Utrecht University
Master Thesis Computing Science

Generic programming with ornaments and dependent types

Yorick Sijsling

Supervisors
dr. Wouter Swierstra
prof. dr. Johan Jeuring

June 29, 2016

Abstract

Modern dependently typed functional programming languages like Agda allow very specific restrictions to be built into datatypes by using indices and dependent types. Properly restricted types can help programmers to write correct-by-construction software. However, code duplication will occur because the language does not recognise that similarly-structured datatypes with slightly different program-specific restrictions can be related. Some functions will be copy-pasted for lists, vectors, sorted lists and bounded lists.

Ornaments specify the exact relation between of different datatypes and may be a path towards a solution. It is a first step in structuring the design space of datatypes in dependently typed languages. Literature has shown how ornaments can produce conversion functions between types, and how they can help to recover code reuse by transporting functions across ornaments.

This thesis presents an Agda library for experimentation with ornaments. We have implemented a generic programming framework where datatypes are represented as descriptions. A description can be generated from a real datatype and patched with an ornament to create new description, which in turn can be converted back to a new datatype.

Our descriptions are carefully designed to always be convertible to actual datatypes, resulting in an unconventional design. They pass along a context internally to support dependent types and they can be used with multiple parameters and multiple indices.

Contents

1 Introduction **3**

2 Usage **7**

3 Generics and ornaments **11**
- 3.1 Descriptions . . . 12
- 3.2 Maps and folds . . . 15
- 3.3 Ornaments . . . 16
- 3.4 Ornamental algebras . . . 18
- 3.5 Discussion . . . 19
  - 3.5.1 Σ-descriptions . . . 20
  - 3.5.2 Finding the right ornaments . . . 23

4 Ornaments on dependently typed descriptions **25**
- 4.1 Contexts and environments . . . 26
- 4.2 Descriptions . . . 27
- 4.3 Ornaments . . . 29

5 Ornaments on families of datatypes **33**
- 5.1 Descriptions . . . 33
- 5.2 Ornaments . . . 38
- 5.3 Algebraic ornaments . . . 42
- 5.4 Discussion . . . 44
  - 5.4.1 Separating parameters from contexts . . . 46

6 Generic programming with descriptions **49**
- 6.1 Descriptions and ornaments . . . 50
- 6.2 Quoting datatypes . . . 51
- 6.3 Deriving an embedding-projection pair . . . 53
- 6.4 Generic functions . . . 54
- 6.5 Unquoting descriptions . . . 56
- 6.6 Higher-level ornaments . . . 57
  - 6.6.1 Structure-preserving ornaments . . . 58
  - 6.6.2 Ornament composition . . . 58
  - 6.6.3 More ornaments . . . 59
  - 6.6.4 Reornaments . . . 60
- 6.7 Discussion . . . 62
  - 6.7.1 Embedding-projection instances . . . 62

7 Discussion **65**
7.1 Explicit parameter use 65
7.2 Induction-recursion and strict positivity 68

8 Conclusion **70**
8.1 Future work 71

Chapter 1

Introduction

One of the strong points of functional programming languages like Haskell and Agda is that they allow very precise types. The type List A not only tells us that it is a list, but also that every element in the list is of type A. This is already a lot more static information than untyped or dynamically typed programming languages, and allows programmers to adopt an 'if it type checks it works' mentality. But how precise should we make our types? Consider the take function, which takes a number of elements from the front of a list:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
take : ∀ {A} → (n : Nat) → List A → List A
take zero _      = []
take (suc n) [] = ?1
take (suc n) (x :: xs) = x :: take n xs
```

What needs to be done when the list is too short? One option is to return a default value, like the empty list [], but such behavior may hide bugs which would be discovered otherwise. Another solution is to change the return type to Maybe (List A), so the nothing value can be returned. This makes the call site responsible for error handling. An entirely different approach is to avoid the situation by restricting the types appropriately. Agda supports inductive families [10], so a length index could be added to the List type—resulting in the following Vec datatype:

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

A new take function for Vec can be defined, which only accepts lists of at least length n. Under these circumstances the problematic clause simply disappears:

```
takeᵥ : ∀ {A m} → (n : Nat) → Vec A (n + m) → Vec A n
takeᵥ zero _ = []
takeᵥ (suc n) (x :: xs) = x :: takeᵥ n xs
```

By adding the proper indices, properties of the data can be encoded within a datatype. One could build trees that are always sorted, trees bounded by a minimum and

maximum value, red-black trees or trees that are always balanced. Building datatypes such that they precisely match the required properties of the data is an essential aspect of writing correct-by-construction programs in functional programming languages. This specialization of datatypes can however be an obstacle to code reuse. For example; one may have defined a function to find the first value with the property P in a list of natural numbers:

```
find : List Nat → (P : Nat → Bool) → Maybe Nat
find [] P = nothing
find (x :: xs) P = if (P x) then (just x) else (find xs P)
```

Although we have explained Vecs as being a List with a length index, it is *defined* as an entirely separate thing. Agda has no idea that these two are related. To search in a Vec of naturals, we have to define an entirely new function:

```
find_v : ∀ {n} → Vec Nat n → (P : Nat → Bool) → Maybe Nat
find_v [] P = nothing
find_v (x :: xs) P = if (P x) then (just x) else (find_v xs P)
```

The same problem will occur again and again if one uses other list-like types. We can define bounded lists that are parameterised by a maximum value, or sorted lists that are indexed by the lowest value in the list (so at each _::_ a proof can be included that the element is at least as low as the lowest value in the tail). The implementation of find for both of these can be copy-pasted:

```
find_b : ∀ {mx} → BoundedNatList mx → (P : Nat → Bool) → Maybe Nat
find_b [] P = nothing
find_b (x :: xs) P = if (P x) then (just x) else (find_b xs P)
find_s : ∀ {l} → SortedNatList l → (P : Nat → Bool) → Maybe Nat
find_s [] P = nothing
find_s (x :: xs) P = if (P x) then (just x) else (find_s xs P)
```

Maybe it would be useful if Agda knew about the relations between all these different variants of datatypes. Conor McBride [19] has presented *ornaments* as a way to express relations between datatypes. Loosely speaking, one type can be said to be an ornament of another if it contains more information in some way, for example by the refinement with indices or the addition of data. They can be used to express that Vec is an ornament of List, or that List can be defined as an ornament on Nat by attaching an element of type A to each suc constructor. Before we can start working with ornaments we need a way to model datatypes within Agda itself.

Datatype-generic programming talks about types by using *descriptions*. These descriptions of types can take the form of a description datatype. This is combined with a decoding function which assigns a type to every inhabitant of the description datatype [1]. The descriptions and decoding function together form a *universe*[17] of descriptions. To give an example; the following Desc datatype can describe the unit type, pairs and products:

```
data Desc : Set where
  ‘1 : Desc
  _⊕_ : Desc → Desc → Desc
  _⊗_ : Desc → Desc → Desc
```

These descriptions can be used to describe, for instance, booleans as ‘1 ⊕ ‘1. Descriptions are decoded to types using the $⟦_⟧_{desc}$ function:

```
⟦_⟧desc : Desc → Set
⟦ ‘1 ⟧desc = ⊤
⟦ A ⊕ B ⟧desc = Either ⟦ A ⟧desc ⟦ B ⟧desc
⟦ A ⊗ B ⟧desc = ⟦ A ⟧desc × ⟦ B ⟧desc
```

When the decoding of a description D produces a type which is isomorphic to a type X, we can say that D describes X. Indeed we see that $⟦$ ‘1 ⊕ ‘1 $⟧_{desc}$ normalises to Either ⊤ ⊤, a type which is isomorphic to the type for booleans. Generic programming frameworks like Haskell's generic deriving mechanism [15] automatically derive an *embedding-projection* pair [21] that converts between elements of the decoded description and elements of the real datatype. An embedding-projection pair for booleans can be defined as follows:

```
bool-to : Bool → ⟦ ‘1 ⊕ ‘1 ⟧desc
bool-to false = left tt
bool-to true = right tt

bool-from : ⟦ ‘1 ⊕ ‘1 ⟧desc → Bool
bool-from (left tt) = false
bool-from (right tt) = true
```

By choosing more advanced descriptions, more of Agda's datatypes can be described. One may add support for inductive datatypes (such as natural numbers and lists), for datatype parameters, or for indices. Descriptions can be used as a foundation to define ornaments. An ornament is written as a patch for an existing description, and can be applied to get a new description. In this way, the ornament expresses the relation between the original description and the ornamented description.

When ornaments are used to compute new descriptions, it would also be convenient if new datatypes could be generated from computed descriptions. Most generic programming frameworks do not require this feature, because they never make modifications to descriptions. The availability of this feature puts some unique constraints on the definition of descriptions, because every description must be convertible to a real datatype. We have to be careful that the sums and products in our descriptions can never occur in the wrong places—For instance, a description (‘1 ⊕ ‘1) ⊗ (‘1 ⊕ ‘1) describes a pair of booleans, but is not a sum-of-products and can therefore not be written as just one datatype. At least two datatypes (the product type and Bool for example) have to be used to get a type that is isomorphic to $⟦$ (‘1 ⊕ ‘1) ⊗ (‘1 ⊕ ‘1) $⟧_{desc}$.

Agda provides a *reflection* mechanism which can be leveraged to build the generic deriving and declaring framework. With reflection, existing datatype declarations can be inspected and descriptions can be generated for them. The functions constituting the embedding-projection pair can be generated as well. The current version of Agda (2.5.1) does not yet allow the declaration of new datatypes, but we can do this semi-automatically by generating the types for the individual constructors.

In this thesis we combine reflection, generics and ornaments to build a library with which we can perform operations on user defined Agda datatypes. This thesis makes the following contributions:

1. A universe of descriptions is built to encode datatypes. The descriptions support dependent types (chapter 4) by passing along a context within the constructors. Multiple parameters and multiple indices can be encoded (chapter 5) and

the names of arguments can be stored (chapter 6). The descriptions are structured such that they are guaranteed to be convertable to Agda datatypes, so modifications to descriptions can be made freely without having to worry whether the resulting description makes sense or not.

2. Ornaments are defined for each version of these descriptions (section 3.3, 4.3, 5.2 and 6.1). The ornaments allow insertion and deletion of arguments, refinement of parameters and refinement of indices. Many ornament-related concepts are translated to our universe, including ornamental algebras, algebraic ornaments (section 5.3) and reornaments (section 6.6.4). Some high-level operations are defined which can be used to modify descriptions without having deep knowledge of our implementation (section 6.6).

3. We implement a framework which uses reflection to derive descriptions and their embedding-projection pairs for real datatypes (chapter 6). Some operations like fold and depth are defined generically, to work on every datatype for which a description has been derived (section 6.4). Ornaments can be applied to descriptions, and these descriptions can be used to semi-automatically declare the corresponding datatype (section 6.5).

With these contributions we hope to provide a framework which can be used for experimentation with ornaments, as well as a practical example of how ornaments can be integrated into a language. Along the way a library for the reflection of datatypes has been built which, to our knowledge, does not yet exist for Agda.

All the code in this thesis is developed in Agda 2.5.1 and all is based on Ulf Norell's Agda Prelude (the prelude is better for general purpose and meta programming, while the standard library is more focussed on proving things). If we assume that the reader knows about basic definitions like `_x_`, `Either` or `Σ`, the prelude probably defines it. Every chapter will be treated as a module that is internally consistent without overlapping names. Some definitions will be shared between chapters and some will be redefined implicitly, but if they are explicitly defined they will not cause conflict with other explicit definitions within the chapter. All the source code is available at `http://sijsling.com/`.

Chapter 2

Usage

In this section we provide a short overview of how the generic programming and ornamentation library works. It is meant to show how the different components fit together, so not all implementation details will be presented here. We focus on how an end-user with minimal knowledge about ornaments or generics would use our library. The interested reader is asked to suspend their curiosity—the rest of this thesis explains how the library is implemented.

To start with, we apply the deriveHasDesc function to the Nat datatype. This performs all kinds of meta-programming magic to introduce two new definitions in the current scope: quotedNat and NatHasDesc.

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
unquoteDecl quotedNat NatHasDesc =
  deriveHasDesc quotedNat NatHasDesc (quote Nat)
```

