Utrecht University
Master Thesis
Computing Science & Mathematical Sciences

Generic Numerical Representations as Ornaments

Samuel Klumpers
6057314

Supervisors
dr. Wouter Swierstra
dr. Paige North

December 8, 2023

Abstract

The concept of numerical representations as defined by Okasaki [Oka98] explains how certain datastructures resemble number systems, and motivates how number systems can be used as a basis to design datastructures. Using McBride's ornaments [McB14], the method of designing datastructures starting from number systems can be made precise. In order to study a broad spectrum of indexed and unindexed numerical representations, we encode a universe allowing the expression of nested datatypes, and the internalization of descriptions of composite types. By equipping the universe with metadata, we can describe number systems and numerical representations in the same setup. Adapting ornaments to this universe allows us to generalize well-known sequences of ornaments, such as naturals-lists-vectors. We demonstrate this by implementing the indexed and unindexed numerical representations as ornament-computing functions, producing a sequence of ornaments on top of the number system.

Contents

1 Introduction **4**

2 Background **5**
2.1 Agda . . . 6
2.2 Datatypes in Agda . . . 6
2.3 Proving in Agda . . . 8
2.4 Descriptions . . . 10
2.4.1 Finite types . . . 10
2.4.2 Recursive types . . . 11
2.4.3 Sums of products . . . 12
2.4.4 Parametrized types . . . 12
2.4.5 Indexed types . . . 15
2.4.6 Generic Programming . . . 17
2.5 Ornaments . . . 18
2.6 Ornamental Descriptions . . . 21

3 Descriptions **23**
3.1 Numerical Representations . . . 23
3.2 Room for Improvement . . . 26
3.2.1 Number systems . . . 26
3.2.2 Nested types . . . 27
3.2.3 Composite types . . . 29
3.2.4 Hiding variables . . . 29
3.3 A new Universe . . . 30
3.3.1 Annotating Descriptions with Metadata . . . 33

4 Ornaments **35**
4.1 Ornamental descriptions . . . 35

5 Generic Numerical Representations **39**
5.1 Unindexed Numerical Representations . . . 40
5.2 Indexed Numerical Representations . . . 42

6 Conclusion **44**

7 Discussion **44**
7.1 Σ-descriptions are more natural for expressing finite types . . . 45
7.2 Branching numerical representations . . . 46
7.3 Indices do not depend on parameters . . . 46
7.4 No RoseTrees . . . 47
7.5 No levitation . . . 47
7.6 Metadata more tasteful externally than internally . . . 48
7.7 δ is conservative . . . 48
7.8 No sparse numerical representations . . . 49

Appendices **52**
A Folding . . . 52
B Folding without Axiom K . . . 52
C Nested types as uniformly recursive indexed types . . . 54

1 Introduction

There is a close relation between *number systems* and *datastructures* containing certain numbers of elements. Examples of datastructures designed to resemble a number system, are explored in Okasaki's Purely Functional Data Structures ([Oka98], chapter 9) as *numerical representations*, relating some known datastructures to their underlying number system.

To illustrate such an example, consider the binary numbers in their bijective, or zeroless, form (least significant digit first):

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

This definition declares that a binary number is either formed by `0b`, or by prepending either `1b` or `2b`. The number `0b` represents the number 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: 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
```

Similarly, a random-access list can be formed by `Zero`, or by prepending `One x` for some `x` in `A`, or by `Two x y` with both `x` and `y` in `A`. Note that in the recursive fields of `One` and `Two`, we pass the type of pairs `A × A` as the parameter rather than simply `A` (hence the non-uniformity). In this recursive field, a `One` would thus ask for two values of `A`, and another level deeper for four, and so on.

By forgetting that a random-access list `xs` has fields, we find a binary number `size xs` again:

```
size : Random A → Bin
size Zero         = 0b
size (One _   xs) = 1b size xs
size (Two x y xs) = 2b size xs
```

For example, applying `size` to `One _ (Two _ _ Zero)` gives us back `1b 2b 0b`. Additionally, this number given by `size` 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.

Using ornaments, we can formulate random-access lists as a patch on top of 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 we can construct all numerical representations for a certain generalization of positional number systems, and how some known examples of numerical representations fit into this framework. We make 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 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 implementing 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, a universe construction with this particular combination of features had not been studied before, hopefully allowing for further exploration of the interaction between features like non-uniform recursion and ornaments, and how incorporating generalized metadata can support more precise generic programming.

We formalize our work using the dependently typed proof assistant Agda [Tea23]. We use the unsafe `--type-in-type` option (manual page) so that the presented code is not diluted by the level variables, although our constructions can be modified to work without this flag [EC22]. We also use `--with-K`[1] and omit some type variables using variable generalization (manual page). The source code of this thesis can be found on `https://github.com/samuelhklumpers/master-thesis`.

2 Background

Many of our constructions extend upon or are inspired by 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.

This section describes some common datatypes and their usages, exploring how dependent types let us prove properties of programs, or write programs that are correct-by-construction. We then discuss certain proofs or programs can be generalized to classes of types by encoding datatypes using descriptions. Finally, we take a look at ornaments as a means to relate datatypes by their structure, or construct more datatypes of a given structure, but also as a way to identify comparable programs on structurally similar datatypes.

[1] In B Appendix B, we explain a variant on the universe which is compatible `--without-K`.

2.1 Agda

We formalize our work in the programming language Agda [Tea23]. Agda is a total functional programming language with dependent types. Using dependent types we can use Agda as a proof assistant, reinterpreting types as formulas and functions as proofs, allowing us to state and prove theorems about our datastructures and programs. Since Agda is total, and hence all functions are total, all functions of a given type always terminate in a value of that type. As a bonus, this rules out invalid proofs[2]. 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].

In this section, we will explain and highlight some parts of Agda which we use in the later sections. Many of the types and functions we define in this section are also described and explained in most Agda tutorials ([Nor09], [WKS22], etc.), and can be imported from the standard library [The23].

2.2 Datatypes 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 by declaring:

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

The constructors of this type state that values of `Bool` are produced in exactly two ways: `false` and `true`. We can then define functions on `Bool` by *pattern matching*, using that a variable of `Bool` is necessarily either `false` or `true`. 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 we pattern match on a variable of type `A`, in this case `Bool`, the coverage checker ensures we define the function on all possible cases, 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 natural numbers by encoding the existential axioms of the Peano axioms[3]. By pattern matching and *recursion* on `ℕ`, we define the less-than operator:

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

[2]On the other hand, we sometimes have to put in some effort to convince Agda that a function indeed terminates.

[3]The equality, injectivity, and induction axioms follow from the corresponding principles for arbitrary datatypes.

```
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 a *polymorphic* type. By parameterizing a definition, the context of that definition is extended with a variable of the type parametrized over. Parametrizing lists over a type, we can 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 is *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 does not lie in 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. This, however, merely replaces the check whether `lookup` returns `nothing` with a check if `n <? length xs` is before applying `lookup`.

More elegantly, we can combine natural numbers with an inequality 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 number 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. We can thus conclude 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. Similar to Fin, 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 constructor it was formed by. Namely, if n is zero, then xs is []. Otherwise, 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
```

We can now omit the [] case, where lookup? would return nothing. This happens because a variable 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 _∷_, which ensures that finding [] for xs is impossible. Consequently, lookup always succeeds for vectors. However, this does not yet prove that lookup necessarily returns the right element, and we will need some more logic to verify this.

2.3 Proving in Agda

To describe the 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 _≡_ to describe the behaviour of functions like lookup.

To test lookup, we can 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
```

If lookup is correct, then we expect the following to hold

```
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
```

which essentially states that we find elements where we insert them.

To prove lookup-insert-type, we proceed as when defining any other function. By simultaneous induction on the position i and vector xs, 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 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 correspondence*. For example, we can encode disjunctions (the logical 'or' operation) as

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

Conjunction (logical 'and') can be represented by:[4]

```
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 intentionally empty: since `⊥` has no constructors, there is no proof of false[5].

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 ∀ and ∃ 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 one free variable of type `A` by a function `A → Type` sending values of `A` to types, 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
```

[4] 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 $(a, b) = (c, d) \iff a = c \land b = d$ holds automatically.

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

```
    fst : A
    snd : P fst
```

so that we have `Σ A P` if and only if 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 $\exists a P(a)$.

2.4 Descriptions

In the previous sections we completed a quadruple of types (`ℕ`, `List`, `Vec`, `Fin`) equipped with the nice interactions `length` and `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:

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

This is no coincidence, but happens because `ℕ`, `List`, and `Vec` are structurally similar.

But how can we prove that datatypes have similar structures? In this section, we will explain a framework of datatype *descriptions* and ornaments, allowing us to describe datatypes as codes which can be compared directly, while also forming a foundation for generic programming in Agda [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[6].

