Generic Numerical Representations via Ornaments

Samuel Klumpers
6057314

November 8, 2023

Contents

Todo list

1 Introduction

There is a close relation between number systems and container objects or collections, which contain a certain number of elements. Examples of numerical representations, which are datastructures designed after a number system, are explored in Okasaki's Purely Functional Data Structures ([Oka98], chapter 9), reinterpreting some known datastructures as numerical representations.

To illustrate this, consider the binary numbers in their bijective, or zeroless, form (least significant digit first)

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

Here, `0b` corresponds to 0, `1b n` corresponds to `2n + 1`, representing the positive odd numbers, and `2b` corresponds to `2n + 2`, representing the positive even numbers. As a positional number system, `Bin` has digits 1 and 2, and counting from the left starting at 0, the weight of a digit at the `i`th position is 2^i. For example, the number 5 is represented by `1b 2b 0b`, since $1 \cdot 2^0 + 2 \cdot 2^1 + 0 \cdot 2^2 = 5$.

Compare this to the type of random-access lists (complete binary trees) in their nested (non-uniformly recursive) form ([Oka98], subsections 9.2.2 and 10.1.2)

```
data Random (A : Type) : Type where
  Zero :                         Random A
  One  : A     → Random (A × A) → Random A
  Two  : A → A → Random (A × A) → Random A
```

Note that `One` and `Two` take one and two values of `A` respectively, but in the recursive field we pass the type of pairs `A × A` as the parameter instead, hence the non-uniformity. One level deeper, `One` would ask for two values of `A`, and another level deeper for four, and so on.

Stripping the fields from a random-access list `xs` reveals a binary number `size xs` again For example, applying `size` to `One _ (Two _ _ Zero)` gives us back `1b 2b 0b`. We called this number `size` because it coincides with the number of elements in `xs`: evidently, the `size` and number of elements of `Zero` are both zero. On the other hand, suppose that `xs` of type `Random (A × A)` has size `n`. Since `A × A` contains two values of `A`, we have doubled the weight of `xs`, so that it actually contains `2n` values of `A`. Consequently, `One x xs` contains `2n + 1` values, and `Two x y xs` contains `2n + 2` values, so in general any `ys` contains `size ys` values.

In fact, if we remove the fields from random-access lists, binary numbers and random-access lists are essentially the same datatype. Conversely, we can describe random-access lists as binary numbers decorated with fields. Exactly such "informal human observations" can be made more precise and general using the language of ornaments as described by McBride [McB14]. This language effectively describes up to which modifications, such as adding or deleting fields, one datatype can be seen as a more elaborate version of another. In it, we can formulate random-access lists as an ornament on binary numbers, and get `size` for free as the forgetful function.

Datastructures with relations to number systems occur more commonly, which raises the questions of how we can make this relation explicit in more general cases, but also which number systems have associated numerical representations, and which numerical representations arise from ornaments.

In this thesis we will explore how, for a certain generalization of positional number systems, we can construct all numerical representations as ornaments, and how some known examples of numerical representations fit into this framework, making the following contributions:

1. We define a universe in which we will encode number systems and numerical representations. This universe allows annotations, non-uniform datatypes, and composite datatypes. By encoding those datatypes in the universe, we gain the ability to write generic programs over them.

2. Then, we adapt the language of ornaments to this universe, which lets us relate datatypes up to insertion of fields, nesting, and refinement of parameters, indices, and variables.

3. Finally, we prove the existence of two variants of numerical representations by demonstrating generic functions from number systems to ornaments, establishing that each number system has a numerical representation of the same structure.

As far as we are aware, this provides the first encoding that combines general (indexed) inductive types with non-uniform recursion. The incorporation of metadata also hopefully allows for the exploration of more constrained generic functions without requiring the programmer to redefine the universe they are working over. The accompanying definition of ornaments, while less powerful and theoretically justified than other definitions [Sij16; Ko14], may add a new dimension of flexibility to the space of ornaments by allowing ornaments to modify the nesting of datatypes.

Maybe note we're also effectively studying the effect of nesting, composite types and variable transforms on the theory of descriptions and ornaments.

We formalize our work using the dependently typed proof assistant Agda [Tea23], using the unsafe `--type-in-type` option so that the presented code is not diluted by the level variables, knowing that our universe (and primarily the telescopes) can be made consistent [EC22]. We also use `--with-K` (refer to Appendix C) and omit many type variables by using variable generalization.

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].

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]

[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.

```
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

```
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:

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

```
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
```

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

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

[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`.

```
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
  [] : 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 = ∅ ▷ (λ _ → 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
```
We will name maps `Δ → Γ` of telescopes `Cxf Δ Γ`. Given such a map `g`, name maps `W → V` between extensions `Vxf g W V`:
```
map-var : ∀ {A B C} → (∀ {a} → B a → C a) → Σ A B → Σ A C
map-var f (a , b) = (a , f b)

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
```

We also defined two functions we will use extensively later: `var→par` states that a map of extensions extend to a map of the whole telescope, and `Vxf-▷` lets us extend a map of extensions by acting as the identity on a new variable.

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. The interpretation of the universe is then:

```
⟦_⟧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 (∅ ▷ λ _ → 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 (∅ ▷ (λ _ → 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

```
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 (λ _ → ℕ)
     ( 𝟙 (λ { (_ , (_ , n)) → suc n } ))
     ∷ σ (λ _ → ℕ)
     ( ρ (λ { (_ , (_ , n)) → n } )
     ( 𝟙 (λ { (_ , (_ , n)) → suc n } )))
     ∷ []
```

and

```
VecD : U-ix (∅ ▷ λ _ → Type) ℕ
VecD = 𝟙 (λ _ → zero)
     ∷ σ (λ _ → 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 (∅ ▷ λ _ → 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 F, 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 (λ _ → refl)
           ∷ Δσ {S = λ { ((_ , A), _) → A }}
           ( ρ (λ _ → refl)
           ( 𝟙 (λ _ → 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 (λ _ → refl)
           ∷ Δσ {S = λ _ → ℕ}
           ( σ
           ( ρ {j = λ { (_ , (_ , n) , _) → n }}     (λ _ → refl)
           ( 𝟙 {j = λ { (_ , (_ , n) , _) → suc n }} (λ _ → 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`, and some later arguments we provide to ornaments, are implicit argument: Agda would happily infer them from `ListD` and later `VecD` had we omitted them.

```
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].

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`.

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[8]. 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}
```

[8] Consequently, `OrnDesc Δ J g i D` must simply be a convenient representation of `Σ (U-ix Δ J) (Orn g i D)`.

```
         → 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`:

```
NatOD : OrnDesc (∅ ▷ λ _ → Type) τ ! ! 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

Before we can analyse and describe number systems and their numerical representations using generic programs, we first have to ensure that these types fit into the descriptions. In this section we discuss how some numerical representations are hard to describe using only the descriptions of parametric indexed inductive types `U-ix`, and based on this discussion we present an extension of `U-ix` incorporating metadata, parameter transformation, description composition, and variable transformation.

