```
data Array (A : Type) : Type where
  Nil :                        Array A
  One : A     → Array (A × A) → Array A
  Two : A → A → Array (A × A) → Array A
```

and finger trees [HP06]:

```
data Digit (A : Type) : Type where
  One   : A → Digit A
  Two   : A → A → Digit A
  Three : A → A → A → Digit A

data Node (A : Type) : Type where
  Node2 : A → A → Node A
  Node3 : A → A → A → Node A
```

Kun je aannemelijk maken dat er geen dependently typed encoding bestaat van Finger Trees? Voor binary random access lijsten, perfect trees, en lambda termen bestaan die wel... Of is de construc-

```
data FingerTree (A : Type) : Type where
  Empty  : FingerTree A
  Single : A → FingerTree A

  Deep   : Digit A → FingerTree (Node A) → Digit A
         → FingerTree A
```

Even if we could represent nested types in `U-ix` we would find it still struggles with finger trees: Because adding non-recursive fields modifies the variable telescope, it becomes hard to reuse parts of a description in different places. Apart from that, the number of constructors needed to describe finger trees and similar types also grows quickly when adding fields like `Digit`.

We will resolve these issues as follows. We can describe nested types by allowing parameters to be transformed before they are passed to recursive fields [JG07]. By transforming variables before they are passed to subsequent fields, it becomes possible to hide fields that are not referenced later and to share or reuse constructor descriptions. Finally, by adding a variant of `σ` specialized to descriptions, we can describe composite datatypes more succinctly.

Compare this to Haskell, in which representations are type classes, which directly refer to other types (even to the type itself in a recursive instance). (But that's also just there because in Haskell the type always already exists and they do not care about positivity and termination).

Ik merk dat ik — door de afstand in tijd (een poosje geleden) en
ruimte (best een paar blzs terug) — moeite heb om zo'n grote data
type als DescI te begrijpen. Kun je misschien toch iets meer uitleg
geven? Wat is er veranderd? Welke indices zijn toegevoegd/aangepast?
Er blijft ook een hoop hetzelfde: descriptions zijn lijsten van
constructors. Iets meer uitleg hier is echt nodig om de code te
begrijpen.

Combining these changes, we define the following universe:

```
data DescI (If : Info) (Γ : Tel ⊤) (J : Type) : Type
data ConI (If : Info) (Γ : Tel ⊤) (V : ExTel Γ) (J : Type) : Type
data μ (D : DescI If Γ J) (p : ⟦ Γ ⟧tel tt) : J → Type

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

where the constructors are defined as:

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

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

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

  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) (C : ConI If Γ V J)
    → ConI If Γ V J
```
From this definition, we can recover the ordinary descriptions as
```
Con = ConI Plain
Desc = DescI Plain
```
Let us explain this universe by discussing some of the old and new datatypes we can describe using it. Some of these datatypes do not make use of the full generality of this universe, so we define some shorthands to emulate the simpler descriptions. Using
```
σ+ : (S : Γ & V ⊢ Type) → {If .σi S} → ConI If Γ (V ▷ S) J → ConI If Γ V J
σ+ S {if} C = σ S {if = if} id C

σ- : (S : Γ & V ⊢ Type) → {If .σi S} → ConI If Γ V J → ConI If Γ V J
σ- S {if} C = σ S {if = if} fst C
```
(and the analogues for δ) we emulate unbound and bound fields respectively, and with
```
ρ0 : {if : If .ρi} {V : ExTel Γ} → V ⊢ J → ConI If Γ V J → ConI If Γ V J
ρ0 {if = if} r D = ρ {if = if} r id D
```
we emulate an ordinary (as opposed to nested) recursive field. We can then describe ℕ and List as before
```
NatD : Desc ∅ ⊤
NatD = 𝟙 _
    ∷ ρ0 _ (𝟙 _)
    ∷ []
ListD : Desc (∅ ▷ const Type) ⊤
ListD = 𝟙 _
  ∷ σ- (λ ((_ , A) , _) → A)
  ( ρ0 _ (𝟙 _))
  ∷ []
```
by replacing σ with σ- and ρ with ρ0.

On the other hand, we bind the length of a vector as a field when defining vectors, so there we use σ+ instead:
```
VecD : Desc (∅ ▷ const Type) ℕ
VecD = 𝟙 (const 0)
    ∷ σ- (λ ((_ , A) , _) → A)
    ( σ+ (const ℕ)
    ( ρ0 (λ (_ , (_ , n)) → n)
    ( 𝟙  (λ (_ , (_ , n)) → suc n))))
    ∷ []
```

With the nested recursive field ρ, we can define the type of binary random-access arrays. Recall that for random-access arrays, we have that an array with parameter `A` contains zero, one, or two values of `A`, but the recursive field must contain an array of twice the weight. Hence, the parameter passed to the recursive field is `A times A`, for which we define

```
Pair : Type → Type
Pair A = A × A
```

Passing `Pair` to `rho` we can define random access lists:

```
RandomD : Desc (∅ ▷ const Type) ⊤
RandomD = 𝟙 _
        ∷ σ- (λ ((_ , A) , _) → A)
        ( ρ _ (λ (_ , A) → (_ , Pair A))
        ( 𝟙 _))
        ∷ σ- (λ ((_ , A) , _) → A)
        ( σ- (λ ((_ , A) , _) → A)
        ( ρ _ (λ (_ , A) → (_ , Pair A))
        ( 𝟙 _)))
        ∷ []
```