Using datatype descriptions we can then relate `ℕ`, `List` and `Vec`, explaining how `length` and `toList` are instances of *forgetful* functions. 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`.

2.4.1 Finite types

An encoding of datatypes consists of two parts, a *type of descriptions* `U` of which the values are *codes* representing other datatypes, and an *interpretation* `U → Type` decoding those codes to the represented types. In the terminology of Martin-Löf type theory (MLTT)[Cha+10; Mar84], where types of types (e.g., `Type`) are called *universes*, we can think of a type of descriptions as an internal universe.

To start off, 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:

[6] As examples, consider the generic JSON encoding of suitable datatypes [VL14], or the derivation of functor instances for a broad class of types [Mag+10].

```
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:[7]

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

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

```
BoolD : U-fin
BoolD = 𝟙 ⊕ 𝟙
```

Since the types represented by `𝟘` and `𝟙` are finite, and sums and products of finite types are also finite, we refer to `U-fin` as the universe of finite types. From this, one can immediately conclude that there is no code in `U-fin` representing the (infinite) type of natural numbers `ℕ`.

2.4.2 Recursive types

We saw before that `ℕ` differs from `Bool` by having a recursive field. So, in order to make a universe which can encode `ℕ`, we begin by adding a code `ρ` to `U-fin` representing recursive type occurrences:

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

Then, we also have to redefine the interpretation: consider the interpretation of `𝟙 ⊕ ρ`, for which we need to know that the whole type was `𝟙 ⊕ ρ` while interpreting `ρ`. As a consequence, the interpretation splits in two phases.

In the first, we use functions from `Type` to `Type`[8] to represent types with one free type variable. Interpreting a code `D`, we use the free variable `X` to represent "the type D":

```
⟦_⟧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)
```

We can then model a recursive type by indeed setting the variable to the type itself, taking the *least fixpoint* of the functor:

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

Recall the definition of `ℕ`, which we can reinterpret as the declaration that `ℕ` is the fixpoint of `ℕ ≡ F ℕ` for `F X = ⊤ ⊎ X`. Hence, `ℕ` can simply be encoded as:

---

[7] 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`.

[8] We also refer to these functions as *polynomial functors*, which are polynomial here because they consist of sums and products, and are functors because they have a (functorial) mapping operation, as we will see later.

```
NatD : U-rec
NatD = 𝟙 ⊕ ρ
```

2.4.3 Sums of products

A downside of `U-rec` is that the definitions of types do not mirror their equivalents in user-written Agda very well. Using that polynomials can always be written as *sums of products*[9], we can define a similar universe which more closely resembles handwritten code.

Unlike the arbitrarily shaped polynomials formed by ⊕ and ⊗, a sum of products is analogous a datatype presented as a list of constructors. Thus, we split the descriptions into a stage in which we can form sums, equivalently datatypes

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

on top of a stage where we can form products, equivalently constructors:

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

When doing this, we also let the left-hand side of a product be any type, which allows us to represent ordinary fields. The interpretation of this universe, while similar to the one in the previous section, is also split into a part interpreting datatypes

```
⟦_⟧U-sop : U-sop → Type → Type
⟦ [] ⟧U-sop X = ⊥
⟦ C ∷ D ⟧U-sop X = ⟦ C ⟧C-sop X ⊎ ⟦ D ⟧U-sop X
```

and a part interpreting the constructors:

```
⟦_⟧C-sop : Con-sop → Type → Type
⟦ 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 = nilD ∷ consD ∷ []
  where
  nilD = 𝟙          -- : List A
  consD = σ A λ _ → --   A
          ρ         -- → List A
          𝟙         -- → List A
```

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

2.4.4 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

[9] We do not require these to be reduced, as different representations of the same polynomial represent different datatypes for us.

descriptions. As a consequence, we cannot inspect an arbitrary description in its entirety. At the same time, we could not express `List` fully internally, and needed to handle the parameter externally.

We can resolve both quirks simultaneously by introducing parameters and variables using a new gadget. In a naive attempt, we can represent the parameters of a type as `List Type`. However, this cannot represent some useful types. For example, the second parameter `B` of the existential quantifier `Σ_` has the type `A → Type`, which references back to the first parameter `A`. This referencing between parameters cannot be encoded in an ordinary list of parameters.

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 refer to as *telescopes* [Bru91] (also known as contexts in MLTT [Mar84]). 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, where we used the meaning of a telescope `⟦_⟧tel` to define types in the context of a telescope. This meaning represents valid assignments 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 enables us to write down the type of `Σ`:

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

To encode `Σ`, we will need to be able to bind the argument `a` of `A` and reference it in the field `P a`. While viable, a universe built around `Tel′` would awkwardly confuse parameters and bound arguments.

By quantifying telescopes over a type [EC22; Sij16], we can distinguish parameters and 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 V over the fixed parameter telescope Γ, which can be extended independently of Γ. An extension of Γ can also be interpreted in the context of Γ:

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

To describe conversions of telescopes, we give names to maps of telescopes and extensions:

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

Vxf : Cxf Δ Γ → ExTel Δ → 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, `var→par` states that a map of extensions extends to a map between the whole telescopes, and `Vxf-▷` lets us extend a map of extensions by acting as the identity on a new variable.

In the descriptions, the parameter telescopes are relayed directly to the constructors, but the variable telescope is reset to ∅ at the start of each constructor:

```
data U-par (Γ : Tel ⊤) : Type where
  []  : U-par Γ
  _∷_ : Con-par Γ ∅ → U-par Γ → U-par Γ
```

In the descriptions of constructors, we modify the σ code to request a type S in the context of V, which then also extends the context for the subsequent fields by S:

```
data Con-par (Γ : Tel ⊤) (V : ExTel Γ) : Type 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
```

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 ∘ var→par fst) (p , v , s)
```

where the σ case now provides the current parameters and variables to the field S, and extends the variables by s before passing them to the rest of the interpretation. In this universe, we can describe the parameters of lists with a one-type telescope:

```
ListD : U-par (∅ ▷ λ _ → Type)
ListD = nilD
     :: consD
     :: []
  where
  nilD  = 𝟙
  consD = σ (λ { ((_ , A) , _) → A })
        ( ρ
          𝟙)
```
This description declares that `List` has two constructors: the first has no fields and corresponds to `[]`, and the second has one ordinary field and one recursive field, corresponding to `_::_`. In the second constructor, we use pattern lambdas to deconstruct the telescope[10] and extract the type `A`.

Using the variable bound in `σ`, we can also give a description of the existential quantifier
```
SigmaD : U-par (∅ ▷ (λ _ → Type) ▷ λ { (_ , _ , A) → A → Type })
SigmaD = σ (λ { (((_ , A) , _) , _) → A } )  -- _,_  : (a : A)
       ( σ (λ { ((_ , B) , (_ , a)) → B a } ) --      → B a
         𝟙)                                   --      → Σ A B
       :: []
```
having one constructor with two fields. The first field of type `A` adds a value `a` to the variable telescope, which we pass to `B` in the second field by pattern matching on the variable telescope.

2.4.5 Indexed types

Lastly, we can integrate indexed types [DS06] into the universe by abstracting over indices:
```
data U-ix (Γ : Tel ⊤) (I : Type) : Type where
  []  : U-ix Γ I
  _::_ : Con-ix Γ ∅ I → U-ix Γ I → U-ix Γ I
```
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 : Tel ⊤) (V : ExTel Γ) (I : Type) : Type 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
```
Both `𝟙` and `ρ` now take a value of `I` in context `V`, where for `𝟙` this value represents the *actual index*, and for `ρ` it represents the *expected index* of the recursive field. Consider as an example `Fin`, where "`suc n`" bit of the constructor signatures tells us what the index of a constructor actually is, while the recursive type `Fin n` tells us which index the recursive field needs to have.

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 expected index of the declaration of this term. Hence, in the case of a leaf, we replace the unit type with the necessary equality between the expected `i` and actual index `j`

[10] Due to 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.

[McB14]. For a recursive field, the expected index j is then computed from the parameters and variables:

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

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

With indexed types, we can describe finite types and vectors[11] as

```
FinD : U-ix ∅ ℕ
FinD = zeroD ∷ sucD ∷ []
  where
  zeroD = σ (λ _ → ℕ)                         -- : (n : ℕ)
          ( 𝟙 (λ { (_ , (_ , n)) → suc n } ))  -- → Fin (suc n)
  sucD  = σ (λ _ → ℕ)                         -- : (n : ℕ)
          ( ρ (λ { (_ , (_ , n)) → n } )       -- → Fin n
          ( 𝟙 (λ { (_ , (_ , n)) → suc n } ))) -- → Fin (suc n)
```

and:

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

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. For completeness, let us replicate the natural numbers and lists in U-ix:

```
! : A → ⊤
! x = tt

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

ListD : U-ix (∅ ▷ λ _ → Type) ⊤
```

[11] Unlike some more elaborate encodings, we do not model implicit fields, so the descriptions of Fin and Vec look slightly different.