8 Numerical Representations

Before we start rebuilding our universe, let us look at the construction of the simplest numerical representation `ℕ`, `List` and `Vec`. At first, we defined `Vec` as the length-indexed variant of `List`, such that `lookup` becomes total, and satisfies nice properties like `lookup-insert`. Abstractly, `Vec` is an implementation of finite maps with domain `Fin`, where finite maps are simply those types with operations like `insert`, `remove`, `lookup`, and `tabulate`[9], satisfying relations or laws like `lookup-insert` and `lookup ∘ tabulate ≡ id`.

For comparison, we can define a trivial implementation of finite maps, by reading `lookup` as a prescript

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

Since `lookup` is simply the identity function on `Lookup`, this unsurprisingly satisfies the laws of finite maps, provided we define `insert` and `remove` correctly.

Predictably[10], `Vec` is *representable*, that is, we have that `Lookup` and `Vec` are equivalent, in the sense that there is an isomorphism between `Lookup` and `Vec`[11]

```
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[12]. In terms of

[9]The function `tabulate : (Fin n → A) → Vec A n` collects an assignment of elements `f` into a vector `tabulate f`.

[10]Since `lookup` is an isomorphism with `tabulate` as inverse, as we see from the relations `lookup ∘ tabulate ≡ id` and `tabulate ∘ lookup ≡ id`.

[11]Without further assumptions, we cannot use the equality type `≡` for this notion of equivalence of types: a type with a different name but exactly the same constructors as `Vec` would not be equal to `Vec`.

[12]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 $f(a) = b$.

elements, this means that elements of `A` and `B` are in one-to-one correspondence.

We can also establish properties like `lookup-insert` from this equivalence, rather than deriving it ourselves. 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`. Turning the `Iso` on its head, and 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[13]

```
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
```

[13]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 can be obtained by using the cubical path types.

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].

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.

In conclusion, we computed a type of finite maps (the numerical representation `Vec`) from a number system (`ℕ`), by cases on the number system and making use of the values represented by the number system.

9 Room for Improvement

We could now carry on and attempt to generalize this calculation to more number systems, but we would quickly run into dead ends for certain numerical representations. Let us give an overview of what bits of `U-ix` are still missing if we are going to generically construct all numerical representations we promised.

9.1 Number systems

In the calculation `Vec` from `ℕ`, we analyse and replicate the structure of `ℕ`, deliberately choosing to add 0 fields in the case corresponding to `zero` and 1 field in the case of `one`, knowing the meaning of these constructors in terms of numerical value from the explanation of `ℕ` in words[14].

However, if we want to compute numerical representations generically, we also have to convince the computer that our datatypes indeed represent number systems. As a first step, let us fix `ℕ` as the primordial number system, so that we can compare other number systems by how they are mapped into `ℕ`. For example, `ℕ` is trivially interpreted by `id : ℕ → ℕ`. The binary numbers as described in the introduction can be mapped to `ℕ` by

```
toℕ-Bin : Bin → ℕ
toℕ-Bin 0b = 0
toℕ-Bin (1b n) = 1 + 2 * toℕ-Bin n
toℕ-Bin (2b n) = 2 + 2 * toℕ-Bin n
```

As a more exotic example, we have a number system

```
data Carpal : Type where
  0c : Carpal
  1c : Carpal
  2c : Phalanx → Carpal → Phalanx → Carpal

toℕ-Carpal : Carpal → ℕ
toℕ-Carpal 0c = 0
toℕ-Carpal 1c = 1
toℕ-Carpal (2c l m r) = toℕ-Phalanx l + 2 * toℕ-Carpal m + toℕ-Phalanx r
```

which is composed of smaller "number systems"

```
data Phalanx : Type where
  1p 2p 3p : Phalanx

toℕ-Phalanx : Phalanx → ℕ
toℕ-Phalanx 1p = 1
toℕ-Phalanx 2p = 2
toℕ-Phalanx 3p = 3
```

We could now define a general number system as a type `N` equipped with a map `N → ℕ`, but this would both be too general for our purpose and opaque to generic programs. On the other hand, allowing only traditional positional number systems excludes number systems like `Carpal`, which would otherwise still have valid numerical representations.

Instead, we observe that across the above examples, the interpretation of a number is computed by simple recursion. In particular leaves have associated constants, recursive fields correspond to multiplication and addition, while fields can defer to another function. If we describe the types in `U-sop`, we can thus encode each of these systems by associating a single number to each `𝟙` and `ρ`, and a function to each `σ`, up to equivalence. In essence, this encodes number systems as structures that at each node linearly combine values of subnodes, generalizing positional number systems in *dense* representation[15].

---

[14]More accurately, the meaning of `ℕ` comes from `Fin`, which gets its meaning from our definition of `_<_`.

[15]This excludes some number systems, as we discuss in Section 21.

Using a modified version of `U-sop`, we can encode the examples we gave as follows. Note that in `ℕ`, we have to insert fields of `τ`, so we can express that the second constructor acts as $x \mapsto x + 1$

```
Nat-num : U-num
Nat-num = 𝟙 0
        ∷ σ τ (λ _ → 1)
        ( ρ 1
        ( 𝟙 0 ))
        ∷ []
```

marking all leaves as zero. The binary numbers admit a similar encoding, but multiply their recursive fields by two instead

```
Bin-num : U-num
Bin-num = 𝟙 0
        ∷ σ τ (λ _ → 1)
        ( ρ 2
        ( 𝟙 0 ))
        ∷ σ τ (λ _ → 2)
        ( ρ 2
        ( 𝟙 0 ))
        ∷ []
```

The `Carpal` system can be encoded by using the interpretation of `Phalanx`

