Ornaments and Proof Transport applied to Numerical Representations

Samuel Klumpers
6057314

September 29, 2023

Abstract

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

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

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

long

distracted

Contents

Todo list

1 Introduction

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

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

We use the language Agda \cite{agda}, of which the
    dependent types form the logic we use to specify and
    verify the code we write.
```

'Programming is hard' - citation needed? Misschien beter om de nadruk te leggen op iets als 'statically typed programming languages can rule out certain errors before a program is executed'? Of misschien: the development of complex programs hits the limits of what humans can understand? Zoiets?

```
In our approach, we describe a part of the language
    inside the language itself. This allows us to reason
    about the structure of other code using code itself.
    Such descriptions of code can then be interpreted to
    generate usable code. Using constructions known as
    ornaments \cite{algorn, sijsling}, we can also discuss
     how we can transform one piece of code into another
    by comparing the descriptions of the two pieces.
```

Maar misschien is het nog beter om een iets andere opening gambit te kiezen

We will describe and then generate a class of container types (which are types

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

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

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

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

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

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

Dan kun je de onderzoeksvraag duidelijk maken: dit is geen toeval, maar wat is de relatie dan wel? Ik denk dat een uitgewerkt voorbeeld, ook al is die 'bekend' zoals random-access lijsten oid hier heel nuttig zou kunnen zijn.

Background

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

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

In 6 - misschien wel goed om hier references toe te voegen? Denk aan true sums of products, de tutorial over generic programming met dependent types van Morris, Altenkirch & McBride, ... In de inleiding ook goed

2 Agda

•Add more citations and then double check them below here We formalize our work in the programming language Agda [Tea23]. While we will only occasionally reference Haskell, those more familiar with Haskell may look at Agda (in rough approximation) as an extension of Haskell[**?**]. Agda is a total functional programming language with dependent types. Here, totality means that functions of a given type always terminate in a value of that type, ruling out non-terminating (and not obviously terminating) programs. Using dependent types we can use Agda as a proof assistant, allowing us to state and prove theorems about our datastructures and programs.

extension? reasonable agda is haskell, so agda kind of embeds into haskell

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

We use `--type-in-type` to keep the explanations more readable.

3 Data in Agda

At the level of generalized algebraic datatypes, Agda is close to Haskell. In both languages, one can define objects using data declarations, and interact with them using function declarations. For example, we can define the type of *booleans*:

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

The constructors of this type state that we can make values of `Bool` in exactly two ways: `false` and `true`. We can then define functions on `Bool` by pattern matching. As an example, we can define the conditional operator as

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

When *pattern matching*, the coverage checker ensures we define the function on all cases of the type matched on, and thus the function is completely defined.

We can also define a type representing the natural numbers

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

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

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

```
suc n <? suc m = n <? m
```

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

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

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

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

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

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

However, this function *partial*, as we are relying on the type

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

to handle the case where the position falls outside the list and we cannot return an element. If we know the length of the list `xs`, then we also know for which positions `lookup` will succeed, and for which it will not. We define

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

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

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

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

Like parameters, indices add a variable to the context of a datatype, but unlike parameters, indices can influence the availability of constructors. The type `Fin` is defined such that a variable of type `Fin n` represents a number less than `n`. Since both constructors `zero` and `suc` dictate that the index is the `suc` of some natural `n`, we see that `Fin zero` has no values. On the other hand, `suc` gives a

value of `Fin (suc n)` for each value of `Fin n`, and `zero` gives exactly one additional value of `Fin (suc n)` for each `n`. By induction (externally), we find that `Fin n` has exactly `n` closed terms, each representing a number less than `n`.

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

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

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

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

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

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

4 Proving in Agda

To describe equality of terms we define a new type

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

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

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

we can express the correctness of `lookup` as

```
lookup-insert-type : ∀ {n} → Vec A n → Fin (suc n) → A → Type
lookup-insert-type xs i x = lookup (insert xs i x) i ≡ x
```

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

To prove the statement, we proceed as when defining any other function. By

simultaneous induction on the position and vector, we prove

```
lookup-insert : ∀ {n} (xs : Vec A n) (i : Fin (suc n)) (y : A)
              → lookup-insert-type xs i y
lookup-insert []       zero    y = refl
lookup-insert (x :: xs) zero    y = refl
lookup-insert (x :: xs) (suc i) y = lookup-insert xs i y
```

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

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

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

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

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

True and false are respectively represented by

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

so that always `tt : ⊤`, and

```
data ⊥ : Type where
```

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

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