```
ListD = nilD ∷ consD ∷ []
  where
  nilD  = 𝟙 !
  consD = 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 an extra field `σ` of `A`. Likewise, `VecD` is the same as `ListD`, but now indexing over `ℕ` and with another extra field `σ` of `ℕ`. In Section 2.5, we will explain how this kind of analysis of descriptions can be performed formally inside Agda.

2.4.6 Generic Programming

As a bonus, we can also use `U-ix` for generic programming. For example, we can define the generic `fold` operation[12]:

```
_⇒₃_ : (X Y : A → B → Type) → Type
X ⇒₃ Y = ∀ a b → X a b → Y a b

fold : ∀ {D : U-ix Γ I} {X}
     → ⟦ D ⟧D-ix X ⇒₃ X → μ-ix D ⇒₃ X
```

From the point of view of category theory, we actually get `fold` for free: since `μ-ix D` is the least fixpoint, or initial algebra, of `⟦ D ⟧D`, `fold f` is simply the induced map to the algebra `f : ⟦ D ⟧D X ⇒₃ X`.

More concretely, we can view `⟦ D ⟧D X` as a variant of `μ-ix D`, in which the recursive positions hold values of `X` rather than other values of `μ-ix D`. For example, the type `⟦ ListD ⟧D X` reduces (up to equivalence) to `⊤ ⊎ (A × X A)`, substituting `X A` for what would usually be the recursive field.

In a sense, a term of `⟦ D ⟧D X` is a kind of `D`-structured set of values of `X`. From this perspective, `fold` roughly states that an operation `f`, collapsing `D`-structured sets of `X` into `X`, extends to a function `fold f`. This function sends a whole value of `μ-ix D` into `X`, recursively collapsing applications of `con` from the bottom up.

As an example, we can instantiate `fold` to `ListD`, which corresponds to

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

by the aforementioned equivalence. Much like the familiar `foldr` operation lets us consume a `List A` to produce a value `X A`; provided we give a value `X A` for the `[]` case, and a means to convert a pair `A × X A` to `X A` for the `_∷_` case.

Do note that this version of `fold` 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 the appropriate kind of maps into `C`. For example, by letting `X` be continuation-passing computations into `ℕ`, we can recover:

[12] The full construction can be found in A Appendix A.

```
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
```

2.5 Ornaments

In this section we will introduce a simplified definition of ornaments, and use it to compare descriptions. Simply looking at their descriptions, `ℕ` and `List` are rather similar, except that `List` has some things `ℕ` does not have. We could say that we can form the type of lists by starting from `ℕ` and adding a 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 description `D` to `E`, then `E` should be `D` with added fields, and more specific parameters and indices. This means that we should also be able to discard those extra fields, and revert to the less specific parameters and indices, obtaining a conversion from `E` to `D`. If `D` is a `U-ix Γ I` and `E` is a `U-ix Δ J`, then for a conversion from `E` to `D` to exist, there must also be functions `re-par : Cxf Δ Γ` and `re-index : J → I` for re-parametrization and 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 re-parametrizing with `re-par` and re-indexing 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)
```

An ornament then induces a conversion between types via 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 →₃ bimap (μ-ix D) re-par re-index
```

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

The definition of the constructor ornaments `ConOrn` controls the kinds of modification ornaments allow. Each allowed modification, equivalently each constructor of `ConOrn` also has to be reverted by `ornForget`. Accordingly, some modifications will have preconditions, which are in this case always pointwise equalities:

```
_~_ : {B : A → Type} → (f g : ∀ a → B a) → Type
f ~ g = ∀ a → f a ≡ g a
```

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`[13]. For example, the ornament `𝟙` states that if actual indices `i` and `j` are related, then the datatype constructors of the same names `𝟙 i` and `𝟙 j` are related.

By contrast, the `Δσ` ornament allows adding 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)
            : 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`. As expected, we see that there can only be an ornament from a description `D` to `E` if there are constructor ornaments for all constructors. Likewise, there can only be an ornament between constructors `CD` and `CE` if `CE` consists wholly of added fields and fields copied from `CD`, potentially refining parameters, variables, and indices.

Using these ornaments, we can make the claim that "lists are indeed natural numbers decorated with fields" more precise:

```
NatD-ListD : Orn ! id NatD ListD
NatD-ListD = nilO ∷ consO ∷ []
  where
```

[13]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.

```
  nilO  = 𝟙 (λ _ → refl)                   -- : List A
  consO = Δσ {S = λ { ((_ , A), _) → A }} -- : A
         ( ρ (λ _ → refl)                   -- → List A
         ( 𝟙 (λ _ → refl)))                 -- → List A
```

This ornament preserves most of the structure of ℕ, and only adds a field A using Δσ[14]. 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 re-indexing them over ℕ:

```
ListD-VecD : Orn id ! ListD VecD
ListD-VecD = nilO ∷ consO ∷ []
  where
  nilO  = 𝟙 (λ _ → refl)                         -- : Vec A zero
  consO = Δσ {S = λ _ → ℕ}                        -- : (n : ℕ)
        ( σ                                       -- → A
        ( ρ {j = λ { (_ , (_ , n) , _) → n }}     -- → Vec A n
            (λ _ → refl)
        ( 𝟙 {j = λ { (_ , (_ , n) , _) → suc n }} -- → Vec A (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 natural numbers (length), and from vectors to lists (toList) arise as ornForget, which we define using the fold over an algebra that erases a single layer of decorations:

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

Recursively applying this algebra, which reinterprets layers of E-data as D-data, 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-ix (bimap (μ-ix D) re-par re-index)
        →₃ 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" a single interpretation over an arbitrary type X:

```
ornErase : ∀ {re-par re-index} {X}
         → Orn re-par re-index D E
         →  ⟦ E ⟧D-ix (bimap X re-par re-index)
           →₃ bimap (⟦ D ⟧D-ix 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
```

[14]Note that S, and some later arguments we provide to ornaments, are implicit arguments: Agda would happily infer them from ListD and then later from VecD, had we omitted them.

```
                → ⟦ CE ⟧C-ix (bimap X re-par re-index)
                  →₃ bimap (⟦ CD ⟧C-ix 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
```

By pattern matching on the ornament, `conOrnErase` sees which bits of `CE` are new, and which are copied from `CD`. This tells us which parts of a term `x` under an interpretation of `CE` need to be forgotten, and which parts need 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, the `𝟙 sq` case tells us that `CD` is `𝟙 i'` and `CE` is `𝟙 j'`. Recalling that a leaf `𝟙 j'` at parameter `p` and expected index `j` is interpreted as the equality `j ≡ (j' p)`, we only need to produce the corresponding equality for `𝟙 i'`, which is `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 have to show that `x` can be converted from one `ρ` to the other `ρ` by translating its parameters, but in `σ` case, we can directly copy the field. Only the ornament `Δσ` adds a field, which is easily undone by removing that field.

In this way `ornForget` reinforces the idea that `E` is a decorated version of `D` when there is an ornament from `D` to `E` by associating to each value of `E` an underlying value in `D`. This additionally makes it easier to relate functions between related types. For example, instantiating `ornForget` for `NatD-ListD` yields `length`. Hence, the statement that `length` sends concatenation `_++_` to addition `_+_`, that is `length (xs ++ ys) ≡ length xs + length ys`, is equivalent to the statement that `_++_` and `_+_` are related by `ornForget`[15].

2.6 Ornamental Descriptions

By defining the ornaments `NatD-ListD` and `ListD-VecD` we demonstrated 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`.

If we somehow could leave out the right-hand side of ornaments, then we can also use ornaments to represent descriptions as patches on top of other descriptions. So, *ornamental descriptions* are precisely defined as ornaments without the right-hand side, effectively bundling a description and an ornament to it[16]. Their definition is analogous to that of ornaments, making the arguments which would only appear in the new description explicit:

```
data OrnDesc (Δ : Tel ⊤) (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)
```

[15] Equivalently, `_++_` is a lifting of `_+_` [DM14].

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

```
data ConOrnDesc (Δ : Tel ⊤) (W : ExTel Δ) (J : Type)
                (re-par : Cxf Δ Γ) (re-var : Vxf re-par W V)
                (re-index : J → I)
                : Con-ix Γ V I → Type where
  𝟙 : ∀ {i} (j : W ⊢ J)
    → re-index ∘ j ∼ i ∘ var→par re-var
    → ConOrnDesc Δ W J re-par re-var re-index (𝟙 i)