From our newly obtained quotedNat we can retrieve a *description* which is the representation of a datatype declaration within the system. The description is of type Desc ε ε _; the arguments ε and ε indicate that this datatype has no parameters and no indices. For now we will not be looking at the description itself.

```
natDesc : Desc ε ε _
natDesc = QuotedDesc.desc quotedNat
```

The deriveHasDesc function has also defined NatHasDesc for us. This is a record instance which contains an embedding-projection pair. The embedding-projection pair translates between values of the types Nat and μ natDesc tt tt, where μ natDesc tt tt is the Set which contains the elements as described by natDesc. The record is found automatically with instance search, so once deriveHasDesc has been called the embedding-projection can be used by simply writing to or from:

```
natTo : Nat → μ natDesc tt tt
natTo = to
natFrom : μ natDesc tt tt → Nat
natFrom = from
```

Datatypes can have parameters. A simple example of a datatype with parameters is List. We can use the same deriveHasDesc function to define quotedList and ListHasDesc automatically. When we retrieve the description of the datatype we see that it is of type Desc ε (ε ▷′ Set) _. The (ε ▷′ Set) here tells us that there is one parameter of type Set.

```
data List (A : Set) : Set where
  nil : List A
  cons : (x : A) → (xs : List A) → List A
unquoteDecl quotedList ListHasDesc =
  deriveHasDesc quotedList ListHasDesc (quote List)
listDesc : Desc ε (ε ▷′ Set) _
listDesc = QuotedDesc.desc quotedList
```

Both List and μ listDesc are polymorphic in the type of their elements, so the to and from functions are now polymorphic as well:

```
listTo : ∀ {A} → List A → μ listDesc (tt , A) tt
listTo = to
listFrom : ∀ {A} → μ listDesc (tt , A) tt → List A
listFrom = from
```

With a HasDesc instance we can perform generic operations. For example the function gdepth of type ∀ {A} → ⦃ R : HasDesc A ⦄ → A → Nat) which calculates the depth of any value that has a generic representation. For the Nat type this is just the identity, but for List this is exactly the length of a list:

```
nat-id : Nat → Nat
nat-id = gdepth
length : ∀ {A} → List A → Nat
length = gdepth
```

The length of a list can also be calculated using a fold and an algebra. Actually, that is precisely what gdepth does internally. It uses an algebra depthAlg listDesc and folds it over the list. One may define an alternative length function as follows:

```
length′ : ∀ {A} → List A → Nat
length′ = gfold (depthAlg listDesc)
```

A depth algebra can be calculated for any description—this allows gdepth to be fully generic (i.e. it works for all descriptions). One may also define algebras for a specific type, for instance a countBoolsAlg which counts the number of trues in a List Bool. The generic gfold function can be used to fold this algebra.

```
countBools : List Bool → Nat
countBools = gfold countBoolsAlg
```

We have taken a look at naturals and lists. These datatypes are similar in their recursive structure and we want to exploit that. We can create an ornament which can be used as a patch on the description of naturals to get the description of lists. Descriptions do not include names of the datatype and constructors, but they do include names of arguments so we have to do two things to obtain lists from naturals:

- The recursive argument of suc must be renamed to "xs". We can use the expression renameArguments 1 (just "xs" :: []) to build such an ornament.
- A parameter of type Set must be added, which must be used as an argument in the suc/cons constructor. The ornament to do that is addParameterArg 1 "x".

These two ornaments have to be applied in sequence, so they are composed using the ≫⁺ operator. The resulting ornament can be applied to produce a new description using ornToDesc, and we see that ornToDesc nat→list results in a description which is exactly the same as listDesc:

```
nat→list : Orn _ _ _ _ natDesc
nat→list = renameArguments 1 (just "xs" :: [])
  ≫⁺ addParameterArg 1 "x"
test-nat→list : ornToDesc nat→list ≡ listDesc
test-nat→list = refl
```

Datatype indices can be used to *refine* datatypes. Such a refinement can ensure that values can only be built if they adhere to a certain invariant. For instance, a length index can be added to lists to ensure that only lists of the specified length are allowed. One class of ornaments that inserts indices is that of *algebraic ornaments*. These use an algebra on the original datatype to calculate the values of the indices. By folding the algebra depthAlg listDesc we were able to calculate the length of a list, but we can also use it with algOrn to build an ornament which inserts a length index:

```
list→vec : Orn _ _ _ _ listDesc
list→vec = algOrn (depthAlg listDesc)
```

As expected, this ornament results in a description with an index of type Nat. The list of indices is now (ε ▷′ Nat), and the list of parameters is still (ε ▷′ Set).

```
vecDesc : Desc (ε ▷′ Nat) (ε ▷′ Set) _
vecDesc = ornToDesc list→vec
```