To represent finger trees, we first represent the type of digits `Digit`:

reminder to cite this here if I end up not referencing finger trees earlier.

```
DigitD : Desc (∅ ▷ const Type) ⊤
DigitD = σ- (λ ((_ , A) , _) → A)
       ( 𝟙 _)
       ∷ σ- (λ ((_ , A) , _) → A)
       ( σ- (λ ((_ , A) , _) → A)
       ( 𝟙 _))
       ∷ σ- (λ ((_ , A) , _) → A)
       ( σ- (λ ((_ , A) , _) → A)
       ( σ- (λ ((_ , A) , _) → A)
       ( 𝟙 _)))
       ∷ []
```

We can then define finger trees as a composite type from `Digit`:

```
FingerD : Desc (∅ ▷ const Type) ⊤
FingerD = 𝟙 _
        ∷ σ- (λ ((_ , A) , _) → A)
        ( 𝟙 _)
        ∷ δ _ (λ (p , _) → p) DigitD
        ( ρ _ (λ (_ , A) → (_ , Node A))
        ( δ _ (λ (p , _) → p) DigitD
        ( 𝟙 _)))
        ∷ []
```

Here, the fact that the first `δ-` drops its field from the telescope makes it possible to reuse of `Digit` in the second `δ-`.

These descriptions can be instantiated as before by taking the fixpoint[10]

```
data μ D p where
```

[10]Note that these (obviously?) ignore the `Info` of a description.

```
con : ∀ {i} → ⟦ D ⟧D (μ D) p i → μ D p i
```
of their interpretations as functors
```
⟦_⟧C : ConI If Γ V J → ( ⟦ Γ ⟧tel tt → J → Type)
                     → ⟦ Γ & V ⟧tel  → J → Type
⟦ 𝟙 j       ⟧C X pv         i = i ≡ j pv
⟦ ρ j f D   ⟧C X pv@(p , v) i = X (f p) (j pv) × ⟦ D ⟧C X pv i
⟦ σ S h D   ⟧C X pv@(p , v) i = Σ[ s ∈ S pv ] ⟦ D ⟧C X (p , h (v , s)) i
⟦ δ j g R D ⟧C X pv         i = Σ[ s ∈ μ R (g pv) (j pv) ] ⟦ D ⟧C X pv i

⟦_⟧D : DescI If Γ J → ( ⟦ Γ ⟧tel tt → J → Type)
                    → ⟦ Γ ⟧tel tt   → J → Type
⟦ []    ⟧D X p i = ⊥
⟦ C ∷ D ⟧D X p i = (⟦ C ⟧C X (p , tt) i) ⊎ (⟦ D ⟧D X p i)
```
In this universe, we also need to insert the transformations of parameters `f` in `ρ` and the transformations of variables `h` in `σ` and `δ`.

Part II
Ornaments

In the framework of `DescI` in the last section, we can write down a number system and its meaning as the starting point of the construction of a numerical representation. To write down the generic construction of those numerical representations, we will need a language in which we can describe modifications on the number systems.

In this section, we will describe the ornamental descriptions for the `DescI` universe, and explain their working by means of examples. We omit the definition of the ornaments, since we will only construct new datatypes, rather than relate pre-existing types.

Maybe, I will throw the ornaments into the appendix along with the conversion from ornamental description to ornament

11 Ornamental descriptions

These ornamental descriptions take the same shape as those in Section 7, generalized to handle nested types, variable transformations, and composite types. Like the interpretation of a description `DescI`, ornaments also completely ignore the `Info` of a `DescI`.

do we need to remark more?

We will define `OrnDesc If′ Δ c J i D` to represent the ornaments building on top of a base description `D`, yielding descriptions with information `If′`, parameters `Δ`, and indices `J`:
```
data OrnDesc {If} (If′ : Info) (Δ : Tel ⊤)
             (c : Cxf Δ Γ) (J : Type) (i : J → I)
             : DescI If Γ I → Type where
  []  : OrnDesc If′ Δ c J i []
  _∷_ : ConOrnDesc If′ {c = c} id i {If = If} CD
```

```
        → OrnDesc If′ Δ c J i D
        → OrnDesc If′ Δ c J i (CD ∷ D)
```

We use ~ to write down pointwise equality of functions, which in this case all are commutativity squares. Since ConI allows the transformation of variable telescopes, we have to dedicate a lot of lines to writing down commutativity squares for variables, which along with the generally high number of arguments and implicits[11] makes the definition rather dry and long. However, these squares involving Vxf can generally ignored, as witnessed by the Oσ+ and Oσ- variants of the constructors, which automatically fill those squares in the common cases of binding or ignoring fields.

The constructor ornaments can be split into three segments: structure-preserving ornaments, extensions, and composition. The structure-preserving ornaments are

```
data ConOrnDesc (If′ : Info) {c : Cxf Δ Γ}
                (v : VxfO c W V) (i : J → I)
                : ConI If Γ V I → Type where
  𝟙 : {i′ : Γ & V ⊢ I} (j′ : Δ & W ⊢ J)
    → i ∘ j′ ~ i′ ∘ over v
    → {if : If .𝟙i} {if′ : If′ .𝟙i}
    → ConOrnDesc If′ v i (𝟙 {If} {if = if} i′)

  ρ : {i′ : Γ & V ⊢ I} (j′ : Δ & W ⊢ J)
      {g : Cxf Γ Γ} (h : Cxf Δ Δ)
    → g ∘ c ~ c ∘ h
    → i ∘ j′ ~ i′ ∘ over v
    → {if : If .ρi} {if′ : If′ .ρi}
    → ConOrnDesc If′ v i CD
    → ConOrnDesc If′ v i (ρ {If} {if = if} i′ g CD)

  σ : (S : Γ & V ⊢ Type)
      {g : Vxf Γ (V ▷ S) V′} (h : Vxf Δ (W ▷ (S ∘ over v)) W′)
      (v′ : VxfO c W′ V′)
    → (∀ {p} → g ∘ VxfO-▷ v S ~ v′ {p = p} ∘ h)
    → {if : If .σi S} {if′ : If′ .σi (S ∘ over v)}
    → ConOrnDesc If′ v′ i CD
    → ConOrnDesc If′ v i (σ {If} S {if = if} g CD)

  δ : (R : DescI If″ Θ K) (j : Γ & V ⊢ K) (t : Γ & V ⊢ ⟦ Θ ⟧tel tt)
    → {if : If .δi Θ K} {iff : InfoF If″ If}
      {if′ : If′ .δi Θ K} {iff′ : InfoF If″ If′}
    → ConOrnDesc If′ v i CD
    → ConOrnDesc If′ v i (δ {If} {if = if} {iff = iff} j t R CD)
```