```
Carpal-num : U-num
Carpal-num = 𝟙 0
           ∷ 𝟙 1
           ∷ σ Phalanx toℕ-Phalanx
           ( ρ 2
           ( σ Phalanx toℕ-Phalanx
           ( 𝟙 0 )))
           ∷ []
```

9.2 Nested types

If our construction is going to cast `Random`, as defined in Section 1, as the numerical representation associated to `Bin`, then `Random` needs to be describable to begin with. The recursive fields of `Random` have parameters `A × A` rather than `A`, making `Random` a nested type, as opposed to a uniformly recursive type in which the parameters of the recursive fields are identical to the top-level parameters. Consequently, `Random` has no adequate description in `U-ix`[16].

Due to the work of Johann and Ghani [JG07], we can model general nested types as fixpoints of higher-order functors (i.e., endofunctors on the category of endofunctors)

```
Fun  = Type → Type
HFun = Fun → Fun
```

[16] Here, the "inadequate" descriptions either hardly resemble the user defined `Random`, use indices to store the depth of a node (see Appendix A), or only have a complicated isomorphism to `Random`.

```
{-# NO_POSITIVITY_CHECK #-}
data HMu (H : HFun) (A : Type) : Type where
  con : H (HMu H) A → HMu H A
```

By placing the recursive field `Mu F` under `F`, the functor `F` can modify `Mu F` and `A` to determine the type of the recursive field. `Random` can then be encoded via `Mu` as

```
data HRandom (F : Fun) (A : Type) : Type where
  Zero :                      HRandom F A
  One : A     → F (A × A) → HRandom F A
  Two : A → A → F (A × A) → HRandom F A
```

However, this definition is unsafe (as you might have been able to tell from the pragma disabling the positivity checker), i.e., `Mu` is easily used to derive `⊥` by passing a negative functor for `F`.

Instead, we settle for the weaker, but safe, inner nesting. This kind of nesting can be described by a simple modification to the recursive field `ρ` in `U-ix`

```
ρ : V ⊢ I → Cxf Γ Γ → Con-nest Γ V I → Con-nest Γ V I
```

allowing a recursive field specify a transformation `Cxf` that is applied to the parameters before they are passed to the recursive field. Correspondingly, the interpretation of `ρ` applies `f` before passing `p` to the recursive field `X`

```
⟦ ρ j g C ⟧C-nest X pv@(p , v) i = X (g p) (j pv) × ⟦ C ⟧C-nest X pv i
```

With this modification, `Random` can be transcribed literally

```
RandomD : U-nest (∅ ▷ λ _ → Type) ⊤
RandomD = 𝟙 _
        ∷ σ (λ { ((_ , A) , _) → A })
        ( ρ _ (λ { (_ , A) → (_ , A × A) })
        ( 𝟙 _ ))
        ∷ σ (λ { ((_ , A) , _) → A })
        ( σ (λ { ((_ , A) , _) → A })
        ( ρ _ (λ { (_ , A) → (_ , A × A) })
        ( 𝟙 _ )))
        ∷ []
```

using the map $A \mapsto A \times A$ to describe its nesting like usual.

To avoid the inconvenience caused by `ρ` for uniformly recursive types, we define a shorthand emulating the old behaviour of `ρ`.

9.3 Composite types

In Subsection 9.1, we defined the number system `Carpal-num` as a composite type using `Phalanx`. By the same argument as there, the description `Carpal-num` which relies on `toℕ-Phalanx` to describe the value of `Phalanx`, turns out to be too imprecise to recover the full numerical representation generically. More generally, a generic function may inspect the outer structure of a composite type to construct the outer part of the numerical representation, but it would not see the structure of the other number systems inside.

Inlining the constructors of `Phalanx` into `Carpal` would allow generic constructions to see the structure of `Phalanx`, but is undesirable here and in general, as this yields a type with two of the original constructors of `Carpal`, and 9 more constructors for each combination of constructors of `Phalanx`[17].

Instead, we opt to add a new former to the universe, specialized to fields of known descriptions

```
δ : (R : U-comp Δ J) (d : Cxf Γ Δ) (j : I → J)
  → Con-comp Γ V I → Con-comp Γ V I
```

taking the functions `d` and `j` to determine the parameters and indices passed to `R`. A field encoded by `δ` is then interpreted identically to how it would be if we used `σ` and `μ` instead[18]:

```
⟦ δ R d j C ⟧C-comp X pv@(p , v) i
                        = μ-comp R (d p) (j i) × ⟦ C ⟧C-comp X pv i
```

Using `δ` rather than `σ` allows us to reveal the description of a field to a generic program. Rather than adding `Phalanx` via a `σ`, we would use `δ` to directly add `Phalanx-num` instead.

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).

9.4 Hiding variables

With the modifications described above, we can describe all the structures we want. However, there is one peculiarity in the way `U-ix` handles variables, namely, each field added by a `σ` is treated as bound. Even if the value is then unused, all fields after the `σ` need to work around it. While only a minor inconvenience, this means that two subsequent fields which refer to the same variable will have to be encoded differently. Furthermore, adding fields of complicated types can quickly clutter the context when writing or inspecting a generic program.

Using a simple modification to how telescopes are used in `U-ix`, we can emulate both bound and unbound fields without adding more formers to `U-ix`. By accepting a transformation of variables `Vxf Γ (V ▷ S) W` after a `σ S` in the context of `V`, the remainder of the fields can be described in the context `W`:

```
σ : (S : V ⊢ Type) → Vxf id (V ▷ S) W → Con-var Γ W I → Con-var Γ V I
```

Of course, it would be no use to redefine `σ` in order to save the user some work, and instead leave the with the burden of manually adding these transformations, so we define shorthands emulating precisely the bound field

```
σ+ : ∀ {V} → (S : V ⊢ Type) → Con-var Γ (V ▷ S) I → Con-var Γ V I
σ+ S C = σ S id C
```

and the unbound field

```
σ- : ∀ {V} → (S : V ⊢ Type) → Con-var Γ V I → Con-var Γ V I
σ- S C = σ S fst C
```

[17] If working with 11 constructors sounds too feasible, consider that defining addition on types like `Carpal` (or concatenation its numerical representation) is not (yet) generic and, if fully written out, will instead demand 121 manually written cases.

[18] The omission of `μ R` is intentional, while workable, the construction of ornaments becomes significantly more complicated.

10 A new Universe

Now, we will define a new universe based on `U-ix`, incorporating all modifications we described above. This universe is again the type of lists of constructors