```
¬_ : Type → Type
¬ A = A → ⊥
```

The logical quantifiers ∀ 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 a free variable of type `A` by a function values of `A` to types `A → Type`, also known as a predicate. The universal quantifier $\forall a P(a)$ is true when for all a the formula $P(a)$ is true, so we represent the universal quantification of a predicate `P` as a dependent function type `(a : A) → P a`, producing for each `a`

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

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

of type `A` a proof of `P a`. The existential quantifier $\exists a P(a)$ is true when there is some a such that $P(a)$ is true, so we represent the existential quantification as

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

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

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

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

5 Descriptions

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

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

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

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

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

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

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

5.1 Finite types

A datatype description, which are datatypes of which each value again represents a datatype, consist of two components. Namely, a type of descriptions `U`, also

referred to as codes, and an interpretation `U → Type`, decoding descriptions to the represented types. In the terminology of Martin-Löf type theory[Cha+10], where types of types like `Type` are called universes, we can think of a type of descriptions as an internal universe.

No citation for MLTT? Agda is a rather loose extension, none of the original papers really match.

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

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

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

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

which indeed sends `𝟘` to `⊥`, `𝟙` to `⊤`, sums to sums and products to products[3].

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

```
BoolD : U-fin
BoolD = 𝟙 ⊕ 𝟙
```

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

5.2 Recursive types

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

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

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

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

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

[3] 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`.

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

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

```
ℕD : U-rec
ℕD = 𝟙 ⊕ ρ
```

5.3 Sums of products

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

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

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

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

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

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

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

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

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

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

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

5.4 Parametrized types

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

In the last section, we saw that we could define the parametrized type `List` by quantifying over a type. However, in some cases, we will want to be able to inspect or modify the parameters belonging to a type[4]. To represent the parameters of a type, we will need a new gadget.

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

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

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

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

A telescope can either be empty, or be formed from a telescope and a type in the context of that telescope. Here, we used the meaning of a telescope `⟦_⟧tel` to define types in the context of a telescope. This meaning represents the valid assignment of values to parameters:

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

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

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

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

but is not sufficient to give its definition, as we need to be able to bind a value `a` of `A` and reference it in the field `P a`. By quantifying telescopes over a type, we can represent bound arguments using almost the same setup [EC22]:

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

A `Tel P` then represents a telescope for each value of `P`, which we can view as a telescope in the context of `P`. For readability, we redefine values in the context of a telescope as:

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

```
_⊢_ : 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 $P = \top$, we recover the previous definition of parameter-telescopes. We can then define an extension of a telescope as a telescope in the context of a parameter telescope:

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

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

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

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

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

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

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

```
map₂ : ∀ {A B C} → (∀ {a} → B a → C a) → Σ A B → Σ A C
map₂ f (a , b) = (a , f b)

map-var = map₂
```

and interpret this universe as follows:

```
⟦_⟧U-par : U-par Γ → (⟦ Γ ⟧tel tt → Type) → ⟦ Γ ⟧tel tt → Type
⟦_⟧C-par : Con-par Γ V → (⟦ Γ & V ⟧tel → Type) → ⟦ Γ & V ⟧tel → Type

⟦ []    ⟧U-par X p = ⊥
⟦ C :: D ⟧U-par X p = ⟦ C ⟧C-par (X ∘ fst) (p , tt) × ⟦ D ⟧U-par X p

⟦ 𝟙     ⟧C-par X pv = ⊤
⟦ ρ C   ⟧C-par X pv = X pv × ⟦ C ⟧C-par X pv
```

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

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

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

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

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

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

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

5.5 Indexed types

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

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

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

Recall that in native Agda datatypes, a choice of constructor can fix the indices of the recursive fields and the resultant type, so we encode:

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

If we are constructing a term of some indexed type, then the previous choices of constructors and arguments build up the actual index of this term. This actual index must then match the index we expected in the declaration of this term. This means that in the case of a leaf, we have to replace the unit type with the necessary equality between the expected and actual indices:

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

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

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

In a recursive field, the expected index can be chosen based on parameters and variables.

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

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

and

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

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

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

As a bonus, we can also use `U-ix` for generic programming. For example, by a long construction which can be found in Appendix C, we can define the generic `fold` operation[6]:

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

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

Intuitively, `fold` operation works as follows: Suppose the information of one constructor application of `D`, where the recursive positions are valued in `X`, can be

[6]Do note that this version takes a polymorphic function as an argument, as opposed to the usual fold which has the quantifiers on the outside. The usual fold can be recovered by giving `X` an argument to assert that the type variable is actually equal to some concrete type.

collapsed to a value of `X` again. Then, by recursively collapsing the constructors from the bottom up, we can collapse all values of `μ-ix D` to values of `X`.

As a more concrete example, instantiating `fold` to `ListD`, we get (up to some type equivalences):

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

which, much like the familiar foldr operation lets us consume a list to a value `X A`, provided a value `X A` in the empty case, and a means to convert a pair of `A` and `X A` to `X A`.