These represent the ornaments in which the base description and the target description share the same field, up to conversions of parameters, variables, and indices.

[11] Of which even more are hidden!

The ornaments extending structure are

```
Δσ : (S : Δ & W ⊢ Type) (h : Vxf Δ (W ▷ S) W')
       (v' : VxfO c W' V)
    → (∀ {p} → v ∘ fst ~ v' {p = p} ∘ h)
    → {if' : If' .σi S}
    → ConOrnDesc If' v' i CD
    → ConOrnDesc If' v i CD

Δδ : (R : DescI If″ Θ J) (j : W ⊢ J) (t : W ⊢ ⟦ Θ ⟧tel tt)
    → {if' : If' .δi Θ J} {iff' : InfoF If″ If'}
    → ConOrnDesc If' v i CD
    → ConOrnDesc If' v i CD
```

representing the insertion of fields in the target which are not present in the base description.

Finally, the ornament makes it possible compose an ornament onto a `δ` in the base description.

Compared to the previous ornaments, we have the new constructors `δ`, `Δδ` and `δ•`, where the first two are analogues of `σ` and `Δσ`. The `δ•` constructor states that an ornamental description from a description `R` and a (constructor) ornamental description from `CD` can be composed to form an ornamental description from the composition (in the sense of the `δ` type-former) of `CD` with `R`. The new commutativity squares in all the constructors both ensure the existence of functions such as like for the simpler ornaments, and that these ornamental descriptions indeed still induce ornaments.

The precise meaning of ornamental descriptions as descriptions is given by the conversion:

```
toDesc : {v : Cxf Δ Γ} {i : J → I} {D : DescI If Γ I}
       → OrnDesc If' Δ v J i D → DescI If' Δ J
toDesc [] = []
toDesc (CO ∷ O) = toCon CO ∷ toDesc O

toCon  : {c : Cxf Δ Γ} {v : VxfO c W V} {i : J → I} {D : ConI If Γ V I}
       → ConOrnDesc If' v i D → ConI If' Δ W J
toCon (𝟙 j' x {if' = if})
  = 𝟙 {if = if} j'

toCon (ρ j' h x x₁ {if' = if} CO)
  = ρ {if = if} j' h (toCon CO)

toCon {v = v} (σ S h v' x {if' = if} CO)
  = σ (S ∘ over v) {if = if} h (toCon CO)

toCon {v = v} (δ R j t {if' = if} {iff' = iff} CO)
  = δ {if = if} {iff = iff} (j ∘ over v) (t ∘ over v) R (toCon CO)

toCon (Δσ S h v' x {if' = if} CO)
  = σ S {if = if} h (toCon CO)
```

```
toCon (Δδ R j t {if′ = if} {iff′ = iff} CO)
  = δ {if = if} {iff = iff} j t R (toCon CO)

toCon (•δ m f∧ RR′ p₁ p₂ {if′ = if} {iff′ = iff} CO)
  = δ {if = if} {iff = iff} m f∧ (toDesc RR′) (toCon CO)
```

which makes use of the implicit `If′` fields in the constructor ornaments to reconstruct the information on the target description.

But let us make the uses of `OrnDesc` more clear by means of examples, where we make use of the variants of some ornaments specialized to binding or ignoring fields:

```
Oσ+ : (S : Γ & V ⊢ Type) {CD : ConI If Γ V′ I} {h : Vxf _ _ _}
  → {if : If .σi S} {if′ : If′ .σi (S ∘ over v)}
  → ConOrnDesc If′ (h ∘ VxfO-▷ v S) i CD
  → ConOrnDesc If′ v i (σ {If} S {if = if} h CD)
Oσ+ S {h = h} {if′ = if′} CO
  = σ S id (h ∘ VxfO-▷ v S) (λ _ → refl) {if′ = if′} CO

Oσ- : (S : Γ & V ⊢ Type) {CD : ConI If Γ V I}
  → {if : If .σi S} {if′ : If′ .σi (S ∘ over v)}
  → ConOrnDesc If′ v i CD
  → ConOrnDesc If′ v i (σ {If} S {if = if} fst CD)
Oσ- S {if′ = if′} CO = σ S fst v (λ _ → refl) {if′ = if′} CO
```

With these we can give the familiar ornamental description from `List` to `Vec`:

```
VecOD : OrnDesc Plain (∅ ▷ const Type) id ℕ ! ListD
VecOD = (𝟙 (const zero) (const refl))
      ∷ (OΔσ+ (const ℕ)
      ( Oσ- (λ ((_ , A) , _) → A)
      ( Oρ0 (λ (_ , (_ , n)) → n) (const refl)
      ( 𝟙 (λ (_ , (_ , n)) → suc n) (const refl)))))
      ∷ []
```

Using the new flexibility in ρ, we can now start from a description of binary numbers:

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

and describe random access lists as an ornament from binary numbers:

```
RandomOD : OrnDesc Plain (∅ ▷ const Type) ! ⊤ id LeibnizD
RandomOD = 𝟙 _ (const refl)
         ∷ OΔσ- (λ ((_ , A) , _) → A)
         ( ρ _ (λ (_ , A) → (_ , Pair A)) (const refl) (const refl)
         ( 𝟙 _ (const refl)))
         ∷ OΔσ- (λ ((_ , A) , _) → A)
         ( OΔσ- (λ ((_ , A) , _) → A)
```

```
              ( ρ _ (λ (_ , A) → (_ , Pair A)) (const refl) (const refl)
              ( 𝟙 _ (const refl))))
              ∷ []
```

Likewise, we can define phalanges as

```
ThreeD : Desc ∅ ⊤
ThreeD = 𝟙 _ ∷ 𝟙 _ ∷ 𝟙 _ ∷ []

PhalanxD : Desc ∅ ⊤
PhalanxD = 𝟙 _
         ∷ 𝟙 _
         ∷ δ _ _ ThreeD
         ( ρ0 _
         ( δ _ _ ThreeD
         ( 𝟙 _)))
         ∷ []
```

By giving an ornament turning Three into Digits

```
DigitOD : OrnDesc Plain (∅ ▷ const Type) ! ⊤ id ThreeD
DigitOD = OΔσ- (λ ((_ , A) , _) → A)
        ( 𝟙 _ (const refl))
        ∷ OΔσ- (λ ((_ , A) , _) → A)
        ( OΔσ- (λ ((_ , A) , _) → A)
        ( 𝟙 _ (const refl)))
        ∷ OΔσ- (λ ((_ , A) , _) → A)
        ( OΔσ- (λ ((_ , A) , _) → A)
        ( OΔσ- (λ ((_ , A) , _) → A)
        ( 𝟙 _ (const refl))))
        ∷ []
```

we can then use δ• to compose Digits into phalanges, making binary fingertrees

```
FingerOD : OrnDesc Plain (∅ ▷ const Type) ! ⊤ id PhalanxD
FingerOD = 𝟙 _ (const refl)
         ∷ OΔσ- (λ ((_ , A) , _) → A)
         ( 𝟙 _ (const refl))
         ∷ •δ _ (λ (p , _) → p) DigitOD (λ _ _ → refl) (λ _ _ → refl)
         ( ρ _ (λ (_ , A) → (_ , Pair A)) (const refl) (const refl)
         ( •δ _ (λ (p , _) → p) DigitOD (λ _ _ → refl) (λ _ _ → refl)
         ( 𝟙 _ (const refl))))
         ∷ []
```

Part III
Numerical representations

The ornamental descriptions of the last section, together with the descriptions and number systems from before, complete the toolset we will use to construct numerical representations as ornaments.

To summarize, we using `DescI Number` to represent number systems, we will paraphrase the calculation of Section 8 as an ornament rather than a direct definition. In fact, we have already seen ornaments to numerical representations before, such as `ListOD` and `RandomOD`. Generalizing those ornaments, we construct numerical representations by means of an ornament-computing function, sending number systems to the ornamental descriptions that describe their numerical representations.

12 Generic numerical representations

In this section, we will demonstrate how we can use the ornamental descriptions to generically compute numerical representations.

The reasoning here proceeds differently from the calculation of `Vec` from `ℕ`. Indeed, here we will directly construct datatypes via ornamental descriptions, rather than deriving them step-by-step using isomorphism reasoning. Nevertheless, the choices of fields depending on the analysis of a number system follow the same strategy. We will first present the unindexed numerical representations, explaining case-by-case which fields it adds and why. Then, we will demonstrate the indexed numerical representations as an ornament on top of the unindexed variant, and how the indices built up incrementally as we descend over the structure of the number system.

Recall the "natural numbers"-information `Number`, which gets its semantics from the conversion to `ℕ`:

```
value : {D : DescI Number Γ τ} → ∀ {p} → μ D p tt → ℕ
```

which is defined by generalizing over the inner information bundle and folding using

```
value-desc : (D : DescI If Γ τ) → ∀ {a b} → ⟦ D ⟧D (λ _ _ → ℕ) a b → ℕ
value-con : (C : ConI If Γ V τ) → ∀ {a b} → ⟦ C ⟧C (λ _ _ → ℕ) a b → ℕ

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

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

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

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

value-con (δ {if = if} {iff = iff} j g R C) (r , x)
    with ϕ .δf _ _ if
... | refl , refl , k
    = k * value-lift R (ϕ ∘InfoF iff) r + value-con C x
```

The choice of interpretation restricts the numbers to the class of numbers which are evaluated as linear combinations of digits[12]. This class certainly does not include all interesting number systems, but does include many systems that have associated arrays[13].