```
data DescI (Me : Meta) (Γ : Tel ⊤) (I : Type) : Type where
  []  : DescI Me Γ I
  _∷_ : ConI Me Γ ∅ I → DescI Me Γ I → DescI Me Γ I
```

Compared to `U-ix`, `DescI` is also parametrized over the metadata `Meta`, which we will use later to encode number systems in `DescI`.

The constructors for this universe are defined as follows

```
data ConI (Me : Meta) (Γ : Tel ⊤) (V : ExTel Γ) (I : Type) : Type where
  𝟙 : {me : Me .𝟙i} (i : Γ & V ⊢ I) → ConI Me Γ V I

  ρ : {me : Me .ρi}
      (g : Cxf Γ Γ) (i : Γ & V ⊢ I) (C : ConI Me Γ V I)
    → ConI Me Γ V I

  σ : (S : V ⊢ Type) {me : Me .σi S}
      (w : Vxf id (V ▷ S) W) (C : ConI Me Γ W I)
    → ConI Me Γ V I

  δ : {me : Me .δi Δ J} {iff : MetaF Me′ Me}
      (d : Γ & V ⊢ ⟦ Δ ⟧tel tt) (j : Γ & V ⊢ J)
      (R : DescI Me′ Δ J) (C : ConI Me Γ V I)
    → ConI Me Γ V I
```

Remark that `𝟙` remains the same, but `ρ` can now accept the transformation `Cxf Γ Γ` to encode non-uniform parameters. Likewise, `σ` now also takes the transformation `w` from `V ▷ S` to `W` allowing us to replace the context after a field with `W` rather than `V ▷ S`. Finally, `δ` is added to directly describe composite datatypes by giving a description `R` to represent a field `μ R`.

Let us take a fresh look at some datatypes from before, now through the lens of `DescI`. We will leave the metadata aside for now by using

```
Con = ConI Plain
Desc = DescI Plain
```

Like before, we use the shorthands `σ+`, `σ-`, and `ρ0` to keep descriptions which do not make use of the new features concise.

We can describe `ℕ` and `List` as before

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

ListD : Desc (∅ ▷ λ _ → Type) ⊤
ListD = 𝟙 _
  ∷ σ- (λ ((_ , A) , _) → A)
  ( ρ0 _ (𝟙 _))
  ∷ []
```

replacing σ with σ- and ρ with ρ0.

On the other hand, if we define Vec, we bind the length as a (implicit) field, so we use σ+ instead

```
VecD : Desc (∅ ▷ λ _ → Type) ℕ
VecD = 𝟙 (λ _ → 0)
     ∷ σ- (λ ((_ , A) , _) → A)
     ( σ+ (λ _ → ℕ)
     ( ρ0 (λ (_ , (_ , n)) → n)
     ( 𝟙  (λ (_ , (_ , n)) → suc n))))
     ∷ []
```

and extract the length n like we would in U-ix.

With the nested recursive field ρ, we can almost repeat the definition of Random from U-nest:

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

Binary fingertrees (as a simplification of 2-3 fingertrees [HP06]), a nested datatype like Random, instead storing elements in variably sized digits on both sides, can be composed from digits

```
DigitD : Desc (∅ ▷ λ _ → 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 _)))
       ∷ []
```

using δ to add fields represented by DigitD

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

```
            ∷ []
```

These descriptions can be instantiated as before by taking the fixpoint

```
data μ (D : DescI Me Γ I) (p : ⟦ Γ ⟧tel tt) : I → Type where
  con : ∀ {i} → ⟦ D ⟧D (μ D) p i → μ D p i
```

of their interpretations as functors

```
⟦_⟧C : ConI Me Γ V I → ( ⟦ Γ ⟧tel tt → I → Type)
                     → ⟦ Γ & V ⟧tel  → I → Type
⟦ 𝟙 i′      ⟧C X pv         i = i ≡ i′ pv
⟦ ρ g i′ D  ⟧C X pv@(p , v) i = X (g p) (i′ pv) × ⟦ D ⟧C X pv i
⟦ σ S w D   ⟧C X pv@(p , v) i = Σ[ s ∈ S pv ] ⟦ D ⟧C X (p , w (v , s)) i
⟦ δ d j R D ⟧C X pv         i = Σ[ s ∈ μ R (d pv) (j pv) ] ⟦ D ⟧C X pv i

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

inserting the transformations of parameters `g` in `ρ` and the transformations of variables `w` in `σ`.

Like `U-ix`, `DescI` comes with a generic `fold`

```
fold : ∀ {D : DescI Me Γ I} {X} → ⟦ D ⟧D X ⇉ X → μ D ⇉ X
```

which is defined analogously.

10.1 Annotating Descriptions with Metadata

We promised encodings of number systems in `DescI`, so let us show how number systems are an instance of `Meta` and how this lets use `DescI` like we used `U-num` to describe type and numerical value in one go.

By generalizing `DescI` over `Meta`, rather than coding the specification of number systems into the universe directly, we give ourselves the flexibility to both represent plain datatypes and number systems in the same universe. `DescI` then uses the specific type of `Meta` to query bits of information in the implicit fields in each of the type-formers. A term of `Meta` simply lists the type of information to be queried at each type former:

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

When a `δ` includes another description, the metadata on that description is a priori unrelated to the top-level metadata. When this happens, we ask that both sides is related by a transformation:

```
record MetaF (L R : Meta) : 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 metadata. This allows the inclusion of an annotated type `DescI Me` into an ordinary datatype `Desc` without duplicating the former definition in `Desc` first.

The encoding of number systems by associating numbers to `𝟙` and `ρ`, and functions to `σ`, can be summarized as

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

The `δ`-former, which was not described when we discussed encoding number systems in `U-num`, is assigned a single number, representing multiplication analogous to `ρ`. The equalities in the metadata of a `δ` ensure that number systems have no parameters or indices.