6 Ornaments

We also skip over field deletion and such because we don't need them, exercise to the reader.

recheck

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

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

! : A → ⊤
! x = tt

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

ListD : Desc (⊘ ▷ const Type) ⊤
ListD = 𝟙 !
      :: σ (λ ((_ , A) , _) → A) (ρ ! (𝟙 !))
      :: []
```

Purely looking at their descriptions, `ℕ` and `List` are rather similar, except that `List` has a parameter and an extra field `ℕ` does not have. We could say that we can form the type of lists by starting from `ℕ` and adding this parameter and field, while keeping everything else the same. In the other direction, we see that each list corresponds to a natural by stripping this information. Likewise, the type of vectors is almost identical to `List`, can be formed from it by adding indices, and each vector corresponds to a list by dropping the indices.

These and similar observations can be generalized using ornaments [McB14; KG16; Sij16], which define a language or binary relation, describing which datatypes can be formed by decorating others. Conceptually, an ornament from a type `A` to a type `B` represents that `B` can be formed from `A` by adding information or making the indices more specific. Consequently, for each ornament

from A to B, we expect to get a function from B to A erasing this information and reverting to less specific indices.

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

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

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

should then come equipped with a function:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

However, since the parameters, indices, and variables need not be identical on both sides (in particular, the variables can diverge even more depending on the preceding ornament), we have to ask that for `𝟙` and `ρ`, these are related by a structure respecting conversion, or more graphically, a commuting square[7]. In fact, we will soon see that these pieces of information are exactly what we need to complete `ornForget`.

The other constructors `Δ` states that we can add fields.

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

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

Using that `ℕ` has no parameters or indices, we see that `List` has more specific parameters, namely a single type parameter, and also no indices. Because of this, all commuting squares factor through the unit type and are hence (fortunately) trivial. This ornament preserves most structure of `ℕ`, only adding a field of the type parameter of `List` using `Δ`.

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

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

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

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

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

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

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

which uses the commutativity squares we required earlier to revert some values (and parameters, indices, and variables) to the unornamented type. For example, in the case of the `𝟙` preserving ornament[8]:

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

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

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

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

We can now make good use of the generic `fold` we defined for `U-ix`!

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

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

7 Ornamental Descriptions

And why we skip ornaments in the next part

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

[8] The other cases can be found in Appendix D.

```
    ornaments can make this kind of usage possible:
    ornamental descriptions.

An ornamental description drops the left-hand side when
    compared to an ornament, and interprets the remaining
    right-hand side as the starting point of the new
    datatype:
\ExecuteMetaData[Ornament/OrnDesc]{ConOrnDesc-type}
The definition of ornamental descriptions can be derived
    in a straightforward manner from ornaments, removing
    all mentions of the LHS and making all fields which
    then no longer appear in the indices explicit\footnote
    {One might expect to need less equalities, alas, this
    is difficult because of \autoref{rem:orn-lift}.}. We
    will show the leaf-preserving rule as an example, the
    others are derived analogously:
\ExecuteMetaData[Ornament/OrnDesc]{OrnDesc-1}
As we can see, the only change we need to make is that \
    AgdaBoundFontStyle{k} becomes explicit and fully
    annotated.