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

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

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

Using OrnDesc, we can describe ListD as a patch on NatD, inserting a σ in the constructor corresponding to suc:

```
ListOD : OrnDesc (∅ ▷ λ _ → Type) ⊤ ! ! NatD
ListOD = nilOD ∷ consOD ∷ []
  where
  nilOD  = 𝟙 (λ _ → tt) (λ _ → refl)    -- : List A
  consOD = Δσ (λ { ((_ , A) , _) → A }) -- : A
         ( ρ (λ _ → tt) (λ _ → refl)    -- → List A
         ( 𝟙 (λ _ → tt) (λ _ → refl)) ) -- → List A
```

Since an ornamental description simply represents a patch on top of a description, we can also extract the patched description and the ornament to it. To extract the description, 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)
```

which would extract ListD out of ListOD.

The other projection reconstructs the ornament to the extracted description

```
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)
```

and would extract `NatD-ListD` from `ListOD`. 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.

3 Descriptions

Before we can analyze number systems and their numerical representations using generic programs, we first have to ensure that these types fit into the descriptions. Some numerical representations are hard to describe using only the descriptions of parametric indexed inductive types `U-ix`. In order to keep things running smoothly for the generic programmer, we present an extension of `U-ix` incorporating metadata, parameter transformation, description composition, and variable transformation.

3.1 Numerical Representations

Before we start rebuilding our universe, let us look at the construction of the simplest numerical representation `Vec` from `ℕ`. At first, we defined `Vec` as the length-indexed variant of `List`, so that `lookup` becomes total and satisfies nice properties like `lookup-insert`. Later, we gave another description of `Vec` as an ornament on top of `List`. More abstractly, `Vec` is an implementation of finite maps with domain `Fin`. Here finite maps are simply those types with operations like `insert`, `remove`, `lookup`, and `tabulate`[17], 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 immediately satisfies the laws of finite maps, provided we define `insert` and `remove` correctly.

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

Unsurprisingly, `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`:[18]

```
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[19]. In terms of elements, this means that elements of `A` and `B` are in one-to-one correspondence.

Rather than deriving them ourselves, we can also establish properties like `lookup-insert` from this equivalence. Instead of finding the properties of `Vec` that were already there, let us view `Vec` as a consequence of the definition of `ℕ` and `lookup`. By 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 were solving an 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 isomorphisms readable, we construct them as chains of simpler isomorphisms using a variant of *equational reasoning* [The23; WKS22], which lets us compose isomorphisms 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))
```

If we allow reading isomorphisms as "*is*", then the terminology of Section 2.3, `⊥-strict` states that "if A is false, then A *is* empty", while `<-split` states that the set of numbers

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

[19] Compare this 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 $f(a) = b$.

below $n + 1$ has one more element than the set of numbers below n. Using these, we can calculate:[20]

```
Fin-def : ∀ n → Def (Σ[ m ∈ ℕ ] m < n)
Fin-def zero              =
  Σ[ m ∈ ℕ ] (m < zero) ≃⟨ ⊥-strict (λ ()) ⟩
  ⊥                     ≃-█ use-as-def
Fin-def (suc n)                   =
  Σ[ m ∈ ℕ ] (m < suc n)          ≃⟨ <-split n ⟩
  (⊤ ⊎ (Σ[ m ∈ ℕ ] m < n))        ≃⟨ cong (⊤ ⊎_) (by-definition (Fin-def n)) ⟩
  (⊤ ⊎ defined-by (Fin-def n)) ≃-█ use-as-def
```

This gives a different (but equivalent) definition of `Fin` compared to `FinD`; the description `FinD` describes `Fin` as an inductive family, whereas `Fin-def` describes `Fin` equivalently as a type-computing function [KG16]. From this `Def` we can extract a definition of `Fin`:

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

To derive `Vec`, we 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. With these laws, we calculate the type of vectors

```
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
```

yielding a definition of vectors and the `Iso` to `Lookup` in one go:

```
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)
```

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.

[20] Making 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.

3.2 Room for Improvement

We could now carry on and attempt to generalize this calculation to other 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.

3.2.1 Number systems

In the calculation `Vec` from `ℕ`, we analyzed and replicated the structure of `ℕ`. There, we used the meaning of these constructors as numbers assigned to them by our explanation of `ℕ` in words[21]. Based on that interpretation of constructors as numbers, we deliberately choose to add zero fields in the case corresponding to `zero` and choose to add one field in the case of `one`.

However, if we want to compute numerical representations generically, we also have to convince Agda 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`. Trivially, `ℕ` can be interpreted as a number system via `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 can describe 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 consists 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

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

like `Carpal`, which would otherwise still have valid numerical representations, as we will see later.

Across the above examples, the interpretation of a number is computed by a simple fold. In particular, leaves have associated constants, recursive fields correspond to multiplication and addition, while fields can defer to another function. We can thus modify `Con-sop` to encode each of these systems. The modified constructor descriptions `Con-num` associate a single number to each `𝟙` and `ρ`, and a function to each `σ`:

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

This essentially encodes number systems as trees that evaluate nodes by linearly combining values of subnodes, generalizing *dense* representations of positional number systems[22].

We can encode the examples we gave as follows:

```
Nat-num : U-num
Nat-num = zeroD ∷ sucD ∷ []
  where
  zeroD = 𝟙 0
  sucD  = ρ 1
        ( 𝟙 1 )
```

The binary numbers admit a similar encoding, but multiply their recursive fields by two instead:

```
Bin-num : U-num
Bin-num = 0bD ∷ 1bD ∷ 2bD ∷ []
  where
  0bD = 𝟙 0
  1bD = ρ 2
      ( 𝟙 1 )
  2bD = ρ 2
      ( 𝟙 2 )
```

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

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

3.2.2 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 have a description to begin with. However,

[22] As a consequence, this excludes the *sparse* number systems, as we discuss in Section 7.8.

the recursive fields of Random are not given the parameter A, but A × A. This makes Random a nested type, as opposed to a uniformly recursive type, in which the parameters of the recursive fields would be identical to the top-level parameters. Consequently, Random has no adequate description in U-ix[23].

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

{-# 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. We can encode Random by a HFun 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[24], so we settle for the weaker but safe inner nesting instead. Rather than placing the function that describes the nesting around the fixpoint like in HMu, we precisely emulate nested types which only modify their parameters.

When a type has parameters Γ, we can describe a change in parameters by a map g : Cxf Γ Γ from Γ to itself. So, we modify the recursive field ρ of U-ix to be

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

and update the interpretation of ρ to g 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 directly transcribed

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

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

Uniformly recursive types then simply become the nested types which only use the identity as a parameter transformation:

[23] Here, the "inadequate" descriptions either hardly resemble the user defined Random, use indices to store the depth of a node (which we work out in C Appendix C), or only have a complicated isomorphism to Random.

[24] As you might have deduced from the pragma disabling the positivity checker. Consider HBad F A = F A → ⊥.

```
ρ0 : ∀ {V} → V ⊢ I → Con-nest Γ V I → Con-nest Γ V I
ρ0 v C = ρ v id C
```

3.2.3 Composite types

In Section 3.2.1, we defined the number system `Carpal-num` as a *composite type*, in the sense that its description references another concrete type `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 complete numerical representation generically.

In comparison, generic programming facilities like the deriving-mechanism in Haskell allow for code like:

```
{-# LANGUAGE DeriveFunctor #-}

data Two a = Two a a deriving Functor
data Even a = Zero | More (Two a) (Even a) deriving Functor
```

In this example, we can define lists of even numbers of elements as lists of pairs of elements, and the Functor instance for `Even` can be derived generically, using that `Two` has a (derived) Functor instance. This would not work for `U-ix` or `U-num`, as a generic function would not be able to decide whether a field is of the form `μ D` to begin with.

Inlining the constructors of `Phalanx` into `Carpal` does allow generic constructions to see the structure of `Phalanx`, but is undesirable in this case and in general. Here, it would yield a type with two of the original constructors of `Carpal`, and 9 more constructors for each combination of constructors of `Phalanx`[25].

In order to make the descriptions of fields that have them visible to generics, we simply add a more specific former of fields to `U-ix` and call the resulting universe `U-comp` for now. The new former `δ` in `U-comp` is specialized to adding *composite fields* from provided descriptions:

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

This former then also has to take functions `d` and `j` to determine the parameters and indices passed to `R`. A composite field encoded by `δ` is then interpreted identically to how it would be if we used `σ` and `μ` instead[26]:

```
⟦ 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. Instead of inserting a plain field containing `Phalanx` and `toℕ-Phalanx`, we can use `δ` to directly add `Phalanx-num` to `Carpal-num`.

3.2.4 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 `S` added

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

[26] The omission of `μ R` from the variable telescope is intentional. While adding it is workable, it also significantly complicates the treatment of ornaments.

by a σ is treated as a bound or dependent argument: Even if the value `(s : S)` is then unused, all fields afterwards have to be treated as types depending on `S`. This only poses a minor inconvenience, but this does mean that two subsequent fields referring 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.