Using `Number`, we can for example reproduce the binary numbers `Bin-num` in `DescI` as

```
BinND : DescI Number ∅ ⊤
BinND = 𝟙 {me = 0} _
      ∷ ρ0 {me = 2} _ (𝟙 {me = 1} _)
      ∷ ρ0 {me = 2} _ (𝟙 {me = 2} _)
      ∷ []
```

Functions between metadata come in when we represent `Carpal-num` in its more accurate form by first defining

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

and directly including it into `Carpal`

```
CarpalND : DescI Number ∅ ⊤
CarpalND = 𝟙 {me = 0} _
         ∷ 𝟙 {me = 1} _
         ∷ δ {me = refl , refl , 1} {id-MetaF} _ _ PhalanxND
         ( ρ0 {me = 2} _
         ( δ {me = refl , refl , 1} {id-MetaF} _ _ PhalanxND
         ( 𝟙 {me = 0} _)))
         ∷ []
```

where we can use the identity function as both sides exactly use `Number`.

The metadata on a `DescI Number` can then be used to define a generic function sending terms of number systems to their `value` in `ℕ`

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

which is defined by generalizing over the inner metadata and `fold`ing using

```
value-desc : (D : DescI Me Γ ⊤) → ∀ {a b} → ⟦ D ⟧D (λ _ _ → ℕ) a b → ℕ
value-con : (C : ConI Me Γ V ⊤) → ∀ {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 {me = k} j) refl
    = ϕ .𝟙f k

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

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

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

On the other hand, we can also declare that a description has no metadata at all by querying `⊤` for all type-formers:

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

By making the fields querying information implicit in the type of descriptions, we can ensure that descriptions from `U-ix` can be imported into `Desc` without having to insert metadata anywhere.

But it is also possible to use `Meta` to encode conventionally useful metadata such as field names

```
Names : Meta
Names .𝟙i = ⊤
Names .ρi = String
Names .σi _ = String
Names .δi _ _ = String
```

Part II
Ornaments

In the framework of `DescI` of 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.

11 Ornamental descriptions

The ornamental descriptions for `DescI` take the same shape as those in Section 7, generalized to handle nested types, variable transformations, and composite types. These ornamental descriptions are defined such that a `OrnDesc Me′ Δ re-par J re-index D` represents a patch from a base description `D` to a description with metadata `Me′`, parameters `Δ` and indices `J`. Note that metadata, as a non-structural property, no direct influence on ornaments, so we simply generalize over the information on `D`, and query the information for the new description without imposing constraints.

Ornamental descriptions themselves are again lists of constructor ornaments

```
data OrnDesc {Me} (Me′ : Meta) (Δ : Tel τ)
             (re-par : Cxf Δ Γ) (J : Type) (re-index : J → I)
             : DescI Me Γ I → Type where
  []  : OrnDesc Me′ Δ re-par J re-index []
  _::_ : ConOrnDesc Me′ {re-par = re-par} ! re-index {Me = Me} CD
      → OrnDesc Me′ Δ re-par J re-index D
      → OrnDesc Me′ Δ re-par J re-index (CD :: D)
```

The constructor ornaments are also where we pay the price for the flexibility we built into `ConI`. For example, as `ConI` allows us to transform variables, `ConOrnDesc` has to relate the transformations on both sides to guarantee the existence of `ornForget`. A lot of lines are dedicated to the commutativity squares for variables, but these squares involving `Vxf` can generally ignored, as witnessed by the `Oσ+` and `Oσ-` variants of the `σ` ornament, automatically filling those squares in the usual cases of binding or ignoring fields.

The structure-preserving ornaments are defined as usual

```
data ConOrnDesc (Me′ : Meta) {re-par : Cxf Δ Γ}
                (re-var : Vxf re-par W V) (re-index : J → I)
                : ConI Me Γ V I → Type where
  𝟙 : {i : Γ & V ⊢ I} (j : Δ & W ⊢ J)
    → re-index ∘ j ~ i ∘ var→par re-var
    → {me : Me .𝟙i} {me′ : Me′ .𝟙i}
    → ConOrnDesc Me′ re-var re-index (𝟙 {Me} {me = me} i)

  ρ : {g : Cxf Γ Γ} (d : Cxf Δ Δ)
      {i : Γ & V ⊢ I} (j : Δ & W ⊢ J)
    → g ∘ re-par ~ re-par ∘ d
    → re-index ∘ j ~ i ∘ var→par re-var
    → {me : Me .ρi} {me′ : Me′ .ρi}
    → ConOrnDesc Me′ re-var re-index CD
    → ConOrnDesc Me′ re-var re-index (ρ {Me} {me = me} g i CD)

  σ : (S : Γ & V ⊢ Type)
```

```
    {g : Vxf id (V ▷ S) V′} (h : Vxf id (W ▷ (S ∘ var→par re-var)) W′)
    (v′ : Vxf re-par W′ V′)
  → (∀ {p} → g ∘ Vxf-▷ re-var S ~ v′ {p = p} ∘ h)
  → {me : Me .σi S} {me′ : Me′ .σi (S ∘ var→par re-var)}
  → ConOrnDesc Me′ v′ re-index CD
  → ConOrnDesc Me′ re-var re-index (σ {Me} S {me = me} g CD)

δ : (R : DescI If″ Θ K) (t : Γ & V ⊢ ⟦ Θ ⟧tel tt) (j : Γ & V ⊢ K)
  → {me : Me .δi Θ K} {iff : MetaF If″ Me}
    {me′ : Me′ .δi Θ K} {iff′ : MetaF If″ Me′}
  → ConOrnDesc Me′ re-var re-index CD
  → ConOrnDesc Me′ re-var re-index (δ {Me} {me = me} {iff = iff} t j R CD)
```

where ρ has a new field relating the old and new nesting transforms `g` and `d`. Likewise, σ now has a field relating the old and new variable transforms, which for example prevents us from unbinding a field in the new description which was used in the old description. The ornament δ now represents the direct copying of a δ in descriptions up to `re-par` and `re-var`.

Where only Δσ could add fields before, we can now also add fields described by δ using Δδ

```
Δσ : (S : Δ & W ⊢ Type) (h : Vxf id (W ▷ S) W′)
     (v′ : Vxf re-par W′ V)
   → (∀ {p} → re-var ∘ fst ~ v′ {p = p} ∘ h)
   → {me′ : Me′ .σi S}
   → ConOrnDesc Me′ v′ re-index CD
   → ConOrnDesc Me′ re-var re-index CD