Almost by construction, we have that an ornamental
    description can be decomposed into a description of
    the new datatype
\ExecuteMetaData[Ornament/OrnDesc]{toDesc}
and an ornament between the starting description and this
     new description
\ExecuteMetaData[Ornament/OrnDesc]{toOrn}
```

Part I
Descriptions

If we are going to simplify working with complex sequence types by instantiating generic programs to them, we should first make sure that these types fit into the descriptions. We construct descriptions for nested datatypes by extending the encoding of parametric and indexed datatypes from Subsection 5.5 with three features: information bundles, parameter transformation, and description composition. Also, to make sharing constructors easier, we introduce variable transformations. Transforming variables before they are passed to child descriptions allows both aggressively hiding variables and introducing values as if by let-constructs.

We base the encoding of off existing encodings [Sij16; EC22]. The descriptions take shape as sums of products, enforce indices at leaf nodes, and have

explicit parameter and variable telescopes. Unlike some other encodings [eff20; EC22], we do not allow higher-order inductive arguments. We use `--type-in-type` and `--with-K` to simplify the presentation, noting that these can be eliminated respectively by moving to `Typeω` and by implementing interpretations as datatypes, as described in Appendix A.

8 Numerical Representations

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

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

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

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

We remark that without further assumptions, we cannot use the equality type from Section 4 for this notion of sameness of types: repeating the definition of a type gives two distinct types with no equality between them. Instead, we import another notion of sameness, known as isomorphisms:

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

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

To express such a definition by isomorphism, we define:

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

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

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

using

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

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

To keep the resulting `Iso`s readable, we construct them as chains of smaller `Iso`s using a variant of "equational reasoning" [The23; WKS22], which lets us compose `Iso`s while displaying the intermediate steps. In the calculation of `Fin`, we will use the following lemmas[10]

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

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

Using these, we can calculate

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

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

This `Def` then extracts to a definition of `Fin`

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

To derive `Vec`, we will use the isomorphisms

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

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

[10] Note! `ua`.

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

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

which yields us a definition of vectors

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

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

and the `Iso` to `Lookup` in one go.

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

9 Augmented Descriptions

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

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

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

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

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

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

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

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

which will then ensure that each `𝟙` and `ρ` both are assigned a number `ℕ`, and each `σ` is assigned a function that converts values of the type of its field to `ℕ`[11].

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

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

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

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

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

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

10 The Universe

We also need to take care that the numerical representations we will construct indeed fit in our universe. The final universe `U-ix` of Subsection 5.5, while already quite general, still excludes many interesting datastructures. In particular, the

[11] More about `δ` later.

encoding of parameters forces recursive type occurrences to have the same applied parameters, ruling out nested datatypes such as (binary) random-access lists [HS22; Oka98]: •Redo (or check) the Agda snippets below here.

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

and finger trees [HP06]:

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

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

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

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

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

We will resolve these issues as follows. We can describe nested types by allowing parameters to be transformed before they are passed to recursive fields [**?**]. By transforming variables before they are passed to subsequent fields, it becomes possible to share constructor descriptions[12]. Finally, by adding a variant of σ specialized to descriptions, we can describe composite datatypes more succinctly.

Combining these changes, we define the following universe:

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

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

where the constructors are defined as:

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

[12] Downsides upsides.

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

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

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

From this definition, we can recover the ordinary descriptions as

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

Let us explain this universe by discussing some of the old and new datatypes we can describe using it. Some of these datatypes do not make use of the full generality of this universe, so we define some shorthands to emulate the simpler descriptions. Using

```
σ+ : (S : Γ & V ⊢ Type) → {If .σi S} → ConI If Γ (V ▷ S) J → ConI If Γ V J
σ+ S {if} C = σ S {if = if} id C

σ- : (S : Γ & V ⊢ Type) → {If .σi S} → ConI If Γ V J → ConI If Γ V J
σ- S {if} C = σ S {if = if} fst C
```

(and the analogues for δ) we emulate unbound and bound fields respectively, and with

```
ρ0 : {if : If .ρi} {V : ExTel Γ} → V ⊢ J → ConI If Γ V J → ConI If Γ V J
ρ0 {if = if} r D = ρ {if = if} r id D
```

we emulate an ordinary (as opposed to nested) recursive field. We can then describe ℕ

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

and List as before

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

by replacing σ with σ- and ρ with ρ0.

On the other hand, we bind the length of a vector as a field when defining vectors, so there we use σ+ instead:

```
VecD : Desc (∅ ▷ const Type) ℕ
VecD = 𝟙 (const 0)
```

```
        :: σ- (λ ((_ , A) , _) → A)
        ( σ+ (const ℕ)
        ( ρ0 (λ (_ , (_ , n)) → n)
        ( 𝟙  (λ (_ , (_ , n)) → suc n))))
        :: []
```

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

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

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

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

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

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

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

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

Here, the fact that the first `δ-` drops its field from the telescope makes it possible

to reuse of `Digit` in the second `δ-`.

These descriptions can be instantiated as before by taking the fixpoint

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

of their interpretations as functors

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

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

Also note that this (obviously?) completely ignores the Info of a description.

In this universe, we also need to insert the transformations of parameters `f` in `ρ` and the transformations of variables `h` in `σ` and `δ`.

Like for `U-ix`, we can give the generic `fold` for `DescI`

Part II
Ornaments

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

•Somewhat final version above, draft/notes/rough comments/outline below.

In this section, we will describe the ornamental descriptions for the `DescI` universe, and explain their working by means of (plenty of examples). We omit the definition of the ornaments, since we will only construct new datatypes, rather than relate pre-existing types[13].

do we need to remark more?

11 Ornamental descriptions

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

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

Recall that a `OrnDesc If′ Δ c J i D` represents the ornament building on top of `D`, which yields a description with information `If′`, parameters `Δ`, and indices `J`. We use `~` to write down pointwise equality of functions, which in this case are all commutativity squares. Since `ConI` allows the transformation of variable telescopes, we have to dedicate a lot of lines to writing down commutativity squares for variables, which along with the generally high number of arguments and implicits[14] makes the definition rather dry and long.

One tip is to ignore all squares involving a `Vxf`, these are trivial when using the `+-` variants of the `σ` and `δ` formers anyway! Due to the last constructor `δ•`, `OrnDesc`, `ConOrnDesc`, and `toDesc`[15] become tightly connected, so the definition is given in one large mutual block:

```
data OrnDesc    {If} (If′ : Info) (Δ : Tel τ)
                (c : Cxf Δ Γ) (J : Type) (i : J → I)
                : DescI If Γ I → Type where
  []   : OrnDesc If′ Δ c J i []
  _::_ : ConOrnDesc If′ {c = c} id i {If = If} CD
       → OrnDesc If′ Δ c J i D
       → OrnDesc If′ Δ c J i (CD :: D)

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

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

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

  δ : (R : DescI If″ Θ J) (j : Γ & V ⊢ J) (t : Γ & V ⊢ ⟦ Θ ⟧tel tt)
```

[14] Of which even more are hidden!

[15] We left out the variable square for `δ•`, because it is honestly just too long. If this was included, then we also would involve `ornForget`.

```
    {g : Vxf Γ _ V'} (h : Vxf Δ _ W')
    {v' : VxfO c W' V'}
  → (∀ {p} → g ∘ VxfO-▷ v (liftM2 (μ R) t j) ~ v' {p = p} ∘ h)
  → {if : If .δi Θ J} {iff : InfoF If'' If}
    {if' : If' .δi Θ J} {iff' : InfoF If'' If'}
  → ConOrnDesc If' v' i CD
  → ConOrnDesc If' v i (δ {If} {if = if} {iff = iff} j t R g CD)

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

Δδ : (R : DescI If'' Θ J) (j : W ⊢ J) (t : W ⊢ ⟦ Θ ⟧tel tt)
     (h : Vxf Δ (W ▷ liftM2 (μ R) t j) W')
     {v' : VxfO c W' V}
   → (∀ {p} → v ∘ fst ~ v' {p = p} ∘ h)
   → {if' : If' .δi Θ J} {iff' : InfoF If'' If'}
   → ConOrnDesc If' v' i CD
   → ConOrnDesc If' v i CD

•δ : {R : DescI If'' Θ K} {c' : Cxf Λ Θ} {k' : M → K} {k : V ⊢ K}
     {fΘ : V ⊢ ⟦ Θ ⟧tel tt} {g : Vxf _ (V ▷ liftM2 (μ R) fΘ k) V'}
     (m : W ⊢ M) (fΛ : W ⊢ ⟦ Λ ⟧tel tt)
   → (RR' : OrnDesc If''' Λ c' M k' R)
     (h : Vxf _ (W ▷ liftM2 (μ (toDesc RR')) fΛ m) W')
     {v' : VxfO c W' V'}
   → (p₁ : ∀ q w → c' (fΛ (q , w)) ≡ fΘ (c q , v w))
   → (p₂ : ∀ q w → k' (m (q , w)) ≡ k (c q , v w))
   → ∀ {if} {iff} {if' : If' .δi Λ M} {iff' : InfoF If''' If'}
   → (DE : ConOrnDesc If' v' i CD)
   → ConOrnDesc If' v i (δ {If} {if = if} {iff = iff} k fΘ R g CD)
```

Here the implicit `If'` contain the information necessary to recover the `DescI` from an `OrnDesc`:

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

toCon  : {c : Cxf Δ Γ} {v : VxfO c W V} {i : J → I} {D : ConI If Γ V I}
       → ConOrnDesc If' v i D → ConI If' Δ W J
toCon (𝟙 j' x {if' = if}) = 𝟙 {if = if} j'
toCon (ρ j' h x x₁ {if' = if} CO) = ρ {if = if} j' h (toCon CO)
toCon {v = v} (σ S h v' x {if' = if} CO) = σ (S ∘ over v) {if = if} h (toCon CO)
toCon {v = v} (δ R j t h x {if' = if} {iff' = iff} CO) = δ {if = if} {iff = iff} (j ∘ over v) (t ∘ over v
```

```
toCon (Δσ S h v′ x {if′ = if} CO) = σ S {if = if} h (toCon CO)
toCon (Δδ R j t h x {if′ = if} {iff′ = iff} CO) = δ {if = if} {iff = iff} j t R h (toCon CO)
toCon (•δ m fΛ RR′ h p₁ p₂ {if′ = if} {iff′ = iff} CO) = δ {if = if} {iff = iff} m fΛ (toDesc RR′) h (to
```

The commutativity squares again ensure the existence of functions like `ornForget`, and that these ornamental descriptions indeed induce ornaments.

Compared to the previous ornaments, we have the new constructors `δ`, `Δδ` and `δ•`, where the first two are analogues of `σ` and `Δσ`. The `δ•` constructor states that an ornamental description from a description `R` and a (constructor) ornamental description from `CD` can be composed to form an ornamental description from the composition (in the sense of the `δ` type-former) of `CD` with `R`.