With a simple modification to the handling of telescopes 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 an attempt to save the user some effort, while leaving them 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
```

3.3 A new Universe

Using the modifications described above we define a new universe based on `U-ix`, in which the descriptions are again 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
```

The universe `DescI` is also parametrized over the metadata `Meta`, generalizing the annotations from Section 3.2.1 which we will use later to encode number systems in `DescI`.

The constructors of described datatypes can be formed 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 is the same as before, but ρ now accepts the transformation `Cxf Γ Γ` to encode non-uniform parameters. Likewise, σ takes a transformation `w` from `V ▷ S` to `W`, allowing us to replace the context `V ▷ S` after a field with a context `W` of our choice. 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 avoid cluttering descriptions which do not make use of the corresponding features.

In `DescI`, we can encode `ℕ` and `List` as before, replacing `σ` with `σ-` and `ρ` with `ρ0`:

```
NatD : Desc ∅ ⊤
NatD = zeroD ∷ sucD ∷ []
  where
  zeroD = 𝟙 _  -- : ℕ
  sucD  = ρ0 _ -- : ℕ
        ( 𝟙 _) -- → ℕ

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

If we define `Vec`, we bind the length as a (implicit) field, for which we use `σ+` instead, so we can extract the length `n` in `ρ0` and `𝟙`:

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

By passing a recursive field `ρ` the function taking `A` to `A × A`, we can almost repeat the definition of `Random` from `U-nest`:

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

Binary finger trees (as opposed to 2-3 finger trees [HP06]) are the numerical representation associated to `Carpal`. Like `Random`, they are a nested datatype, but instead store their elements in variably sized digits on both sides instead. In `DescI`, we can then first define digits as a datatype which holds one to three elements:

```
DigitD : Desc (∅ ▷ λ _ → Type) ⊤
DigitD = OneD ∷ TwoD ∷ ThreeD ∷ []
  where
  OneD   = σ- (λ ((_ , A) , _) → A) -- : A
           ( 𝟙 _)                   -- → Digit A
  TwoD   = σ- (λ ((_ , A) , _) → A) -- : A
           ( σ- (λ ((_ , A) , _) → A) -- → A
           ( 𝟙 _))                  -- → Digit A
  ThreeD = σ- (λ ((_ , A) , _) → A) -- : A
           ( σ- (λ ((_ , A) , _) → A) -- → A
           ( σ- (λ ((_ , A) , _) → A) -- → A
           ( 𝟙 _)))                 -- → Digit A
```

Then, we can use `δ` to include them into a separate description of finger trees:

```
FingerD : Desc (∅ ▷ λ _ → Type) ⊤
FingerD = EmptyD ∷ SingleD ∷ DeepD ∷ []
  where
  EmptyD  = 𝟙 _                              -- : Finger A
  SingleD = σ- (λ ((_ , A) , _) → A)         -- : A
            ( 𝟙 _)                           -- → Finger A
  DeepD   = δ  (λ (p , _) → p) _ DigitD      -- : Digit A
            ( ρ  (λ (_ , A) → (_ , (A × A))) _ -- → Finger (A × A)
            ( δ  (λ (p , _) → p) _ DigitD    -- → Digit A
            ( 𝟙 _)))                         -- → Finger A
```

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`, we can define a generic `fold` for `DescI`

```
fold : ∀ {D : DescI Me Γ I} {X} → ⟦ D ⟧D X →₃ X → μ D →₃ X
```

which comes in equally handy when using ornaments.

3.3.1 Annotating Descriptions with Metadata

We promised encodings of number systems in `DescI`, so let us explain how number systems can be described as *metadata* and how this lets use `DescI` in the same way we used `U-num` to describe type and numerical value in the same description.

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. The specific `Meta` passed to `DescI` determines the types of information to be queried (in the implicit `me` fields) at each of the type-formers. A term of `Meta` simply lists the type of information to be queried at each type former[27]:

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

In composite fields `δ`, the metadata on the other description is not necessarily the same as the top-level metadata. When this happens, we ask that both sides are 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
```

making it possible to downcast (or upcast) between the different types of metadata. This, for example, allows one to include 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
```

We let the composite field `δ`, which was not described when we discussed encoding number systems in `U-num`, act similar to `ρ`, also multiplying the value in its field by a constant. The equalities in the metadata of a `δ` ensure that number systems have no parameters or indices.

Using `Number`, we describe the binary numbers `Bin-num` in `DescI` as:

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

[27] One can compare this to how generic representations of datatypes in Haskell can be (optionally) annotated with metadata making the names of datatypes, constructors and fields available on the type level.

```
        ( 𝟙 {me = 1} _)
    2bD = ρ0 {me = 2} _
        ( 𝟙 {me = 2} _)
```

The metadata transformations help us when we represent `Carpal-num` in its more accurate form, by first defining

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

and directly including it into `Carpal`

```
CarpalND : DescI Number ∅ ⊤
CarpalND = 0cD ∷ 1cD ∷ 2cD ∷ []
  where
  0cD = 𝟙 {me = 0} _
  1cD = 𝟙 {me = 1} _
  2cD = 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 to indicate both sides have metadata of type `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
```

Furthermore, 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
```

On the other extreme, 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 _ _ = ⊤
```

Because the queries for metadata are implicit in `DescI`, descriptions from `U-ix` can be imported into `Desc`, without having to insert metadata anywhere.

4 Ornaments

In the framework of `DescI` of the last section, we can write down a number system and its meaning in one description, and we can use this as the starting point for constructing numerical representations. To write down a generic construction of numerical representations from number systems, 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. As we will be constructing new datatypes, rather than relating pre-existing ones, we omit the definition of the ornaments.

4.1 Ornamental descriptions

The ornamental descriptions for `DescI` take the same shape as those in Section 2.6, 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, has no direct influence on ornaments. So, we simply generalize over the metadata on `D`, querying the metadata for the new description without imposing constraints.

As always, we start off by defining ornamental descriptions as lists of constructor ornaments :

```
data OrnDesc {Me} (Me′ : Meta) (Δ : Tel ⊤)
             (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)
```

Most of the modifications in DescI are reflected in the constructor ornaments, and as a consequence this is also where we pay the price for the flexibility we built into ConI. For example, because ConI allows us to transform variables, ConOrnDesc has to relate the transformations on both sides in order for ornForget to exist. We (have to) dedicate a lot of lines to such commutativity squares of variables, but these squares involving Vxf can generally be ignored; this is 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
```

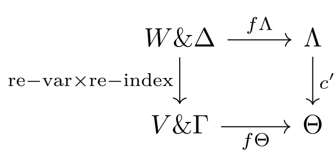
```
    → ConOrnDesc Me′ re-var re-index CD

  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 requires the relation of old and new variables.

Now, if we have a description `D'` with a composite field `δ R d j R D` referencing `R`, then we expect that a patch on `R` also induces a patch on `D'`. We generalize this by defining a kind of sequential composition of ornaments[28], taking two ornamental descriptions, one on `R` and one 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′}
    → (DE : ConOrnDesc Me′ re-var re-index CD)
    → ConOrnDesc Me′ re-var re-index (δ {Me} {me = me} {iff = iff} fΘ k R CD)
```

If we try to forget `∙δ`, the parameters to `R` can be computed in two ways. Namely, we can first convert back to the context of `CD` according to `DE` and compute the parameter for `R` there with the original `fΘ`, or we can first compute the parameter in the new context using the new `fΛ` and then convert this back to the parameter for `R` according to `RR′`. To avoid any ambiguity that arises from this, we require that both ways around this square are equal:



Using these and the other new commutativity squares, we can again define `ornForget` from an ornamental algebra analogous to the one for `U-ix`:

```
  ornForget : {re-var : Cxf Δ Γ} {re-index : J → I} {D : DescI Me Γ I}
            → (OD : OrnDesc Me′ Δ re-var J re-index D)
            → μ (toDesc OD) →₃ λ d j → μ D (re-var d) (re-index j)
  ornForget OD = fold (ornAlg OD)
```

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
```

[28] As opposed to Ko's parallel composition [Ko14], which composes two ornaments on the same description `D`, producing something that incorporates changes from both.

```
toCon (𝟙 j _ {me′ = me})
  = 𝟙 {me = me} j

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

toCon {re-var = v} (σ S h _ _ {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 _ _ {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′ _ _ {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.