We have built a new description using ornamentation, but it does not yet have a corresponding Agda datatype. Our descriptions are defined in such a way that they can always be converted back to a real datatype definition. The reflection mechanism in Agda does not yet support the definition of datatypes, but we *can* calculate the types of every constructor and of the datatype itself. All we have to do is write the skeleton of the datatype definition, but not the types themselves. Using deriveHasDescExisting we can derive VecHasDesc which connects the datatype Vec to the existing description vecDesc, so the to and from functions go between Vec A n and μ vecDesc (tt , A) (tt , n).

```
data Vec (A : Set) : unquoteDat vecDesc where
  nil : unquoteCon vecDesc 0 Vec
  cons : unquoteCon vecDesc 1 Vec
unquoteDecl quotedVec VecHasDesc =
  deriveHasDescExisting quotedVec VecHasDesc (quote Vec) vecDesc
```

An essential property of ornaments is that each element of the ornamented type can be transformed back to an element of the original type. The generic operation gforget does that for a given ornament. We can use it to define the function which transforms a Vec to the corresponding List:

```
vecToList : ∀ { A n } → Vec A n → List A
vecToList = gforget list→vec
```

We have seen how this implementation can be used to perform generic operations and to build and use ornaments on a high level with a fairly limited amount of knowledge. We did not once have to look at the actual descriptions and ornaments which are used internally. In the rest of this thesis we will be taking a better look on how these descriptions and ornaments have to be defined and how meta-programming can be used to connect the descriptions to actual datatypes.

Chapter 3

Generics and ornaments

Datatype-generic programming in functional languages is all about figuring out how to build complicated types from a small set of components. For finite types—those types which have a finite number of inhabitants, i.e. they do not allow recursion—this is quite easy. By using just the basic components ⊤, _×_ and Either for unit, products and coproducts, we can already build some simple types like Bool and Maybe. The Maybe type has a type parameter which needs to be instantiated to get an inhabited type.

```
boolLike : Set
boolLike = Either ⊤ ⊤
maybeLike : (A : Set) → Set
maybeLike A = Either A ⊤
```

Of course, finite types are very limited. Not just in the number of elements contained in these types, but in their utility too. To implement a wider range of types we need recursion. Naively, we might try to define lists as listLike A = Either ⊤ (listLike A), but the evaluation of a term like that does not terminate. The common approach to work around this problem is by associating a *pattern functor* with every datatype [14, 22]. Instead of building a recursive expression of type Set, we build a functor of type Set → Set. For example, we can build these pattern functors for naturals and lists:

```
natPF : Set → Set
natPF X = Either ⊤ X
listPF : (A : Set) → Set → Set
listPF A X = Either ⊤ (A × X)
```

Each occurence of X signifies that it is a recursive position. The definitions natPF and listPF provide the structure—or *pattern*—for the types we want, and the values of the Xes are of later concern. By taking the fixpoint of the pattern functor we let the argument X refer to the type itself, effectively representing induction. The fixpoint is closed in the μ′ datatype[1].

```
data μ′ (F : Set → Set) : Set where
  ⟨_⟩ : F (μ′ F) → μ′ F
```

[1]The observant reader may notice that the μ′ datatype does not pass the strict-positivity check. This problem is solved with a new definition of μ′ in the next section.

Now whenever a μ′ somePF is expected, we can provide a somePF (μ′ somePF) wrapped in ⟨_⟩. The recursive positions within that somePF (μ′ somePF) are expected to contain a μ′ somePF again, closing the loop neatly.

```
listPF-example : μ′ (listPF String)
listPF-example = ⟨ right ("one" , ⟨ right ("two" , ⟨ left tt ⟩) ⟩) ⟩
```

We have seen how simple algebraic datatypes can be built using a few basic components, namely the unit type, products, coproducts and fixpoints. To reason about these types we have to formalise the fact that only these components, and no others, can be used to form our types. In section 3.1 we reify these components to build a *universe of descriptions*. In general, the concept of a universe in Martin Löf's type theory [17] involves two things: firstly there are codes which describe types; secondly there is a decoding function to translate codes into the type they represent. In this work, the descriptions form the codes of the universe and the decoding function ⟦_⟧ gives a pattern functor for a description. In a sense, the decoding function provides a pattern functor semantics for descriptions. By taking the fixpoint of the resulting pattern functor we obtain a Set which can be used as a type.

With descriptions and their interpretations in place, ornaments for these descriptions are defined in section 3.3. Every ornament is built for a specific description, and can represent the copying, insertion and removal of parts of the description. If something is to be called an ornament, it must be possible to produce a forget function for every ornament. The forget function goes from elements of the ornamented type to elements of the original type. In section 3.4 an *ornamental algebra* is defined which gives rise to the forget function.

3.1 Descriptions

As promised, we will build a universe of descriptions. For the descriptions in this chapter, we will be using the following codes:

- The _⊕_ operator represents a choice. For our purposes this always means a choice between different constructors. A list of constructors is separated by _⊕_'es and terminated with the empty type ‘0.
- The _⊗_ operator is used for products. A chain of _⊗_'s terminated with the unit type ι can be formed to represent the arguments of a constructor.
- For recursive arguments a special operator rec-⊗_ can be used in the same places where _⊗_ is allowed.

These codes are formalised using the ConDesc and DatDesc datatypes, defined in listing 3.1. ConDesc contains the constructors ι, _⊗_ and rec-⊗_; these are sufficient to describe the types of constructors. DatDesc is basically a Vec of ConDescs; it is indexed by the number of constructors and uses ‘0 and _⊕_ to chain the ConDescs together. The reasons for this split between ConDescs and DatDescs are discussed at the end of this chapter.