Explain better from here

We let this interpretation into ℕ guide the construction of the associated numerical representation. In each case, we follow the computation in `value` by inserting vectors of sizes corresponding to the weights of the number system:

```
trieifyOD : (D : DescI Number ∅ τ) → OrnDesc Plain (∅ ▷ const Type) ! τ ! D
trieifyOD D = trie-desc D id-InfoF
  module trieifyOD where
  trie-desc : (D : DescI If ∅ τ) → InfoF If Number
            → OrnDesc Plain (∅ ▷ const Type) ! τ ! D

  trie-con  : {f : VxfO ! W V} (C : ConI If ∅ V τ) → InfoF If Number
            → ConOrnDesc {Δ = ∅ ▷ const Type} {W = W} {J = τ} Plain f ! C

  trie-desc [] ϕ = []
  trie-desc (C ∷ D) ϕ = trie-con C ϕ ∷ trie-desc D ϕ

  trie-con (𝟙 {if = k} j) ϕ
    = OΔσ- (λ ((_ , A) , _) → Vec A (ϕ .𝟙f k))
    ( 𝟙 _ (const refl))

  trie-con (ρ {if = k} j g C) ϕ
    = ρ _ (λ (_ , A) → (_ , Vec A (ϕ .ρf k))) (const refl) (const refl)
    ( trie-con C ϕ)

  trie-con (σ S {if = if} h C) ϕ
    = Oσ+ S
    ( OΔσ- (λ ((_ , A) , _ , s) → Vec A (ϕ .σf _ if _ s))
    ( trie-con C ϕ))

  trie-con {f = f} (δ {if = if} {iff = iff} j g R C) ϕ
    with ϕ .δf _ _ if
  ... | refl , refl , k
    = •δ ! (λ { ((_ , A) , _) → (_ , Vec A k) }) (trie-desc R (ϕ ∘InfoF iff))
          (λ _ _ → refl) (λ _ _ → refl)
    ( trie-con C ϕ)
```

In the case of a leaf 𝟙 of weight `k`, we insert a vector of size `k`. Similarly, in a field σ, where the weight is determined by a value `s` of `S`, we insert a vector of the weight corresponding to the value of `s`. Note that the actual value/number of elements a leaf or field contributes depends on the preceding multipliers of

[12] An arbitrary `Number` system 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).

[13] Notably, arbitrary polynomials also have numerical representations, interpreting multiplication as precomposition.

recursive fields: a recursive field of a number can have a weight `k`, so we multiply the number of elements in a recursive sequence by wrapping the parameter in a vector of size `k`. By roughly the same reasoning we pass the trieification of a subdescription `R` the parameter wrapped in a vector, which we compose into the current numerical description by using the ornament `•δ`. Since `R` can have a different `Info`, we generalized the whole construction over `ϕ : InfoF If Number`.

As an example, let us define `PhalanxD` as a number system and walk through the computation of its `trieifyOD`. We define

```
ThreeND : DescI Number ∅ ⊤
ThreeND = 𝟙 {if = 1} _
        ∷ 𝟙 {if = 2} _
        ∷ 𝟙 {if = 3} _
        ∷ []

PhalanxND : DescI Number ∅ ⊤
PhalanxND = 𝟙 {if = 0} _
          ∷ 𝟙 {if = 1} _
          ∷ δ {if = refl , refl , 1} {iff = id-InfoF} _ _ ThreeND
          ( ρ0 {if = 2} _
          ( δ {if = refl , refl , 1} {iff = id-InfoF} _ _ ThreeND
          ( 𝟙 {if = 0} _)))
          ∷ []
```

Now, we see that applying `trieifyOD` sends leaves with a value of `k` to `Vec A k`, so applying it to `DigitND` yields

```
DigitOD : OrnDesc Plain (∅ ▷ const Type) ! ⊤ id ThreeND
DigitOD = OΔσ- (λ ((_ , A) , _) → Vec A 1)
          ( 𝟙 _ (const refl))
          ∷ OΔσ- (λ ((_ , A) , _) → Vec A 2)
          ( 𝟙 _ (const refl))
          ∷ OΔσ- (λ ((_ , A) , _) → Vec A 3)
          ( 𝟙 _ (const refl))
          ∷ []
```

which is equivalent to the `DigitOD` from before, up to expanding a vector of `k` elements into `k` fields. The same happens for the first two constructors of `PhalanxND`, replacing them with an empty vector and a one-element vector respectively. The `ThreeND` in the last constructor gets trieified to `DigitOD′` and composed by `O•δ+`, and the recursive field gets replaced by a recursive field nesting over vectors of length. Again, this is equivalent to `FingerOD`, up to wrapping values in length one vectors, replacing `Pair` with a two-element vector, and inserting empty vectors.

Explain better until here

Like how `List` has an ornament `VecOD` to its ℕ-indexed variant `Vec`, the numerical representation `trieifyOD D` has an ornament `itrieifyOD D` to its `D`-indexed variant:

```
itrieifyOD : (N : DescI Number ∅ ⊤)
           → OrnDesc Plain (∅ ▷ const Type)
             id (μ N tt tt) ! (toDesc (trieifyOD N))
```

```
itrieifyOD N = itrie-desc N N (λ _ _ → con) id-InfoF
```

Continuing the analogy to `VecOD`, this ornament adds fields reflecting the recursive indices, and threads the partially applied constructor `n` of `N` through the resulting description. In addition to generalizing over `If` to facilitate the `δ` case, as in `trieifyOD`, we also generalize over the index type `N'`. When mapping over descriptions, the choice of constructor also selects the corresponding constructor of `N'`

```
itrie-desc : ∀ {If} (N' : DescI If ∅ ⊤) (D : DescI If ∅ ⊤)
           (n : ⟦ D ⟧D (μ N') ⇉ μ N') (ϕ : InfoF If Number)
           → OrnDesc Plain (∅ ▷ const Type)
             id (μ N' tt tt) ! (toDesc (trie-desc D ϕ) )
itrie-desc N' [] n ϕ      = []
itrie-desc N' (C ∷ D) n ϕ = itrie-con N' C (λ p w x → n _ _ (inj₁ x)) ϕ
                          ∷ itrie-desc N' D (λ p w x → n _ _ (inj₂ x)) ϕ
```