Like DescI, the ornaments support variable transformations and nesting, of which we rarely utilize the full potential. In the common use-cases the commutativity squares the ornaments require are trivial, such as copying or adding (non-)dependent fields, and copying a uniformly recursive field. This means that we will mostly rely on the following shorthands to hide those trivial proofs:

`Oσ+ S CO`	`= σ S id _ (λ _ → refl) CO`	copy dependent field
`Oσ- S CO`	`= σ S fst re-var (λ _ → refl) CO`	copy non-dependent "
`OΔσ+ S CO`	`= Δσ S id (re-var ∘ fst) (λ _ → refl) CO`	insert dependent "
`OΔσ- S CO`	`= Δσ S fst re-var (λ _ → refl) CO`	insert non-dependent "
`Oρ0 j q CO`	`= ρ id j (λ _ → refl) q CO`	uniformly recursive "

With OrnDesc we can reproduce the examples of the ornamental descriptions for U-ix, such as Vec from List:

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

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

```
RandomOD : OrnDesc Plain (∅ ▷ λ _ → Type) ! ⊤ id BinND
RandomOD = ZeroOD ∷ OneOD ∷ TwoOD ∷ []
```

```
  where
  ZeroOD = 𝟙 _ (λ _ → refl)
  OneOD  = OΔσ- (λ ((_ , A) , _) → A)
         ( ρ (λ (_ , A) → (_ , Pair A)) _ (λ _ → refl) (λ _ → refl)
         ( 𝟙 _ (λ _ → refl)))
  TwoOD  = 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 = OneOD ∷ TwoOD ∷ ThreeOD ∷ []
  where
  OneOD   = OΔσ- (λ ((_ , A) , _) → A)
          ( 𝟙 _ (λ _ → refl))
  TwoOD   = OΔσ- (λ ((_ , A) , _) → A)
          ( OΔσ- (λ ((_ , A) , _) → A)
          ( 𝟙 _ (λ _ → refl)))
  ThreeOD = OΔσ- (λ ((_ , A) , _) → A)
          ( OΔσ- (λ ((_ , A) , _) → A)
          ( OΔσ- (λ ((_ , A) , _) → A)
          ( 𝟙 _ (λ _ → refl))))
```

and assemble these into finger trees with `δ•`

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

5 Generic Numerical Representations

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

In summary, using `DescI Number` to represent number systems, we paraphrase calculations like in Section 3.1 as ornaments, rather than direct definitions. 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.

5.1 Unindexed Numerical Representations

In this section we demonstrate the generic computation of numerical representations. We proceed differently from the calculation of Vec from ℕ. Indeed, we will give ornamental descriptions, rather than deriving direct definitions via step-by-step isomorphism reasoning. Nevertheless, the choices we make when inserting 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.

The unindexed representations are computed by TreeOD in the form of ornamental descriptions, sending a number system to the corresponding type of (nested) full trees over it. The ornament is computed by cases on the number system, and in each case the size of the numerical representation has to match up with the value of the number system.

Let us refer to the sole parameter of a numerical representation as A, and consider the case of a leaf of value k:

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

In this case, the leaf contributes a constant k to the value, so 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 ⊤ → ConI Number ∅ V ⊤
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 a recursive value of size x actually represents k * x values of A. On the other hand, an ordinary field S containing s, of which the value is computed as f s

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

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

```
δ-case : ℕ → DescI Number ∅ ⊤ → ConI Number ∅ V ⊤ → ConI Number ∅ V ⊤
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, which should, like the recursive field, let its values weigh k times their size.

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

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

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

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

Tree-con (σ S {me = f} h C) ϕ                            -- ...
  = Oσ+ S                                                -- → (s : S)
  ( OΔσ- (λ ((_ , A) , _ , s) → Vec A (ϕ .σf _ f _ s)) -- → Vec A (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 R (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 = OneOD ∷ TwoOD ∷ ThreeOD ∷ []
  where
  OneOD   = OΔσ- (λ ((_ , A) , _) → Vec A 1)
            ( 𝟙 _ (λ _ → refl))
  TwoOD   = OΔσ- (λ ((_ , A) , _) → Vec A 2)
            ( 𝟙 _ (λ _ → refl))
  ThreeOD = OΔσ- (λ ((_ , A) , _) → Vec A 3)
            ( 𝟙 _ (λ _ → refl))
```

which, after expanding vectors of `k` elements into `k` fields, is equivalent to the `DigitOD` from before. 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 = EmptyOD :: SingleOD :: DeepOD :: []
  where
  EmptyOD  = OΔσ- (λ ((_ , A) , _) → Vec A 0)
           ( 𝟙 _ (λ _ → refl))
  SingleOD = OΔσ- (λ ((_ , A) , _) → Vec A 1)
           ( 𝟙 _ (λ _ → refl))
  DeepOD   = •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, up to wrapping values in length one vectors and inserting empty vectors.

5.2 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
```

To continue the analogy to VecOD, we can use that TreeOD already sorts out how the parameters should be nested and how many fields have to be added. As a consequence, this ornament only has to add fields reflecting the recursive indices, which are used to report indices corresponding to the number of values of A contained in the numerical representation.

We accomplish this by threading the partially applied constructor n of the number system N through the resulting description; by feeding it all the sizes of the fields added by TreeOD, we can use n to compute the total size of an ornamented constructor.

In addition to generalizing over Me to facilitate the δ case as we did for TreeOD, we now also generalize over the index type N'. When mapping over the lists of constructors (i.e., 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') →₃ 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, binding the sizes of the subtries, to be fed as arguments to the selected constructor n. Since we are continuing where Tree-con left off, we can copy most fields:

```
Trie-con : ∀ {Me} (N : DescI Me ∅ ⊤) {re-var : Vxf id W V}
         → {re-var′ : Vxf ! V U} (C : ConI Me ∅ U ⊤)
         → (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                    -- ... n : N
  = Oσ- _                                        -- → Vec A k
  ( 𝟙 (λ { (p , w) → n p w refl }) (λ _ → refl)) -- → Trie ND A n

Trie-con N (ρ {me = k} g j C) n ϕ                        -- ... n : N × ⟦ C ⟧C N → N
  = OΔσ+ (λ _ → μ N tt tt)                               -- → (i : N)
  ( ρ (λ { (_ , A) → _ }) (λ { (p , w , i) → i })        -- → Trie ND (Vec A k) i
      (λ _ → refl) (λ _ → refl)
  ( Trie-con N C (λ { p (w , i) x → n p w (i , x) }) ϕ)) -- ... curry n i

Trie-con N (σ S {me = f} h C) n ϕ                        -- ... n : S × ⟦ C ⟧C N → N
  = Oσ+ (S ∘ var→par _)                                  -- → (s : S)
  ( Oσ- _                                                -- → Vec A (f s)
  ( Trie-con N C (λ { p (w , s) x → n p w (s , x) }) ϕ)) -- ... curry n s