Δδ : (R : DescI If″ Θ J) (t : W ⊢ ⟦ Θ ⟧tel tt) (j : W ⊢ J)
   → {me′ : Me′ .δi Θ J} {iff′ : MetaF If″ Me′}
   → ConOrnDesc Me′ re-var re-index CD
   → ConOrnDesc Me′ re-var re-index CD
```

Again, Δσ now requires the relation of old and new variables.

The last ornament represents an ornament *inside* a δ: If we have a description `D'` $=$ `δ R d j R D` referencing a description `R`, then we may expect that an ornamental description on top of `R` also induces an ornamental description on top of `D'`. We generalize this by defining a kind of orthogonal composition of ornaments[19], taking ornamental descriptions `RR′` on `R` and `DD′` on `D`, and producing an ornamental description on `D'`:

```
∙δ : {R : DescI If″ Θ K} {c′ : Cxf Λ Θ} {fΘ : V ⊢ ⟦ Θ ⟧tel tt}
     (fΛ : W ⊢ ⟦ Λ ⟧tel tt) {k′ : M → K} {k : V ⊢ K}
     (m : W ⊢ M)
   → (RR′ : OrnDesc If‴ Λ c′ M k′ R)
   → (p₁ : ∀ q w → c′ (fΛ (q , w)) ≡ fΘ (re-par q , re-var w))
   → (p₂ : ∀ q w → k′ (m (q , w)) ≡ k (re-par q , re-var w))
   → ∀ {me} {iff} {me′ : Me′ .δi Λ M} {iff′ : MetaF If‴ Me′}
```

[19] As opposed to Ko's parallel composition [**ko**].

```
        → (DE : ConOrnDesc Me′ re-var re-index CD)
        → ConOrnDesc Me′ re-var re-index (δ {Me} {me = me} {iff = iff} fΘ k R CD)
```

Roughly speaking, the equality `p₁`, respectively `p₂`, demands that the parameter, respectively index, passed to `R` as computed before and after transforming the outer parameters and variables agree.

As before we can define `ornForget` by erasing ornaments, now using the new commutativity squares. The precise meaning of ornamental descriptions as descriptions is given by the conversion:

```
toDesc : {re-var : Cxf Δ Γ} {re-index : J → I} {D : DescI Me Γ I}
       → OrnDesc Me′ Δ re-var J re-index D → DescI Me′ Δ J
toDesc [] = []
toDesc (CO ∷ O) = toCon CO ∷ toDesc O

toCon  : {re-par : Cxf Δ Γ} {re-var : Vxf re-par W V} {re-index : J → I} {D : ConI Me Γ V I}
       → ConOrnDesc Me′ re-var re-index D → ConI Me′ Δ W J
toCon (𝟙 j x {me′ = me})
  = 𝟙 {me = me} j

toCon (ρ j h x x₁ {me′ = me} CO)
  = ρ {me = me} j h (toCon CO)

toCon {re-var = v} (σ S h v′ x {me′ = me} CO)
  = σ (S ∘ var→par v) {me = me} h (toCon CO)

toCon {re-var = v} (δ R j t {me′ = me} {iff′ = iff} CO)
  = δ {me = me} {iff = iff} (j ∘ var→par v) (t ∘ var→par v) R (toCon CO)

toCon (Δσ S h v′ x {me′ = me} CO)
  = σ S {me = me} h (toCon CO)

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

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

which makes use of the implicit metadata fields in the constructor ornaments to reconstruct the metadata on the target description.

With `OrnDesc` we can reproduce the examples of the ornamental descriptions for `U-ix`, but also present some previously inexpressible types as ornamental descriptions. Using the variants of some ornaments specialized to binding or ignoring fields:

```
Oσ+ : (S : Γ & V ⊢ Type) {CD : ConI Me Γ V′ I} {h : Vxf _ _ _}
  → {me : Me .σi S} {me′ : Me′ .σi (S ∘ var→par re-var)}
  → ConOrnDesc Me′ (h ∘ Vxf-▷ re-var S) re-index CD
  → ConOrnDesc Me′ re-var re-index (σ {Me} S {me = me} h CD)
Oσ+ S {h = h} {me′ = me′} CO
  = σ S id (h ∘ Vxf-▷ re-var S) (λ _ → refl) {me′ = me′} CO
```

```
Oσ- : (S : Γ & V ⊢ Type) {CD : ConI Me Γ V I}
  → {me : Me .σi S} {me′ : Me′ .σi (S ∘ var→par re-var)}
  → ConOrnDesc Me′ re-var re-index CD
  → ConOrnDesc Me′ re-var re-index (σ {Me} S {me = me} fst CD)
Oσ- S {me′ = me′} CO = σ S fst re-var (λ _ → refl) {me′ = me′} CO
```

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

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

Rather than defining `Random` in a vacuum, we can use the new flexibility in `ρ` and describe random access lists as an ornament from binary numbers:

```
RandomOD : OrnDesc Plain (∅ ▷ λ _ → Type) ! τ id BinND
RandomOD = 𝟙 _ (λ _ → refl)
         :: OΔσ- (λ ((_ , A) , _) → A)
         ( ρ (λ (_ , A) → (_ , Pair A)) _ (λ _ → refl) (λ _ → refl)
         ( 𝟙 _ (λ _ → refl)))
         :: OΔσ- (λ ((_ , A) , _) → A)
         ( OΔσ- (λ ((_ , A) , _) → A)
         ( ρ (λ (_ , A) → (_ , Pair A)) _ (λ _ → refl) (λ _ → refl)
         ( 𝟙 _ (λ _ → refl))))
         :: []
```

Likewise, we can give an ornament turning phalanges into digits

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

and assemble these into fingertrees with `δ•`

```
FingerOD : OrnDesc Plain (∅ ▷ λ _ → Type) ! τ id CarpalND
FingerOD = 𝟙 _ (λ _ → refl)
         :: OΔσ- (λ ((_ , A) , _) → A)
         ( 𝟙 _ (λ _ → refl))
         :: •δ (λ (p , _) → p) _ DigitOD (λ _ _ → refl) (λ _ _ → refl)
         ( ρ (λ (_ , A) → (_ , Pair A)) _ (λ _ → refl) (λ _ → refl)
         ( •δ (λ (p , _) → p) _ DigitOD (λ _ _ → refl) (λ _ _ → refl)
```

```
( 𝟙 _ (λ _ → refl))))
∷ []
```

Part III
Generic 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, using `DescI Number` to represent number systems, we 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 Unindexed Numerical Representations

In this section, we will demonstrate how we can use the ornamental descriptions to generically compute numerical representations. More precisely, we will define `TreeOD`, which sends a number system to the corresponding type of full (nested) node trees over it.

might need to find better names for TreeOD

Is full nested node trees accurate?