Let us make the uses of `OrnDesc` more clear by means of examples, where we make use of the simpler variants:

Oδ• +- needs ornForget to run

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

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

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

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

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

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

and give the random access lists from before as an ornamental description as well.

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

```
  :: OΔσ- (λ ((_ , A) , _) → A)
  ( OΔσ- (λ ((_ , A) , _) → A)
  ( ρ _ (λ (_ , A) → (_ , Pair A)) (const refl) (const refl)
  ( 𝟙 _ (const refl))))
  :: []
```

Likewise, we can use `δ•` to start from the "fingertree numbers": and compose this with the ornamental description of `Digit` to obtain the ornamental description of finger trees:

finger tree skeleton

DigitOD

FingerTreeOD

Now we can compute everything generically.

```
\ExecuteMetaData[Ornament/OrnDesc]{ConOrnDesc-type}
The definition of ornamental descriptions can be derived
    in a straightforward manner from ornaments, removing
    all mentions of the LHS and making all fields which
    then no longer appear in the indices explicit\footnote
    {One might expect to need less equalities, alas, this
    is difficult because of \autoref{rem:orn-lift}.}. We
    will show the leaf-preserving rule as an example, the
    others are derived analogously:
\ExecuteMetaData[Ornament/OrnDesc]{OrnDesc-1}
As we can see, the only change we need to make is that \
    AgdaBoundFontStyle{k} becomes explicit and fully
    annotated.

Almost by construction, we have that an ornamental
    description can be decomposed into a description of
    the new datatype
\ExecuteMetaData[Ornament/OrnDesc]{toDesc}
and an ornament between the starting description and this
     new description
\ExecuteMetaData[Ornament/OrnDesc]{toOrn}

\section{The ornaments}
we could ditch removal of fields: we don't use it.
    downside: ornament over ornament is the same as field
    removal for deltas

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

\todo{Nuke ornaments, keep ornamental descriptions}

\towrite{Put something that isn't yet in \autoref{ssec:bg
    -orn} here.}

\ExecuteMetaData[Ornament/Orn]{Orn-type}
\ExecuteMetaData[Ornament/Orn]{ornForget-type}
```

```
We will walk through the constructor ornaments
\ExecuteMetaData[Ornament/Orn]{ConOrn-type}
again, an ornament between datatypes is just a list of
    ornaments between their constructors
\ExecuteMetaData[Ornament/Orn]{Orn}
Note that all ornaments completely ignore information
    bundles! They cannot affect the existence of \
    AgdaFunction{ornForget} after all.

Copying parts from one description to another, up to
    parameter and index refinement, corresponds to
    reflexivity. Preservation of leaves follows the rule
\ExecuteMetaData[Ornament/Orn]{Orn-1}
We can see that this commuting square (\texttt{e (k p) ≡
    j (over f p)}) is necessary: take a value of \texttt{E
    } at \texttt{p, i}, where \texttt{i} is given as \
    texttt{k p}. Then \AgdaFunction{ornForget} has to
    convert this to a value of \texttt{D} at \texttt{f p ,
     e i}, but since \texttt{e i} must match \texttt{j (f
    p)}, this is only possible if \texttt{e (k p) = j (f p
    )}.

Preserving a recursive field similarly requires a square
    of indices and conversions to commute
\ExecuteMetaData[Ornament/Orn]{Orn-rho}
additionally requiring the recursive parameters to
    commute with the conversion. \todo{Does adding the
    derivations for the squares everywhere make this
    section clearer?}

Preservation of non-recursive fields and description
    fields is analogous
\ExecuteMetaData[Ornament/Orn]{Orn-sigma-delta}
differing only in that non-recursive fields appears
    transformed on the right hand, while description
    fields have their conversions modified instead. For
    this rule, we need that the variable transformations
    fit into a commuting square with the parameter
    conversions. The condition on indices for descriptions
    , which is a commuting triangle, is encoded in the
    return type\footnote{Should this become a problem like
     with \AgdaInductiveConstructorρ{}, modifying the rule
     to require a triangle is trivial.}.

Ornaments would not be very interesting if they only
```

related identical structures. The left-hand side can also have more fields than the right-hand side, in which case \AgdaFunction{ornForget} will simply drop the fields which have no counterpart on the right-hand side. As a consequence, the description extending rules have fewer conditions than the description preserving rules:
\ExecuteMetaData[Ornament/Orn]{Orn-+-rho}
Note that this extension\footnote{Kind of breaking the ``ornaments relate types with similar recursive structure'' interpretation.} with a recursive field has no conditions.