Trie-con N (δ {me = me} {iff = iff} g j R C) n ϕ
  with ϕ .δf _ _ me
... | refl , refl , k                                    -- ... n : R × ⟦ C ⟧C N → N
  = OΔσ+ (λ _ → μ R tt tt)                               -- → (r : R)
  ( •δ (λ ((_ , A) , _) → (_ , Vec A k))                 -- → Trie R (Vec A k) r
       (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) r → n p w (i , r) }) ϕ)) -- ... curry n r
```

We only have to add fields in the cases for ρ and δ, and in both they are promptly passed as expected indices to the next field using `λ { (p , w , i) → i }`. The only difference is that 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 a 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 i bound in the ρ case forces the number of elements in the recursive field to i, the value in the σ case corresponds to the number of elements added after this field. Likewise, the R-index i bound in the δ case forces the number of elements in the subdescription to be i. Hence, when we arrive at a leaf 𝟙, we know that the total number of elements is exactly given by n, and thus Trie-con is correct. In turn, we find that Trie-desc and TrieOD correctly construct indexed numerical representations.

6 Conclusion

In conclusion, we formulated a universe encoding `DescI`, adapted the language of ornamental descriptions `OrnDesc` to it, and implemented generic programs to calculate numerical representations from number systems in `DescI Number`.

With the program `TreeOD`, we can describe all datastructures we used as examples in other sections: `List`, `Random`, `Finger`, and many more. For example, we can also replicate (nested variants of) the constructions of binomial heaps as an ornament on binary numbers by Ko [Ko14], (dense) skew binary random-access lists and heaps, and their variants in higher bases than binary [Oka98].

On top of this, `TrieOD` lets us describe indexed variants of those datastructures, such as `Vec`, and lets us replay part of the argument to derive indexed random-access lists from binary numbers due to Hinze and Swierstra [HS22].

In turn, the numerical representations immediately enjoy both the generic programs we get for all descriptions (such as `fold`), and the functions we get from their nature as ornaments over number systems (like `length` or `toList`). Furthermore, due to their specific construction, we could also define a kind of extensional equality for numerical representations: We only need decidable equality of the element type, as all other fields are only relevant up to numerical value. Similarly, we can generalize the "forall" and "exists" predicates for W-types[29] to all numerical representations, using that `TreeOD` only ever nests over `Vec`.

The treatment of numerical representations as ornaments on number systems also makes it easier to ask when operations on the number system induce or inspire operations on the datastructure. For example, if we define addition on a number system such that it agrees with `_+_` on `ℕ`, we can use this as inspiration to define concatenation on the datastructure. The work of Dagand and McBride on functional ornaments [DM14] makes it clear when function types can be related by ornaments, which coherences this induces between functions, and how this can help the programmer to directly write functions satisfying those coherences. Effectively, this lets us give a number system and its addition, and get the specification of concatenation on the numerical representation for free.

7 Discussion

Our implementation does have some drawbacks, and also leaves some open questions, which we try to outline in the following sections.

For example: While it is possible to write down a direct proof of correctness for `TrieOD` by comparing it to `Lookup` via `value`, and from this extract a proof of correctness for `TreeOD`, one might expect there to be a more useful and less laborious angle of attack.

Namely, we expect that if we 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 `TrieOD ND n` has a `tabulate`/`lookup` pair for `PathOD ND n`. This proves that `TrieOD ND n A` is equivalent to `PathOD ND n → A`, and in consequence `TrieOD ND` corresponds to `Vec`.

[29] See the Agda standard library `□` predicate for containers.

Due to the remember-forget isomorphism [McB14], we have that TreeOD ND is equivalent to Σ (μ ND) (TrieOD ND), and in turn we also find that TreeOD ND is a normal functor (also referred to as Traversable). This yields traversability of TreeOD ND, with as corollaries toList[30] and properties such as that toList is a lifting of value (again in the sense of [DM14]).

However, it turns out that PathOD is not so easy to define, as we can see by the following.

7.1 Σ-descriptions are more natural for expressing finite types

Due to our representation of types as sums of products, representing the finite types of larger number systems quickly becomes much more complex. Consider the binary numbers from before:

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

Suppose FinB is the finite type associated to Bin. Since the value of 1b n is $2n + 1$, the type FinB (1b n) should be isomorphic to FinB n ⊎ FinB n ⊎ ⊤. While we can reorganize the first two summands into a product with Fin 2 instead, the last summand has a different structure.

For a general number system N, the number and structure of constructors of the finite type FinN associated to N depends directly on the interpretation of N, preventing the construction of FinN by simple recursion[31] on DescI.

Since ornaments preserve the number of constructors, there cannot be an ornament from number systems to their finite types[32].

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 universe with *Σ-descriptions* [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.

Compared to Σ-descriptions, a sums-of-products universe keeps out some more exotic descriptions which do not have an obvious associated Agda datatype[33].

However, this also prevents us from writing down the simpler form of finite types. 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
```

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

[31] That is, without passing up lists of ConI to be assembled at the level of DescI again.

[32] An "intuitive" ornament anyway. It is possible to insert a Three field in 0b of Bin, and then compute the index using λ { one → 1b_ ; two → 2b_ ; three → 2b_ }. However, this shoves the responsibilities of 1b_ and 2b_ onto 0b, is as awkward as passing up lists of ConI, and destroys the useful property that ornForget x lines up with the index of x.

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

In this universe we can present the binary numbers as:

```
BinΣD : Σ-Desc ⊤
BinΣD = 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 Bin
FinBΣD = σ (Fin 3) λ
  { zero             → σ (Fin 0) λ _ → 𝟙 0b
  ; (suc zero)       → σ Bin 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 Bin 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 `Bin`, and as a consequence also the numerical representation associated to `Bin`, this would simplify proving that the indexed numerical representation is indeed equivalent to the representable representation (the maps out of `FinB`). In a framework of ornaments for Σ-descriptions [KG16; McB14], we can even describe the finite type as an ornament on the number system.

7.2 Branching numerical representations

A numerical representation constructed by `TrieOD` looks like a finger tree: the structure typically has a central chain, which rather than directly storing elements directly in nodes, stores the elements in trees of which the depth increases with the level of the node.

For contrast, compare this to structures like Braun trees, as Hinze and Swierstra [HS22] compute from binary numbers, and to the binomial heaps [Ko14] Ko constructs. These structures reflect the weight of a node using branching rather than nesting. Because this kind of branching is uniform, i.e., each branch looks the same, we can still give an equivalent construction. By combining `TreeOD` and `TrieOD`, and using to apply `ρ` `k`-fold in the case of `ρ {if = k}`, rather than over `k`-element vectors, we can replicate the structure of a Braun tree from `BinND`. However, if we use the Σ-descriptions we discussed above, we can more elegantly present these structures by adding an internal branch over `Fin k`.

7.3 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 some useful types.

Allowing indices to depend on parameters lets us describe some types that could be computed generically for numerical representations like the membership relation: It is essential that the `List A` is an index, since each constructor constructs the relation at a different list. If we do not want to rely on `--type-in-type`, the variable `A` must be a parameter, as it would otherwise push `_∈_` up one level. Moreover, the sort of a type can depend on its parameters, but not its indices, so the level of `A` must also be a parameter.

Likewise, indices have to depend on parameters in order to formulate *algebraic ornaments* [McB14] in `OrnDesc` in their fully general form. This is also the case for *singleton types*, which can be used to compute the additional information needed to invert `ornForget`.

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.

7.4 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`[34].

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[35]. 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.

7.5 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 additional features of `DescI`, i.e., parameters, nesting, and composition, would not obstruct a similar levitating variant of `DescI`. Using the concept of functional ornaments [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`.

[34] 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.

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

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

7.6 Metadata more tasteful externally than internally

On the other hand, while incorporating general metadata into `DescI` works out neatly in our case, and in general seems to work out if we think about one use-case at a time, it might not work so nicely in other situations. For example, if we are working with `Number`, but we are given a `DescI Plain` (i.e., `Desc`), then we would have to duplicate that description in `DescI Number` before we could use it. Even worse, if we want to give the constructors of a number system nice names using `Names`, we would have to rewrite our code and descriptions to use something like the product of `Number` and `Names`.

It might be more portable to take the same approach in handling metadata as True sums of products [VL14], where metadata is described externally to the universe and only combined again if needed by a generic function. From this point of view, a type of metadata can simply be a convenient function from `Desc` to `Type`. If `Number` was presented in this way, then `TreeOD` would not have to ask for `DescI Number`, but rather for a `D` of `Desc` paired with `Number D`.

7.7 δ is conservative

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

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, leaving out some details, 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 equivalently be modelled by first adding a new field `μ R'`, and then deleting the original `μ R` field. The ornament `∇` [Ko14] allows one to provide a default value for a field, deleting it from the description. Hence, we can model `•δ` by binding a value `r'` of `μ R'` with `OΔσ+` and deleting the field `μ R` using a default value computed by `ornForget`.

This also partially explains why we did not refer to algebraic ornaments at all in our construction of `TrieOD`; We can see that `TrieOD` looks very similar to the algebraic ornament over `TreeOD`, which sends ornaments from `D` to `E` to an ornament to a `D`-indexed variant of `E`. However, the case of `δ` requires `TrieOD` to step in and re-index the subdescription. In contrast, the algebraic ornament would simply treat a `δ` like its equivalent `σ`. Even though this would produce a correct numerical representation, this amounts to presenting a `Vec` as a tuple of a length `n`, a `List` `xs`, and a proof that `n` is equal to `length` `xs`.

Thus, while it would be possible to present `TrieOD` as a kind of algebraic ornament, this would require redefining algebraic ornaments from algebras that are rather specific about how they treat a `σ`.

7.8 No sparse numerical representations

The encoding of number systems in a universe we explained in Section 3.2.1 corresponds to a generalization of dense number systems. Consequently, this excludes the skew binary numbers [Oka95] in their useful sparse representation.

Representations of sparse number systems can be regained by allowing addition *and* variable multiplication in a `σ`. In such a setup, skew binary numbers and other sparse representations could be described by adding their gaps as fields, and computing the appropriate multiplier from there. While not worked out in this thesis, this extension is compatible with the construction of numerical representations.

Another notable extension of `Number` is to let some recursive and composite fields be interpreted by multiplication, with which we could equip `U-fin` with its obvious interpretation into `ℕ`. This can be compared to the last exponential law we did not use in Section 3.1, which is that $A^{BC} = (A^B)^C$. Furthermore, any indexed numerical representation acts as a representable functor `F`. If `F` and `G` are numerical representations corresponding to number systems `N` and `M`, then the multiplication of `N` and `M` just corresponds to composition `F ∘ G`.

Acknowledgments

I would like to thank Wouter Swierstra for all the fun discussions, suggestions, and helpful feedback. I also want to thank Paige North for the interesting reading suggestions, and taking the time to help me with my questions about related bits of category theory.

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`.

[Mar84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[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`.

[Oka95] Chris Okasaki. "Purely Functional Random-Access Lists". In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95. La Jolla, California, USA: Association for Computing Machinery, 1995, pp. 86–95. ISBN: 0897917197. DOI: `10.1145/224164.224187`. URL: `https://doi.org/10.1145/224164.224187`.

[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/`.

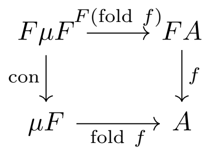
Appendices

A Folding

In Section 2.4.6 and Section 2.5 we used `fold` as a concept to explain a bit of generic programming. We give its definition here, but for `DescI` instead, since the fold of `U-ix` can be seen as a simplification of it.