We proceed differently from the calculation of `Vec` from `ℕ`. Indeed, we will give ornamental descriptions, rather than deriving a direct definition step-by-step through 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. In the next section, we will demonstrate the indexed numerical representations as an ornament on top of the unindexed variant.

To ornament a number system to its unindexed numerical representation, we recall the interpretation `value` of number systems into `ℕ`. Let us consider how each of the cases of `ConI Number` should be ornamented in order to actually give a numerical representation. Consider what happens at a leaf of value `k` in a number system

```
𝟙-case : ℕ → ConI Number ∅ V ⊤
𝟙-case k = 𝟙 {me = k} _
```

Let us refer to the sole parameter of a numerical representation as `A`. Since the `value` contributed by this leaf is constantly `k`, a numerical representation should accordingly have k fields of A before this leaf, or equivalently a field containing k values of A. A recursive field of weight k

```
ρ-case : ℕ → ConI Number ∅ V τ → ConI Number ∅ V τ
ρ-case k C = ρ0 {me = k} _ C
```

multiplies the value contributed by the recursive part by k. Hence, the numerical representation should have a recursive field, in such a way that each "A" in the recursive field actually contains k values of A. On the other hand, an ordinary field, sending its values to ℕ by a mapping f

```
σ-case : (S : V ⊢ Type) → (∀ p → S p → ℕ) → ConI Number ∅ V τ → ConI Number ∅ V τ
σ-case S f C = σ- S {me = f} C
```

is simply represented in the numerical representation by adding a field with k values of A. Finally, a field containing another number system R with weight k

```
δ-case : ℕ → DescI Number ∅ τ → ConI Number ∅ V τ → ConI Number ∅ V τ
δ-case k R C = δ {me = refl , refl , k} {id-MetaF} _ _ R C
```

directly contributes values of R multiplied by k. The outer numerical representation should then replace R with its numerical representation NR, of which each value should represent k values of A, analogous to the recursive field.

To describe the numerical representation, we encode these fields of weight k with k-element vectors, and in the same way, the multiplication by k in the cases of ρ and δ is modelled by nesting over a k-element vector. Combining all these cases and translating them to the language of ornaments we define the unindexed numerical representation:

```
TreeOD : (D : DescI Number ∅ τ) → OrnDesc Plain (∅ ▷ λ _ → Type) ! τ ! D
TreeOD D = Tree-desc D id-MetaF
  module TreeOD where
  Tree-desc : (D : DescI Me ∅ τ) → MetaF Me Number
            → OrnDesc Plain (∅ ▷ λ _ → Type) ! τ ! D

  Tree-con  : {re-var : Vxf ! W V} (C : ConI Me ∅ V τ) → MetaF Me Number
            → ConOrnDesc {Δ = ∅ ▷ λ _ → Type} {W = W} {J = τ} Plain re-var ! C

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

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

  Tree-con (ρ {me = k} _ _ C) ϕ
    = ρ (λ (_ , A) → (_ , Vec A (ϕ .ρf k))) _ (λ _ → refl) (λ _ → refl)
    ( Tree-con C ϕ)

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

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

In most cases, we straightforwardly use `OΔσ-` to insert vectors of the correct size. However, in the case of `ρ`, we can trivially change the nesting function to take the parameter `A` and give `Vec A k` as a parameter to the recursive field instead. In the case of `δ`, we similarly place the parameters in a vector, but these are now directed to the recursively computed numerical representation of `R`. This case is also why we generalize the whole construction over `ϕ : MetaF Me Number`, as `R` is allowed to have a `Meta` that is not `Number`, as long as it is convertible to `Number`. Consequently, everywhere we use the "weight" represented by `k` in the construction, we first apply `ϕ` to compute the actual weights and values from `Me`.

As an example, let us take a look at how `TreeOD` transforms `CarpalND` to its numerical representation, `FingerOD`. Applying `TreeOD` sends leaves with a value of `k` to `Vec A k`, so applying it to `PhalanxND` yields

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

which is equivalent to the `DigitOD` from before, expanding a vector of `k` elements into `k` fields. The same happens for the first two constructors of `CarpalND`, replacing them with an empty vector and a one-element vector respectively. The last constructor is more interesting

```
FingerOD : OrnDesc Plain (∅ ▷ λ _ → Type) ! τ id CarpalND
FingerOD = OΔσ- (λ ((_ , A) , _) → Vec A 0)
           ( 𝟙 _ (λ _ → refl))
           ∷ OΔσ- (λ ((_ , A) , _) → Vec A 1)
           ( 𝟙 _ (λ _ → refl))
           ∷ •δ (λ ((_ , p) , _) → (_ , Vec p 1)) ! DigitOD (λ _ _ → refl) (λ _ _ → refl)
           ( ρ (λ (_ , A) → _ , Vec A 2) _ (λ _ → refl) (λ _ → refl)
           ( •δ (λ ((_ , p) , _) → (_ , Vec p 1)) ! DigitOD (λ _ _ → refl) (λ _ _ → refl)
           ( OΔσ- (λ ((_ , A) , _) → Vec A 0)
           ( 𝟙 _ (λ _ → refl)) )))
           ∷ []
```

The `PhalanxND` in the last constructor gets replaced with `DigitOD` via `O•δ+`, and the recursive field gets replaced by a recursive field nesting over vectors of length. Again, this is equivalent to `FingerOD`, wrapping values in length one vectors and inserting empty vectors.

13 Indexed Numerical Representations

Like how `List` has an ornament `VecOD` to its `ℕ`-indexed variant `Vec`, we can also construct an ornament, which we will call `TrieOD D`, from the numerical representation `TreeOD D` to its `D`-indexed variant:

```
TrieOD : (N : DescI Number ∅ τ)
         → OrnDesc Plain (∅ ▷ λ _ → Type)
           id (μ N tt tt) ! (toDesc (TreeOD N))
TrieOD N = Trie-desc N N (λ _ _ → con) id-MetaF
```

Continuing the analogy to `VecOD`, because `TreeOD` already sorts out how the parameters should be nested and how many fields have to be added, this ornament only has to add fields reflecting the recursive indices, and use these to report indices corresponding to the number of values of `A` contained in the numerical representation. We accomplish this by threading the partially applied constructors `n` of the number system `N` through the resulting description. In addition to generalizing over `Me` to facilitate the `δ` case, like in `TreeOD`, we also generalize over the index type `N'`. When mapping over descriptions, the choice of constructor also selects the corresponding constructor of `N'`.