Extending by a non-recursive field or a description field again only requires the variable transform to interact well with the parameter conversion
\ExecuteMetaData[Ornament/Orn]{Orn-+-sigma-delta}

In the other direction, the left-hand side can also omit a field which appears on the right-hand side, provided we can produce a default value
\ExecuteMetaData[Ornament/Orn]{Orn--sigma-delta}
These rules let us describe the basic set of ornaments between datatypes.

Intuitively we also expect a conversion to exist when two constructors have description fields which are not equal, but are only related by an ornament. Such a composition of ornaments takes two ornaments, one between the field, and one between the outer descriptions. This composition rule reads:\todo{The implicits kind of get out of control here, but the rule is also unreadable without them. I might hide the rule altogether and only run an example with it.}
\ExecuteMetaData[Ornament/Orn]{Orn-comp}
We first require two commuting squares, one relating the parameters of the fields to the inner and outer parameter conversions, and one relating the indices of the fields to the inner index conversion and the outer parameter conversion. Then, the last square has a rather complicated equation, which merely states that the variable transforms for the remainder respect the outer parameter conversion.

We will construct \AgdaFunction{ornForget} as a \AgdaFunction{fold}. Using

```
\ExecuteMetaData[Ornament/Orn]{erase-type}
we can define the algebra which forgets the added
    structure of the outer layer
\ExecuteMetaData[Ornament/Orn]{ornAlg}
Folding over this algebra gives the wanted function
\ExecuteMetaData[Ornament/Orn]{ornForget}

\todo{NatD was removed here}

We can also relate lists and vectors
\ExecuteMetaData[Ornament/Orn]{ListD-VecD}
Now the parameter conversion is the identity, since both
    have a single type parameter. The index conversion is
    \AgdaFunction{!}, since lists have no indices. Again,
    most structure is preserved, we only note that vectors
     have an added field carrying the length.

Instantiating \AgdaFunction{ornForget} to these ornaments
    , we now get the functions \AgdaFunction{length} and \
    AgdaFunction{toList} for free!

%\investigate{Having a function of the same type as \
    AgdaFunction{ornForget} is not sufficient to deduce an
     ornament. An obstacle is that the usual empty type (
    no constructors) and the non-wellfounded empty type (
    only a recursive field) don't have an ornament. Also,
    while the leaf-preservation case spells itself out,
    the substitutions obviously don't give us a way to
    recover the equalities.}
```

Part III
Numerical representations

```
3 Calculating datastructures using Ornaments

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

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

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

12 Numerical representations as ornaments

Reflecting on this derivation for `ℕ`, we could perform the same computation for `Leibniz` to get Braun trees. However, we note that these computations proceed with roughly the same pattern: each constructor of the numeral system gets assigned a value, and is amended with a field holding a number of elements and subnodes using this value as a "weight". This kind of "modifying constructors" is formalized by ornamentation [KG16], which lets us formulate what it means for two types to have a "similar" recursive structure. This is achieved by interpreting (indexed inductive) datatypes from descriptions, between which an ornament is seen as a certificate of similarity, describing which fields or indices need to be introduced or dropped to go from one description to the other. *Ornamental descriptions*, which act as one-sided ornaments, let us describe new datatypes by recording the modifications to an existing description.

Put some minimal definitions here.

Looking back at `Vec`, ornaments let us show that express that `Vec` can be formed by introducing indices and adding a fields holding an elements to `ℕ`. However, deriving `List` from `ℕ` generalizes to `Leibniz` with less notational overhead, so we tackle that case first. We use the following description of `ℕ`

```
NatD : Desc ⊤ ℓ-zero
NatD _ = σ Bool λ
  { false → ▿ []
  ; true → ▿ [ tt ] }
```

Here, `σ` adds a field to the description, upon which the rest of the description can vary, and `▿` lists the recursive fields and their indices (which can only be `tt`). We can now write down the ornament which adds fields to the `suc` constructor

```
NatD-ListO : Type → OrnDesc ⊤ ! NatD
NatD-ListO A (ok _) = σ Bool λ
  { false → ▿ _
  ; true → Δ A (λ _ → ▿ (ok _ , _)) }
```

Here, the `σ` and `▿` are forced to match those of `NatD`, but the `Δ` adds a new field. Using the least fixpoint and description extraction, we can then define `List` from this ornamental description. Note that we cannot hope to give an unindexed ornament from `Leibniz`

```
LeibnizD : Desc ⊤ ℓ-zero
LeibnizD _ = σ (Fin 3) λ
  { zero       → ▿ []
  ; (suc zero) → ▿ [ tt ]
  ; (suc (suc zero)) → ▿ [ tt ] }
```

into trees, since trees have a very different recursive structure! Thus, we must keep track at what level we are in the tree so that we can ask for adequately many elements:

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

Two : Type → Type
Two X = X × X

LeibnizD-TreeO : Type → OrnDesc ℕ ! LeibnizD
LeibnizD-TreeO A (ok n) = σ (Fin 3) λ
  { zero       → ɣ _
  ; (suc zero) → Δ (power n Two A) λ _ → ɣ (ok (suc n) , _)
  ; (suc (suc zero)) → Δ (power (suc n) Two A) λ _ → ɣ (ok (suc n) , _) }
```

We use the `power` combinator to ensure that the digit at position n, which has weight 2^n in the interpretation of a binary number, also holds its value times 2^n elements. This makes sure that the number of elements in the tree shaped after a given binary number also is the value of that binary number.