We define `itrie-con` by induction on `C`, consuming bound values one-by-one as arguments for the selected constructor `n`, which will then produce the actual indices at the leaves. Since we are continuing where `trie-con` left off, we can copy most fields

```
itrie-con : ∀ {If} (N' : DescI If ∅ ⊤) {f : VxfO id W V}
          {g : VxfO ! V U} (C : ConI If ∅ U ⊤)
          (n : ∀ p w → ⟦ C ⟧C (μ N') (tt , g (f {p = p} w)) _ → μ N' tt tt)
          (ϕ : InfoF If Number)
          → ConOrnDesc {Δ = ∅ ▷ const Type} {W = W} {J = μ N' tt tt} Plain
            {c = id} f ! (toCon (trie-con {f = g} C ϕ))
itrie-con N' (𝟙 {if = k} j) n ϕ
  = Oσ- _
  ( 𝟙 (λ { (p , w) → n p w refl }) (const refl))

itrie-con N' (ρ {if = k} j g C) n ϕ
  = OΔσ+ (λ _ → μ N' tt tt)
  ( ρ (λ { (p , w , i) → i }) (λ { (_ , A) → _ })
      (λ _ → refl) (λ _ → refl)
  ( itrie-con N' C (λ { p (w , i) x → n p w (i , x) }) ϕ))

itrie-con N' (σ S {if = if} h C) n ϕ
  = Oσ+ (S ∘ over _)
  ( Oσ- _
  ( itrie-con N' C (λ { p (w , s) x → n p w (s , x) }) ϕ))

itrie-con N' {f = f} (δ {if = if} {iff = iff} j g R C) n ϕ
  with ϕ .δf _ _ if
... | refl , refl , k
  = OΔσ+ (λ _ → μ R tt tt)
  ( •δ (λ { (p , w , i) → i }) (λ ((_ , A) , _) → (_ , Vec A k))
        (itrie-desc R R (λ _ _ → con) (ϕ ∘InfoF iff))
        (λ _ _ → refl) (λ _ _ → refl)
  ( itrie-con N' C (λ { p (w , i) x → n p w (i , x) }) ϕ))
```

Only in the case for ρ and δ do we add fields, which are both promptly passed as expected indices to the next field using `λ { (p , w , i) → i }`. For δ, since `itrie-desc R` will be `R`-indexed, we add a field of `R` rather than `N'`. The values of all fields, including σ are passed to `n`; since `n` starts as one constructor `C` of `N'`, when we arrive at 𝟙, the final argument of `n` can be filled with simply `refl` to determine the actual index.

Since the `N'`-index bound in the ρ case forces the number of elements in the recursive field, the value in the σ case corresponds to the number of elements added after this field, and the `R`-index bound in the δ case likewise forces the number of elements in the subdescription, we know that when we arrive at 𝟙, the total number of elements is exactly given by `n`, and thus `itrie-con` is correct. In turn, we conclude that `itrie-desc` and `itrieifyOD` correctly construct indexed numerical representations.

Proof is left as exercise to the reader. Hint Σ-descriptions will come in handy.

Part IV
Discussion

13 δ is conservative

We define our universe `DescI` with δ as a former of fields with known descriptions, because this makes it easier to write down `trieifyOD`, even though δ is redundant. If more concise universes and ornaments are preferable, we can actually get all the features of δ and ornaments like ∙δ by describing them using σ, annotations, and other ornaments.

Example? I think the explanation of itrieifyOD is extensive enough to not warrant a repetition of fingerod in the indexed case.

Indeed, rather than using δ to add a field from a description `R`, we can simply use σ to add `S = μ R`, and remember that `S` came from `R` in the information

```
Delta : Info
Delta .σi {Γ = Γ} {V = V} S
  = Maybe (
    Σ[ Δ ∈ Tel ⊤ ] Σ[ J ∈ Type ] Σ[ j ∈ Γ & V ⊢ J ]
    Σ[ g ∈ Γ & V ⊢ ⟦ Δ ⟧tel tt ] Σ[ D ∈ DescI Delta Δ J ]
    (∀ pv → S pv ≡ liftM2 (μ D) g j pv))
```

We can then define δ as a pattern synonym matching on the `just` case, and σ matching on the `nothing` case.

Recall that the ornament ∙δ lets us compose an ornament from `D` to `D'` with an ornament from `R` to `R'`, yielding an ornament from `δ D R` to `δ D' R'`. This ornament can be modelled by first adding a new field `μ R'`, and then deleting the original `μ R` field. The ornament ∇ [Ko14] allows one to provide a default value for a field, deleting it from the description. Hence, we can model ∙δ by binding a value `r'` of `μ R'` with `OΔσ+` and deleting the field `μ R` using a default value computed by `ornForget`.

This concludes a bunch of things, including this thesis. Combine conclusion and discussion? "We did X, but there still are many improvements that could be made"

14 Indices do not depend on parameters

In `DescI`, we represent the indices of a description as a single constant type, as opposed to an extension of the parameter telescope [EC22]. This simplification keeps the treatment of ornaments and numerical representations more to the point, but rules out types like the identity type `≡`. Another consequence of not allowing indices to depend on parameters is that algebraic ornaments [McB14] can not be formulated in `OrnDesc` in their fully general form.

By replacing index computing functions `Γ & V ⊢ I` with dependent functions

```
_&_⊧_ : (Γ : Tel ⊤) (V I : ExTel Γ) → Type
Γ & V ⊧ I = (pv : ⟦ Γ & V ⟧tel) → ⟦ I ⟧tel (fst pv)
```

we can allow indices to depend on parameters in our framework. As a consequence, we have to modify nested recursive fields to ask for the index type `⟦ I ⟧tel` precomposed with `g : Cxf Γ Γ`, and we have to replace the square like `i ∘ j' ~ i' ∘ over v` in the definition of ornaments with heterogeneous squares.