```
Trie-desc : ∀ {Me} (N' : DescI Me ∅ τ) (D : DescI Me ∅ τ)
            (n : ⟦ D ⟧D (μ N') ⇉ μ N') (ϕ : MetaF Me Number)
            → OrnDesc Plain (∅ ▷ λ _ → Type)
              id (μ N' tt tt) ! (toDesc (Tree-desc D ϕ) )
Trie-desc N' [] n ϕ      = []
Trie-desc N' (C ∷ D) n ϕ = Trie-con N' C (λ p w x → n _ _ (inj₁ x)) ϕ
                          ∷ Trie-desc N' D (λ p w x → n _ _ (inj₂ x)) ϕ
```

We define `Trie-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 `Tree-con` left off, we can copy most fields

```
Trie-con : ∀ {Me} (N' : DescI Me ∅ τ) {re-var : Vxf id W V}
           {re-var′ : Vxf ! V U} (C : ConI Me ∅ U τ)
           (n : ∀ p w → ⟦ C ⟧C (μ N') (tt , re-var′ (re-var {p = p} w)) _ → μ N' tt tt)
           (ϕ : MetaF Me Number)
           → ConOrnDesc {Δ = ∅ ▷ λ _ → Type} {W = W} {J = μ N' tt tt} Plain
             {re-par = id} re-var ! (toCon (Tree-con {re-var = re-var′} C ϕ))
Trie-con N' (𝟙 {me = k} j) n ϕ
  = Oσ- _
  ( 𝟙 (λ { (p , w) → n p w refl }) (λ _ → refl))

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

Trie-con N' (σ S {me = f} h C) n ϕ
  = Oσ+ (S ∘ var→par _)
```

```
    ( Oσ- _
    ( Trie-con N' C (λ { p (w , s) x → n p w (s , x) }) ϕ))

  Trie-con N' (δ {me = me} {iff = iff} g j R C) n ϕ
    with ϕ .δf _ _ me
  ... | refl , refl , k
    = OΔσ+ (λ _ → μ R tt tt)
    ( •δ (λ ((_ , A) , _) → (_ , Vec A k)) (λ { (p , w , i) → i })
          (Trie-desc R R (λ _ _ → con) (ϕ ∘MetaF iff))
          (λ _ _ → refl) (λ _ _ → refl)
    ( Trie-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 `Trie-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 `Trie-con` is correct. In turn, we conclude that `Trie-desc` and `TrieOD` correctly construct indexed numerical representations.

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

Part IV
Discussion

Expectation:

We can define `PathOD` as a generic ornament from a `DescI Number` to the corresponding finite type, such that `PathOD ND n` is equivalent to `Fin (value n)`. Then, we can show that `itrieOD ND n` has a `tabulate`/`lookup` pair for `PathOD ND n`, from which it follows that `itrieOD ND n A` is equivalent to `PathOD ND n → A`, and in consequence `itrieOD ND` corresponds to `Vec`. From the `Recomputation` lemma it follows that the index `n` of `itrieOD ND n` corresponds to applying `ornForget` twice.

End A

Due to the `remember-forget` isomorphism [McB14], we have that `trieOD ND` is equivalent to `Σ (μ ND) (itrieOD ND)`, whence `trieOD ND` is a normal functor (also referred to as Traversable). This yields traversability of `trieOD ND`, and consequently `toList`[20].

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

[20] Note that the foldable structure we get from the generic `fold` is significantly harder to work with for this purpose.

This concludes a bunch of things, including this thesis. Combine

We know that the upper square in

$$\begin{tikzcd}
\text{itrie ND} \arrow[r, "\text{toVec}"] \arrow[d, "\text{forget}"'] & \text{Vec} \arrow[d, "\text{toList}"] \\
\text{trie ND} \arrow[r, "\text{toList}"] \arrow[d, "\text{forget}"'] & \text{List} \arrow[d, "\text{length}"] \\
\text{ND} \arrow[r, "\text{value}"'] & \mathbb{N}
\end{tikzcd}$$

commutes, and due to the recomputation lemma, the outer square also commutes. Because `ornForget` from `itrieOD ND` to `trieOD ND` is "epi" (that is, it covers by ranging over `n`), we find that the lower square also commutes.

Reality:

14 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[21] which do not have an obvious associated Agda datatype. As a

[21] Consider the constructor `σ ℕ λ n → power ρ n 𝟙` which takes a number `n` and asks for `n`

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.

15 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 themselves. 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

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.

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`.

16 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.

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`.

Note, we don’t bind deltas anymore

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`[22].

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[23]. Alternatively, 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 `Meta` 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.

20 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

[22] 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.

[23] https://github.com/agda/agda/pull/6385

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.

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 : Meta
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`.

21 No sparse numerical representations

\footnote{Consequently, this excludes the skew binary numbers \cite{oka95b} in their useful sparse representation, but this functionality can be regained by allowing for addition \emph{and} variable multiplication in a \AICσ{}. While not worked out in this thesis, this extension is compatible with the later constructions.}.
%The choice of interpretation restricts the numbers to the class of numbers which are evaluated as linear combinations of digits.
\footnote{An arbitrary \AF{Number} system is not necessarily isomorphic to \bN{}, as the system can still be incomplete (i.e., it cannot express some numbers) or redundant (it has multiple representations of some numbers).}. This class certainly does not include all interesting number systems, but does include many systems that have associated arrays\footnote{Notably, arbitrary polynomials also have numerical representations, interpreting multiplication as precomposition.}.

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 Random and friends *do* live in U-ix

Use `power` and indices.

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 constructie te omslachtig?

B Index-first

C Without K but with universe hierarchies

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

D Sigma descriptions

E ornForget and ornErase in full

F fold and mapFold in full