13 Generic numerical representations

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

We first define the kind of information constituting a type of "natural numbers"

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

which gets its semantics from the conversion to `ℕ`

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

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

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

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

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

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

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

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

Hence, a number can have a list of alternatives, which can be one of

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

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

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

```
NatND : DescI Number ⊘ τ
NatND = 𝟙 {if = 0} _
      :: ρ0 {if = 1} _ (𝟙 {if = 1} _)
      :: []
```

In such a calculation, all we really needed was a translation between the type of numbers, and a type of shapes. This encoding precisely captures all information we need to form such a type of shapes.

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

We could directly compute indexed array, using the index for the proof of representability, and from it the correctness of numbers of elements. However, we give the unindexed array first: we can get the indexed variant for free [McB14]!

no, rewrite this

Conjecture 13.1. *We claim then that the description given by*

```
TrieO : (D : DescI Number ⊘ τ) → OrnDesc Plain (⊘ ▷ const Type) ! τ ! D
```

and the number of elements coincides with the underlying number, as given by `ornForget`.

Currently, without proof

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

The hard work of `TrieO` is done by

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

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

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

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

When we encounter a recursive field

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

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

The case for a non-recursive field is similar

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

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

Consider the case of a description field[17]

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

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

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

Part IV
Related work

Part V
Discussion

References

[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.

[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.

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

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

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

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

[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.

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

Part VI
Appendix

:warning: The other way of parameters–indices

A Without K but with universe hierarchies

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

B Big sigma

C gfold

D ornForget

E Finger trees

F Heterogenization

G More equivalences for less effort

Noting that constructing equivalences directly or from isomorphisms as in **??** can quickly become challenging when one of the sides is complicated, we work out a different approach making use of the initial semantics of W-types instead. We claim that the functions in the isomorphism of **??** were partially forced, but this fact was unused there.

First, we explain that if we assume that one of the two sides of the equivalence is a fixpoint or initial algebra of a polynomial functor (that is, the `μ` of a `Desc′`), this simplifies giving an equivalence to showing that the other side is also initial.

We describe how we altered the original ornaments [KG16] to ensure that `μ` remains initial for its base functor in Cubical Agda, explaining why this fails otherwise, and how defining base functors as datatypes avoids this issue.

In a subsection focussing on the categorical point of view, we show how we can describe initial algebras (and truncate the appropriate parts) in such a way that the construction both applies to general types (rather than only sets), and still produces an equivalence at the end. We explain how this definition, like the usual definition, makes sure that a pair of initial objects always induces a pair of conversion functions, which automatically become inverses. Finally, we explain that we can escape our earlier truncation by appealing to the fact that "being an equivalence" is a proposition.

Next, we describe some theory, using which other types can be shown to be initial for a given algebra. This is compared to the construction in **??**, observing that intuitively, initiality follows because the interpretation of the zero constructor is forced by the square defining algebra maps, and the other values are forced by repeatedly applying similar squares. This is clarified as an instance of recursion over a polynomial functor.

To characterize when this recursion is allowed, we define accessibility with respect to polynomial functors as a mutually recursive datatype as follows. This datatype is constructed using the fibers of the algebra map, defining accessibility of elements of these fibers by cases over the description of the algebra. Then we remark that this construction is an atypical instance of well-founded recursion, and define a type as well-founded for an algebra when all its elements are accessible.

We interpret well-foundedness as an upper bound on the size of a type, leading us to claim that injectivity of the algebra map gives a lower bound, which is sufficient to induce the isomorphism. We sketch the proof of the theorem, relating part of this construction to similar concepts in the formalization of well-founded recursion in the Standard Library. In particular, we prove an irrelevance and an unfolding lemma, which lets us show that the map into any other algebra induced by recursion is indeed an algebra map. By showing that

it is also unique, we conclude initiality, and get the isomorphism as a corollary.

The theorem is applied and demonstrated to the example of binary naturals. We remark that the construction of well-foundedness looks similar to view-patterns. After this, we conclude that this example takes more lines that the direct derivation in **??**, but we argue that most of this code can likely be automated.

Merge