With this set of components we can build some simple datatypes. To get some feeling for this, we look at the descriptions for unit, naturals, non-dependent pairs and lists. Note that _⊗_ and rec-⊗_ take precedence over _⊕_, so products are applied before sums.

```
data ConDesc : Set₁ where
  ι : ConDesc
  _⊗_ : (S : Set) → (xs : ConDesc) → ConDesc
  rec-⊗_ : (xs : ConDesc) → ConDesc
data DatDesc : Nat → Set₁ where
  ‘0 : DatDesc 0
  _⊕_ : ∀ {#c} (x : ConDesc) (xs : DatDesc #c) → DatDesc (suc #c)
```

Listing 3.1: Sum-of-products descriptions

```
⟦_⟧conDesc : ConDesc → Set → Set
⟦ ι ⟧conDesc X = ⊤
⟦ S ⊗ D ⟧conDesc X = S × ⟦ D ⟧conDesc X
⟦ rec-⊗ D ⟧conDesc X = X × ⟦ D ⟧conDesc X
lookupCtor : ∀ {#c} (D : DatDesc #c) → Fin #c → ConDesc
lookupCtor ‘0 ()
lookupCtor (x ⊕ _) zero = x
lookupCtor (_ ⊕ xs) (suc k) = lookupCtor xs k
⟦_⟧datDesc : ∀ {#c} → DatDesc #c → Set → Set
⟦_⟧datDesc {#c} D X = Σ (Fin #c) λ k → ⟦ lookupCtor D k ⟧conDesc X
-- The notation ⟦_⟧ is overloaded to mean ⟦_⟧datDesc
data μ {#c : Nat} (F : DatDesc #c) : Set where
  ⟨_⟩ : ⟦ F ⟧ (μ F) → μ F
```

Listing 3.2: Sum-of-products semantics

```
unitDesc : DatDesc 1
unitDesc = ι ⊕ ‘0
natDesc : DatDesc 2
natDesc = ι ⊕ rec-⊗ ι ⊕ ‘0
pairDesc : (A B : Set) → DatDesc 1
pairDesc A B = A ⊗ B ⊗ ι ⊕ ‘0
listDesc : (A : Set) → DatDesc 2
listDesc A = ι ⊕ A ⊗ rec-⊗ ι ⊕ ‘0
```

It’s noteworthy that even though we can parameterise these descriptions, the descriptions themselves are not really aware of it. It is merely a *shallow* embedding of parametricity. The extended descriptions in chapter 5 include the parameters within the descriptions, creating a deep embedding of parametric polymorphism.

Listing 3.2 shows how descriptions are decoded to pattern functors. The decoding of ConDesc is fairly straightforward, producing λ X → S × X × ⊤ for S ⊗ rec-⊗ ι. The decoding of DatDesc is somewhat more involved. While the conventional approach would be to convert all _⊕_ constructors to Either and the ‘0 constructor to ⊥, we instead choose to produce a Σ-type: Σ (Fin #c) λ k → ⟦ lookupCtor D k ⟧$_{conDesc}$ X. This type means that the first argument is used to indicate the choice of constructor and the second argument must then be an element of that particular constructor. This prevents long chains of lefts and rights due to nested Eithers. We will write ⟦_⟧ to mean ⟦_⟧$_{conDesc}$ or ⟦_⟧$_{datDesc}$ when that is not ambiguous in the context..

The fixpoint μ (Listing 3.2) is similar to the fixpoint in the introduction of this chapter, but specialised to the decoding of DatDesc. This specialisation is necessary to convince Agda that the datatype μ is strictly positive. This works as long as there are only strictly-positive occurences of X in $[\![_]\!]_{\text{datDesc}}$ and $[\![_]\!]_{\text{conDesc}}$. Since μ already includes the call to $[\![_]\!]_{\text{datDesc}}$, we can get the Set corresponding to a description by simply prepending the description with μ. For instance, μ natDesc is a Set which corresponds to the natural numbers.

The following code gives an example of how a μ natDesc can be constructed. In nat-example$_1$, the hole has to be of type ⟦ natDesc ⟧ (μ natDesc). When we expand that type we get a Σ-type where the first argument must be a Fin 2, allowing us to pick one of the two constructors.

```
nat-example₁ : μ natDesc
nat-example₁ = ⟨ ?0 ⟩
-- ?0 : ⟦ natDesc ⟧ (μ natDesc)
-- ?0 : Σ (Fin 2) (λ k → ⟦ lookupCtor natDesc k ⟧ (μ natDesc))
```

In nat-example$_2$ we pick the second constructor (the numbering starts at 0), the description of this constructor is rec-⊗ ι, so we are left to fill in a ⟦ rec-⊗ ι ⟧ (μ natDesc), a type which is equal to μ natDesc × ⊤. The definition nat-example$_3$ shows how this process could be completed by filling in ⟨ 0 , tt ⟩ in the recursive spot, resulting in an expression which should be read as suc zero, i.e. the number 1.

```
nat-example₂ : μ natDesc
nat-example₂ = ⟨ 1 , ?1 ⟩
-- ?1 : ⟦ lookupCtor natDesc 1 ⟧ (μ natDesc)
-- ?1 : ⟦ rec-⊗ ι ⟧ (μ natDesc)
-- ?1 : μ natDesc × ⊤

nat-example₃ : μ natDesc
nat-example₃ = ⟨ 1 , ⟨ 0 , tt ⟩ , tt ⟩
```

Whenever we want to give a value belonging to μ someDescription, we start by writing ⟨_⟩ and picking the number of the constructor we want to use. This corresponds with the fact that for some datatype DT, a value of type DT can be created by using one of the constructors of that datatype. The following functions for naturals and lists show how every constructor-call is represented by a particular term ⟨ i , ? ⟩:

```
‘zero : μ natDesc
‘zero = ⟨ 0 , tt ⟩
‘suc : μ natDesc → μ natDesc
‘suc n = ⟨ 1 , n , tt ⟩

‘[] : ∀ {A} → μ (listDesc A)
‘[] = ⟨ 0 , tt ⟩
_‘∷_ : ∀ {A} → A → μ (listDesc A) → μ (listDesc A)
_‘∷_ x xs = ⟨ 1 , x , xs , tt ⟩
```