15 Σ-descriptions are more natural for expressing finite types

Due to our representation of types as sums of products, representing the finite types of arbitrary number systems quickly becomes hard. Consider the binary numbers from before

```
data Leibniz : Type where
  0b      : Leibniz
  1b_ 2b_ : Leibniz → Leibniz
```

The finite type associated to `Leibniz` then has more constructors than `Leibniz`:

```
data FinB : Leibniz → Type where
  0/1     : FinB (1b n)
  0/2 1/2 : FinB (2b n)

  0-1b_ 1-1b_ : FinB n → FinB (1b n)
  0-2b_ 1-2b_ : FinB n → FinB (2b n)
```

In general, given a description of a number system `N`, the number of constructors of the finite type `FinN` of `N` depends directly on the interpretation of `N`, preventing the construction `FinN` by simple recursion on `DescI` (that is, without passing around lists of constructors instead). Furthermore, since our definition of ornaments insists ornaments preserve the number of constructors, there cannot be an ornament from an arbitrary number system to its finite type.

The apparent asymmetry between number systems and finite types stems from the definition of `σ` in `DescI`. In `DescI` and similar sums-of-products universes [EC22; Sij16], the remainder of a constructor `C` after a `σ S` simply has its context extended by `S`. In contrast, a Σ-descriptions universe [eff20; KG16; McB14] (in the terminology of [Sij16]) encodes a dependent field `(s : S)` by asking for a function `C` assigning values `s` to descriptions.

Maybe example, maybe one can be expected to gather this from the confusingly named U-sop in background.

In comparison, a sums-of-products universe keeps out some more exotic descriptions[14] which do not have an obvious associated Agda datatype. As a consequence, this also prevents us from introducing new branches inside a constructor.

If we instead started from Σ-descriptions, taking functions into `DescI` to encode dependent fields, we could compute a "type of paths" in a number system by adding and deleting the appropriate fields. Consider the universe

```
data Σ-Desc (I : Type) : Type where
  𝟙 : I → Σ-Desc I
  ρ : I → Σ-Desc I → Σ-Desc I
  σ : (S : Type) → (S → Σ-Desc I) → Σ-Desc I
```

In this universe we can present the binary numbers as

```
LeibnizΣD : Σ-Desc ⊤
LeibnizΣD = σ (Fin 3) λ
  { zero             → 𝟙 _
  ; (suc zero)       → ρ _ (𝟙 _)
  ; (suc (suc zero)) → ρ _ (𝟙 _) }
```

The finite type for these numbers can be described by

```
FinBΣD : Σ-Desc Leibniz
FinBΣD = σ (Fin 3) λ
  { zero             → σ (Fin 0) λ _ → 𝟙 0b
  ; (suc zero)       → σ Leibniz λ n → σ (Fin 2) λ
    { zero       → σ (Fin 1) λ _ →       𝟙 (1b n)
    ; (suc zero) → σ (Fin 2) λ _ → ρ n ( 𝟙 (1b n)) }
  ; (suc (suc zero)) → σ Leibniz λ n → σ (Fin 2) λ
    { zero       → σ (Fin 2) λ _ →       𝟙 (2b n)
    ; (suc zero) → σ (Fin 2) λ _ → ρ n ( 𝟙 (2b n)) } }
```

Since this description of `FinB` largely has the same structure as `Leibniz`, and as a consequence also the numerical representation associated to `Leibniz`, this would simplify proving that the indexed numerical representation is indeed equivalent to the representable representation (the maps out of `FinB`). In a more flexible framework ornaments, we can even describe the finite type as an ornament on the number system.

16 Branching numerical representations

The numerical representations we construct via `trieifyOD` look like random-access lists and finger trees: the structures have central chains, storing the elements of a node in trees of which the depth increases with the level of the node.

In contrast, structures like Braun trees, as Hinze and Swierstra [HS22] compute from binary numbers, reflect the weight of a node by branching them-

[14] Consider the constructor `σ ℕ λ n → power ρ n 𝟙` which takes a number `n` and asks for `n` recursive fields (where `power f n x` applies `f n` times to `x`). This description, resembling a rose tree, does not (trivially) lie in a sums-of-products universe.

selves. Because this kind of branching is uniform, i.e., each branch looks the same, we can still give an equivalent construction. By combining `trieifyOD` and `itrieifyOD`, and using to apply `ρ` `k`-fold in the case of `ρ {if = k}`, rather than over `k`-element vectors, we can replicate the structure of a Braun tree from `BinND`. However, if we use the Σ-descriptions we discussed above, we can more elegantly present these structures by adding an internal branch over `Fin k`.

17 Indexed numerical representations are not algebraic ornaments

Algebraic ornaments [McB14], generalize observations such as that `Vec` is an indexed variant of `List`, in a single definition `aOoA` (the algebraic ornament of the ornamental algebra). The construction of that ornament takes an ornament between types `A` and `B`, and returns an ornament from `B` to a type indexed over `A`, representing "`B`s of a given underlying `A`". Instantiating this for naturals, lists and vectors, the algebraic ornament takes the ornament from naturals to lists, and returns an ornament from lists to vectors, by which vectors are lists of a fixed length.

While we gave an explicit ornament `itrieifyOD` on `trieifyOD`, we might expect `itrieifyOD` to be the algebraic ornament of `trieifyOD`. However, this fails if we want to describe composite types like `FingerTree` (unless we first flatten `Digit` into the description of `FingerTree`): The algebraic ornament (obviously) preserves a `σ`, so it cannot convert the unindexed numerical representation under a `δ` to the indexed variant. This means that the algebraic ornament on `FingerTree = toDesc (trieifyOD PhalanxND)` would only index the outer structure, leaving the `Digit` fields unindexed.

Nevertheless, we expect that if one defines `indexO` by inlining `ornAlg` into `aOoA`, the definition of `indexO` can be modified to apply itself in the case of `•δ`. Then, applying `indexO` to `trieifyOD` should coincide with `itrieifyOD`.