```
fold : ∀ {D : DescI Me Γ I} {X} → ⟦ D ⟧D X →₃ X → μ D →₃ X
```

As `fold f` is the algebra map `con ⇒ f`, the following commutes:



However, by defining `fold f (con x)` as `f (map (fold f) x)`, we prevent the termination checker from seeing that `fold` is only applied to terms strictly smaller than `x` (much like our fellow universe constructions find out somewhere along the line). To help out the termination checker, we inline `fold` into `map`, which gives us an equivalent definition:

```
mapDesc : ∀ {D' : DescI Me Γ I} (D : DescI Me Γ I) {X}
        → ∀ p i → ⟦ D' ⟧D X →₃ X
        → ⟦ D ⟧D (μ D') p i → ⟦ D ⟧D X p i

mapCon : ∀ {D' : DescI Me Γ I} {V} (C : ConI Me Γ V I) {X}
       → ∀ p i v → ⟦ D' ⟧D X →₃ X
       → ⟦ C ⟧C (μ D') (p , v) i → ⟦ C ⟧C X (p , v) i

fold f p i (con x) = f p i (mapDesc _ p i f x)

mapDesc (C ∷ D) p i f (inj₁ x) = inj₁ (mapCon C p i tt f x)
mapDesc (C ∷ D) p i f (inj₂ y) = inj₂ (mapDesc D p i f y)

mapCon (𝟙 j)       p i v f x       = x
mapCon (ρ g j C)   p i v f (r , x) = fold f (g p) (j (p , v)) r
                                   , mapCon C p i v f x
mapCon (σ S w C)   p i v f (s , x) = s , mapCon C p i (w (v , s)) f x
mapCon (δ d j R C) p i v f (r , x) = r , mapCon C p i v f x
```

Here `mapDesc` (and `mapCon`) simply peel off and reassemble all non-recursive structure, applying `fold` to the recursive fields; `fold` is then defined in the usual way by applying its algebra `f` to itself mapped over `x`.

B Folding without Axiom K

The axiom of univalence (or cubical type theory) gives us another interesting context to study ornaments in. In the way we presented it, the theory of ornaments produces a lot of isomorphisms from relations between types, which are not yet as powerful as they could be when comparing properties between related types. Univalence gives us the means to turn equivalences[36] into equalities, allowing us to put an isomorphism between types to work by transporting properties over it.

Unfortunately, a direct port of ornaments into `--cubical` is quickly thwarted by the absence of Axiom K, as one would discover that the definitions of `mapDesc` and `mapCon` illegally pattern match on the types calculated by interpretations[37].

This can be remedied by presenting interpretations as datatypes[38]. Effectively, we are applying the duality between type computing functions and indexed types. Since `Desc` and `Con` are unindexed types, they cannot accidentally carry equational content, and pattern matching on them does not generate transports in `⟦_⟧D` and `⟦_⟧C`. Hence, the definition of `fold` is (morally speaking) safe.

With that out of the way, we can define the interpretations as indexed types:

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

  data IntpC (X : ⟦ Γ ⟧tel tt → I → Type)
             (pv : ⟦ Γ & V ⟧tel) (i : I)
             : Con Γ V I → Type where
    𝟙-i : ∀ {i'}
        → i ≡ i' pv → IntpC X pv i (𝟙 i')

    ρ-i : ∀ {g i' D}
        → X (g (pv .fst)) (i' pv) → IntpC X pv i D
        → IntpC X pv i (ρ g i' D)

    σ-i : ∀ {S D} {w : Vxf id (V ▷ S) W}
        → (s : S pv) → IntpC X (pv .fst , w (pv .snd , s)) i D
        → IntpC X pv i (σ S w D)

    δ-i : ∀ {d j D} {R : Desc Δ J}
        → (s : μ R (d pv) (j pv)) → IntpC X pv i D
        → IntpC X pv i (δ d j R D)

  data IntpD (X : ⟦ Γ ⟧tel tt → I → Type)
             (p : ⟦ Γ ⟧tel tt) (i : I)
             : Desc Γ I → Type where
    ∷-il : ∀ {C D} → IntpC X (p , tt) i C → IntpD X p i (C ∷ D)
    ∷-ir : ∀ {C D} → IntpD X p         i D → IntpD X p i (C ∷ D)

⟦_⟧D : Desc Γ I → (⟦ Γ ⟧tel tt → I → Type) → ⟦ Γ ⟧tel tt → I → Type
⟦_⟧D = λ D X p i → IntpD X p i D

⟦_⟧C : Con Γ V I → (⟦ Γ ⟧tel tt → I → Type) → ⟦ Γ & V ⟧tel → I → Type
⟦_⟧C = λ C X pv i → IntpC X pv i C
```

[36] Equivalences can be considered as a correction to isomorphisms for types which are not sets (in the sense of being discrete); since all types we describe here are sets, equivalences and isomorphisms coincide.

[37] The Without K documentation explains why pattern matching on non-datatypes is not safe in general.

[38] Albeit a bit dubiously; at the time of writing, this is also how you can circumvent a restriction on pattern matching emplaced by `--cubical-compatible`, see the relevant GitHub issue.

Since the interpretations are datatypes now, we can pattern match on them to define `mapDesc` and `mapCon` in a way that is accepted:

```
mapDesc : ∀ {D' : Desc Γ I} (D : Desc Γ I) {X}
        → ∀ p i → ⟦ D' ⟧D X →₃ X → ⟦ D ⟧D (μ D') p i → ⟦ D ⟧D X p i

mapCon : ∀ {D' : Desc Γ I} {V} (C : Con Γ V I) {X}
       → ∀ p i v → ⟦ D' ⟧D X →₃ X → ⟦ C ⟧C (μ D') (p , v) i → ⟦ C ⟧C X (p , v) i

fold f p i (con x) = f p i (mapDesc _ p i f x)

mapDesc (C ∷ D) p i f (∷-il x) = ∷-il (mapCon C p i tt f x)
mapDesc (C ∷ D) p i f (∷-ir y) = ∷-ir (mapDesc D p i f y)

mapCon (𝟙 j)      p i v f
       (𝟙-i   x) = 𝟙-i x

mapCon (ρ g j C)  p i v f
       (ρ-i r x) = ρ-i (fold f (g p) (j (p , v)) r) (mapCon C p i v f x)

mapCon (σ S w C)  p i v f
       (σ-i s x) = σ-i s (mapCon C p i (w (v , s)) f x)

mapCon (δ d j R C) p i v f
       (δ-i r x) = δ-i r (mapCon C p i v f x)
```

C Nested types as uniformly recursive indexed types

Although `U-ix` has no direct support for expressing nested types, we can actually give equivalent encodings for some of them.

Indeed, indices are readily repurposed as parameters. If we apply this to random-access lists, we can write:

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

More interestingly, perhaps, the depth of a random-access list determines the types of its fields. Namely, `One` will ask for 1 element at the highest level, one level down it asks for 2, and one more it asks for 4, and so on. Hence, in a way that vaguely resembles defunctionalization, we can define

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

data Pair (A : Type) : Type where
  pair : A → A → Pair A
```

and describe a field at depth `n` by `power n Pair A`. This can be applied to describe random-access lists which track their depth in their index instead:

```
RandomD-2 : U-ix (∅ ▷ const Type) ℕ
RandomD-2 = σ (λ _ → ℕ)
          ( 𝟙 λ { (_ , (_ , n)) → n })
          ∷ σ (λ _ → ℕ)
          ( σ (λ { ((_ , A) , (_ , n)) → power n Pair A })
          ( ρ (λ { (_ , ((_ , n) , _)) → suc n })
          ( 𝟙 λ { (_ , ((_ , n) , _)) → n } )))
          ∷ σ (λ _ → ℕ)
          ( σ (λ { ((_ , A) , (_ , n)) → power (suc n) Pair A })
          ( ρ (λ { (_ , ((_ , n) , _)) → suc n })
          ( 𝟙 λ { (_ , ((_ , n) , _)) → n } )))
          ∷ []
```

Since we cannot (yet) construct path types generically (Section 7.1), we cannot make this construction generic. If we did have such constructions, the argument for random-access lists generalizes to an operation that splits a nested datatype `D` into three parts:

1. a type of paths in `D` (not necessarily pointing to a field)
2. a lookup function that sends a path to the accumulated parameter transformation
3. the (uniform) datatype, indexed over the paths, using the lookup function to calculate the types of its fields.