With these functions, we can write values almost like we would with normal datatypes. They illustrate the similarity between the descriptions and real datatypes:

```
nat-example : ‘suc ‘zero ≡ ⟨ 1 , ⟨ 0 , tt ⟩ , tt ⟩
nat-example = refl

list-example : 7 ‘∷ 8 ‘∷ ‘[] ≡ ⟨ 1 , 7 , ⟨ 1 , 8 , ⟨ 0 , tt ⟩ , tt ⟩ , tt ⟩
list-example = refl
```

```
conDescmap : ∀ {X Y} (f : X → Y) (D : ConDesc) →
  (v : ⟦ D ⟧ X) → ⟦ D ⟧ Y
conDescmap f ι tt = tt
conDescmap f (S ⊗ xs) (s , v) = s , conDescmap f xs v
conDescmap f (rec-⊗ xs) (s , v) = f s , conDescmap f xs v
datDescmap : ∀ {#c X Y} (f : X → Y) (D : DatDesc #c) →
  (v : ⟦ D ⟧ X) → ⟦ D ⟧ Y
datDescmap f xs (k , v) = k , conDescmap f (lookupCtor xs k) v
```

Listing 3.3: Map over the pattern functors

If we want to be absolutely certain that our descriptions match up to the types they represent, we could provide an isomorphism between them. In the case of lists for some given type A, an isomorphism between List A and μ (listDesc A) would entail four functions. The functions to and from convert between the representation using generics and the Agda datatype. Within the context of generic programming the from and to functions are commonly referred to as the *embedding-projection pair*. The functions to-from and from-to provide proofs that to and from are mutual inverses.

```
to : List A → μ (listDesc A)
from : μ (listDesc A) → List A
to-from : ∀ x → from (to x) ≡ x
from-to : ∀ x → to (from x) ≡ x
```

More often than not, we will skip the proofs and just give the embedding-projection pair or the constructor-functions. This already rules out many mistakes and suffices to convince ourselves that a description is *probably* right.

3.2 Maps and folds

In the previous section, we claimed that the decoding of a description results in a so-called pattern functor. Clearly, $⟦_⟧_{datDesc}$ returns something of type Set → Set, but we have not yet shown that it is really a functor. To prove this, we define the functorial map for the decoding of any description in listing 3.3. For a function f : X → Y and a description D, we can always compute a function ⟦ D ⟧ X → ⟦ D ⟧ Y.

A typical operation which can be performed generically is *folding*, defined in listing 3.4. Given a description D and an algebra of type Alg D X, the fold function can calculate a result of type X for any value of type μ D. As seen in listing 3.4, we define Alg D X to be ⟦ D ⟧ X → X. The intuition here is that the user has to provide the fold function with a method to calculate a result for every possible value, given that a result has already been calculated for the recursive positions. The fold function first maps the fold over the recursive positions and then the algebra can be applied.

Example 3.2.1. An example of a simple algebra is one that counts the true values in a list of booleans. To define the algebra we can pattern match on the argument of type ⟦ listDesc Nat ⟧ Nat. A case split is done on the first field in the Σ-type, such that each case corresponds to a constructor of the list datatype. The first case, for the empty list, always returns a 0. In the second case—suc zero , x , xs , tt—the variable x is of type Bool because it is an element in the list. The variable xs is of type Nat because that is

```
Alg : ∀ {#c} → DatDesc #c → Set → Set
Alg D X = ⟦ D ⟧ X → X
fold : ∀ {#c} {D : DatDesc #c} {X} (α : Alg D X) → μ D → X
fold {D = D} α ⟨ xs ⟩ = α (datDescmap (fold α) D xs)
```

Listing 3.4: Generic fold

the result of the algebra. By applying fold to the algebra we obtain a function which counts the number of trues in a list of booleans.

```
countTruesAlg : Alg (listDesc Bool) Nat
countTruesAlg (zero , tt) = 0
countTruesAlg (suc zero , x , xs , tt) = if x then suc xs else xs
countTruesAlg (suc (suc ()) , _)
countTrues : μ (listDesc Bool) → Nat
countTrues = fold countTruesAlg
```

△

3.3 Ornaments

Now that we have a good way to describe some basic datatypes within Agda, we can implement ornaments for those descriptions. Ornaments for such simple datatypes without indices are of limited use, but getting some practice with this basic form now should make things easier when we extend the descriptions with more features. Our ornaments are mostly based on those presented by McBride [19] and later by Dagand and McBride [6]. Our choice of descriptions does require some novel modifications to the original presentation of ornaments.

The characterising feature of ornaments is that elements of an ornamented type are at least as informative as elements of the base type. More formally, a transformation from one description to another is an ornament *iff* it comes with a forget function that takes ornamented values back to their corresponding values of the base type. The next section will show that all the ornaments we define indeed come with a forget function.

The ornaments and their interpretation are defined in listing 3.5. Ornaments for constructors and datatypes are defined separately; ConOrn works on ConDescs and DatOrn works on DatDescs. Both are indexed by their respective descriptions, such that an ornament for a datatype description D has type DatOrn D. The ornaments contain several groups of operations:

- The unit copy operation ι just keeps the ι constructor.
- Argument copy operations: -⊗_ and rec-⊗_ keep the argument, but an ornament has to be given for the tail. The Set for -⊗_ does not have to be given; it is simply copied.
- Argument insertion operations: _+⊗_ and rec-+⊗_ insert a new argument in a constructor.
- The argument deletion operation give-K removes a non-recursive argument from a constructor.

```
data ConOrn : (D : ConDesc) → Set₁ where
  ι : ConOrn ι
  -⊗_ : ∀ {S xs} → (xs⁺ : ConOrn xs) → ConOrn (S ⊗ xs)
  rec-⊗_ : ∀ {xs} → (xs⁺ : ConOrn xs) → ConOrn (rec-⊗ xs)
  _+⊗_ : ∀ {xs} (S : Set) → (xs⁺ : ConOrn xs) → ConOrn xs
  rec-+⊗_ : ∀ {xs} → (xs⁺ : ConOrn xs) → ConOrn xs
  give-K : ∀ {S xs} (s : S) → (xs⁺ : ConOrn xs) → ConOrn (S ⊗ xs)
data DatOrn : ∀ {#c} (D : DatDesc #c) → Set₁ where
  ‘0 : DatOrn ‘0
  _⊕_ : ∀ {#c x xs} →
    (x⁺ : ConOrn x) (xs⁺ : DatOrn xs) → DatOrn {suc #c} (x ⊕ xs)
conOrnToDesc : ∀ {D} → ConOrn D → ConDesc
conOrnToDesc ι = ι
conOrnToDesc (-⊗_ {S = S} xs⁺) = S ⊗ conOrnToDesc xs⁺
conOrnToDesc (rec-⊗ xs⁺) = rec-⊗ conOrnToDesc xs⁺
conOrnToDesc (S +⊗ xs⁺) = S ⊗ conOrnToDesc xs⁺
conOrnToDesc (rec-+⊗ xs⁺) = rec-⊗ conOrnToDesc xs⁺
conOrnToDesc (give-K s xs⁺) = conOrnToDesc xs⁺
ornToDesc : ∀ {#c} {D : DatDesc #c} → DatOrn D → DatDesc #c
ornToDesc ‘0 = ‘0
ornToDesc (x⁺ ⊕ xs⁺) = conOrnToDesc x⁺ ⊕ ornToDesc xs⁺
```

Listing 3.5: Definition of ornaments

- Datatype copy operations ‘0 and _⊕_. Constructors can not be inserted or removed with the chosen ornaments, so the ‘0 and _⊕_ have to be copied. An ornament has to be given for every constructor in the datatype. The choice to disallow insertion and removal of constructors is discussed in section 3.5.

The functions conOrnToDesc and ornToDesc define the semantics of ConOrn and DatOrn respectively. They convert an ornament into a description. If we talk about applying an ornament, we actually mean the application of conOrnToDesc or ornToDesc. Furthermore, we will once again overload the ⟦_⟧ notation such that when it is used with an ornament it means $⟦$ ornToDesc o $⟧_{datDesc}$ or $⟦$ conOrnToDesc o $⟧_{conDesc}$.

We can now build some simple ornaments. We have previously defined natDesc as ι ⊕ rec-⊗ ι ⊕ ‘0 and listDesc A as ι ⊕ A ⊗ rec-⊗ ι ⊕ ‘0. These are clearly quite similar. We can build an ornament on natDesc which extends the second constructor with an argument of type A, using the copy operations and _+⊗_. Interpreting this ornament with ornToDesc gives us exactly the description of lists of A:

```
nat→list : ∀ {A} → DatOrn natDesc
nat→list {A} = ι ⊕ A +⊗ rec-⊗ ι ⊕ ‘0
test-nat→list : ∀ {A} → ornToDesc nat→list ≡ listDesc A
test-nat→list = refl
```

```
conForgetNT : ∀ {D} (o : ConOrn D) →
  ∀ {X} → ⟦ conOrnToDesc o ⟧ X → ⟦ D ⟧ X
conForgetNT ι tt = tt
conForgetNT (-⊗ xs⁺) (s , v) = s , conForgetNT xs⁺ v
conForgetNT (rec-⊗ xs⁺) (s , v) = s , conForgetNT xs⁺ v
conForgetNT (_+⊗_ S xs⁺) (s , v) = conForgetNT xs⁺ v
conForgetNT (rec-+⊗_ xs⁺) (s , v) = conForgetNT xs⁺ v
conForgetNT (give-K s xs⁺) v = s , conForgetNT xs⁺ v
forgetNT : ∀ {#c} {D : DatDesc #c} (o : DatOrn D) →
  ∀ {X} → ⟦ ornToDesc o ⟧ X → ⟦ D ⟧ X
forgetNT ‘0 (() , _)
forgetNT (x⁺ ⊕ xs⁺) (zero , v) = 0 , conForgetNT x⁺ v
forgetNT (x⁺ ⊕ xs⁺) (suc k , v) = (suc *** id) (forgetNT xs⁺ (k , v))
-- Alg (ornToDesc o) (μ D) is ⟦ ornToDesc o ⟧ (μ D) → μ D
forgetAlg : ∀ {#c} {D : DatDesc #c} (o : DatOrn D) →
  Alg (ornToDesc o) (μ D)
forgetAlg o = ⟨_⟩ ∘ forgetNT o
forget : ∀ {#c} {D : DatDesc #c} (o : DatOrn D) →
  μ (ornToDesc o) → μ D
forget o = fold (forgetAlg o)
```

Listing 3.6: Ornamental algebras

3.4 Ornamental algebras

Each ornament induces an *ornamental algebra* [19]. The ornamental algebra turns elements of the ornamented type back into elements of the original type, such that folding the ornamental algebra for an ornament (o : DatOrn D) results in a function from μ (ornToDesc o) to μ D. The ornamental algebra is built using a natural transformation between the pattern functors ⟦ o ⟧ and ⟦ D ⟧, that is a function which goes from ⟦ o ⟧ X to ⟦ D ⟧ X for all values of X.

```
forgetNT : ∀ {#c} {D : DatDesc #c} (o : DatOrn D) →
  ∀ {X} → ⟦ o ⟧ X → ⟦ D ⟧ X
forget : ∀ {#c} {D : DatDesc #c} (o : DatOrn D) →
  μ (ornToDesc o) → μ D
```

The forget function is implemented using forgetNT. In other words; when we can transform one pattern functor into another pattern functor, we can make that same transformation between the fixed points of those pattern functors. The full definition is given in listing 3.6. Note that we use the function _***_, which is defined as the bimap on pairs such that (f *** g) (x , y) is (f x , g y). The actual ornamental *algebra* forgetAlg arises as an intermediary step between forgetNT and forget.