18 No RoseTrees

In `DescI`, we encode nested types by allowing nesting over a function of parameters `Cxf Γ Γ`. This is less expressive than full nested types, which may also nest a recursive field under a strictly positive functor. For example, rose trees

```
data RoseTree (A : Type) : Type where
  rose : A → List (RoseTree A) → RoseTree A
```

cannot be directly expressed as a `DescI`[15]. Can still do

If we were to describe full nested types, allowing applications of functors in the types of recursive arguments, we would have to convince Agda that these functors are indeed positive, possibly by using polarity annotations[16]. Alterna-

[15] And, since `DescI` does not allow for higher-order inductive arguments like Escot and Cockx [EC22], we can also not give an essentially equivalent definition.

[16] `https://github.com/agda/agda/pull/6385`

tively, we could encode strictly positive functors in a separate universe, which only allows using parameters in strictly positive contexts [Sij16]. Finally, we could modify `DescI` in such a way that we can decide if a description uses a parameter strictly positively, for which we would modify `ρ` and `σ`, or add variants of `ρ` and `σ` restricted to strictly positive usage of parameters.

19 No levitation

Since our encoding does not support higher-order inductive arguments, let alone definitions by induction-recursion, there is no code for `DescI` in itself. Such self-describing universes have been described by Chapman et al. [Cha+10], and we expect that the other features of `DescI`, such as parameters, nesting, and composition, would not obstruct a similar levitating variant of `DescI`. Due to the work of Dagand and McBride [DM14], ornaments might even be generalized to inductive-recursive descriptions.

If that is the case, then modifications of universes like `Info` could be expressed internally. In particular, rather than defining `DescI` such that it can describe datatypes with the information of, e.g., number systems, `DescI` should be expressible as an ornamental description on `Desc`, in contrast to how `Desc` is an instance of `DescI` in our framework. This would allow treating information explicitly in `DescI`, and not at all in `Desc`.

Furthermore, constructions like `trieifyOD`, which have the recursive structure of a fold over `DescI`, could indeed be expressed by instantiating `fold` to `DescI`.

Maybe a bit too dreamy.

References

[AMM07] Thorsten Altenkirch, Conor McBride, and Peter Morris. "Generic Programming with Dependent Types". In: Nov. 2007, pp. 209–257. ISBN: 978-3-540-76785-5. DOI: `10.1007/978-3-540-76786-2_4`.

[Bru91] N.G. de Bruijn. "Telescopic mappings in typed lambda calculus". In: *Information and Computation* 91.2 (1991), pp. 189–204. ISSN: 0890-5401. DOI: `https://doi.org/10.1016/0890-5401(91)90066-B`. URL: `https://www.sciencedirect.com/science/article/pii/089054019190066B`.

[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`.

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

[DM14] Pierre-Évariste Dagand and Conor McBride. "Transporting functions across ornaments". In: *Journal of Functional Programming* 24.2-3 (Apr. 2014), pp. 316–383. DOI: `10.1017/s0956796814000069`. URL: `https://doi.org/10.1017%2Fs0956796814000069`.

[DS06] Peter Dybjer and Anton Setzer. "Indexed induction–recursion". In: *The Journal of Logic and Algebraic Programming* 66.1 (2006), pp. 1–49. ISSN: 1567-8326. DOI: `https://doi.org/10.1016/j.jlap.2005.07.001`. URL: `https://www.sciencedirect.com/science/article/pii/S1567832605000536`.

[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.

[KG16] Hsiang-Shang Ko and Jeremy Gibbons. "Programming with ornaments". In: *Journal of Functional Programming* 27 (2016), e2. DOI: `10.1017/S0956796816000307`.

[Ko14] H Ko. "Analysis and synthesis of inductive families". PhD thesis. Oxford University, UK, 2014.

[Mag+10] José Pedro Magalhães et al. "A Generic Deriving Mechanism for Haskell". In: *SIGPLAN Not.* 45.11 (Sept. 2010), pp. 37–48. ISSN: 0362-1340. DOI: `10.1145/2088456.1863529`. URL: `https://doi.org/10.1145/2088456.1863529`.

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

[Nor09] Ulf Norell. “Dependently Typed Programming in Agda”. In: *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Ed. by Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 230–266. ISBN: 978-3-642-04652-0. DOI: `10.1007/978-3-642-04652-0_5`. URL: `https://doi.org/10.1007/978-3-642-04652-0_5`.

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

[Rey83] John C Reynolds. “Types, abstraction and parametric polymorphism”. In: *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*. 1983, pp. 513–523.

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

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

[The23] The Agda Community. *Agda Standard Library*. Version 1.7.2. Feb. 2023. URL: `https://github.com/agda/agda-stdlib`.

[VL14] Edsko de Vries and Andres Löh. “True Sums of Products”. In: *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*. WGP '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 83–94. ISBN: 9781450330428. DOI: `10.1145/2633628.2633634`. URL: `https://doi.org/10.1145/2633628.2633634`.

[Wad89] Philip Wadler. “Theorems for Free!” In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. Imperial College, London, United Kingdom: Association for Computing Machinery, 1989, pp. 347–359. ISBN: 0897913280. DOI: `10.1145/99370.99404`. URL: `https://doi.org/10.1145/99370.99404`.

[WKS22] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. Aug. 2022. URL: `https://plfa.inf.ed.ac.uk/22.08/`.

Part V
Appendix

When finished, shuffle the appendices to the order they appear in

A Index-first

B Without K but with universe hierarchies

See [EC22] and the small blurb rewriting interpretations as datatypes.

C Sigma descriptions

D ornForget and ornErase in full

E fold and mapFold in full