Example 3.4.1. Let us take a look at the ornamental algebra for the nat→list ornament. The forget function for this ornament should take a list to a natural. More precisely, applying forget to nat→list for a given parameter A results in a function which takes a μ (listDesc A) to a μ natDesc. Each nil is taken to a zero and cons is taken to a suc—the extra elements of type A which were introduced by the ornament are forgotten. This

happens to result in exactly the length of the list, so we might define a length function as forget nat→list.

```
'length : ∀ { A } → μ (listDesc A) → μ natDesc
'length = forget nat→list

test-length : 'length ("one" ':: "two" ':: '[]) ≡ 'suc ('suc 'zero)
test-length = refl
```

△

Example 3.4.2. The give-K ornament is useful if one wishes to specialise a datatype, instantiating some argument to a particular value. For instance, is we *know* that all the elements in a list of naturals are always 7, we might as well remove that element altogether. If we choose to remove it, we must still remember that the value was 7 for every element. Coincidentally, this ornament results in the same description as that for natural numbers.

```
list→listof7s : DatOrn (listDesc Nat)
list→listof7s = ι ⊕ give-K 7 (rec-⊗ ι) ⊕ '0

test-list→listof7s : ornToDesc list→listof7s ≡ natDesc
test-list→listof7s = refl
```

It seems odd that we can have ornaments which go from naturals to lists, and ornaments from lists to naturals as well. The point here is that within the context of the list→listof7s ornament that natural has a very particular meaning, namely the length of the list. This becomes obvious when we forget the ornament. By passing the number two, we get a list of length two where the values for the elements are provided by the ornament itself.

```
forget-listof7s : forget list→listof7s ('suc ('suc 'zero)) ≡ (7 ':: 7 ':: '[])
forget-listof7s = refl
```

So in fact, we are replicating 7s here. We can generalise the ornament a bit to get a function which repeats a given element a given number of times:

```
'replicate : ∀ { A } → A → μ natDesc → μ (listDesc A)
'replicate x = forget (ι ⊕ give-K x (rec-⊗ ι) ⊕ '0)
```

Interestingly, the 'length function which was obtained by the ornamental algebra of nat→list is the inverse of this 'replicate function which we got with the ornamental algebra of list→listof7s. This is not a coincidence. The nat→list ornament (ι ⊕ A +⊗ rec-⊗ ι ⊕ '0) inserts exactly the argument which was removed by the list→listof7s ornament (ι ⊕ give-K 7 (rec-⊗ ι) ⊕ '0), while keeping the rest of the description the same. Say that o_2 is an inverse ornament of o_1 iff forget o_2 is the inverse of forget o_1, then we could say that nat→list is the inverse ornament of list→listof7s. △

3.5 Discussion

In this chapter we have presented a universe of descriptions for simple datatypes. At the root they are ordinary sum-of-products descriptions which support recursion using fixpoints. Although we did represent types with datatype parameters, the parameter

```
data Code : Set where
  ι : Code
  rec : Code
  _⊕_ : (F G : Code) → Code
  _⊗_ : (F G : Code) → Code
```

Listing 3.7: Codes for a universe of regular types

abstractions were always done externally and the descriptions are not aware of any differences between datatype parameters and other arguments.

The Regular [21] library is a datatype-generic programming library for Haskell which has a similar scope, where parameters and indices are not supported. An Agda formalisation of the Regular library is presented by Magalhães and Löh [16]. The codes for the universe they use are shown in listing 3.7. There are codes for units, recursive positions, sums and products. The decoding function ⟦_⟧ and the fixpoint datatype μ are similar to those in this chapter.

The types that can be represented with the descriptions in this chapter are limited to the *regular* types, those which can be defined using the units, sums, products and the fixpoints μ [21]. Regular types do not allow nested datatypes [3] or mutual recursion. A regular type does not necessarily correspond to one single datatype though. For instance, to write the regular type μ X. ι ⊕ (ι ⊕ ι) ⊗ X in Agda one would need at least two types: List and Bool (it is a list of booleans).

With regards to the ultimate goal of using our descriptions to accurately represent and even define Agda datatypes, we need to impose some restrictions on our descriptions which libraries like Regular do not need. We have to make sure that every description corresponds to exactly one Agda datatype. An Agda datatype is always a sum of products, where each term of the top-level sum corresponds to a constructor and the factors of those terms correspond to constructor arguments. The split between ConDesc and DatDesc enforces this structure.

Our descriptions also differ from those in listing 3.7 in that ours have a list-like structure where ι and _‘0_ function as nil and _⊗_, rec-⊗_ and _⊕_ as cons. This has two benefits: It ensures that every description has one canonical representation and it is easier to work with, both in construction and consumption.

3.5.1 Σ-descriptions

The descriptions we have seen all have sums and products using _⊕_ and _⊗_. In dependently typed languages we have Σ-types, which can be used to encode both sums and products [4]. Some of the work on ornaments which we will be referring to uses descriptions with Σ's, so we will take a look at them. To start with, we define a universe of descriptions and their decoding in listing 3.8.

The σ description is used to describe Σ-types, the rest should be familiar (it is the same as our descriptions). The following description of the Either type illustrates quite well how a σ can mean different things. The Either type has two constructors; the choice between them is made by asking for a Fin 2 in the outer σ, the pattern-matching lambda then picks the description of a constructor based on the Fin 2 value. The top-level σ thus works as a sum of two constructors. The inner σ's function like _⊗_; in the first constructor an A is asked for, in addition to the rest of the description for that constructor where the value (a : A) may be used.

```
data DescΣ : Set₁ where
  ι : DescΣ
  σ : (S : Set) → (xs : S → DescΣ) → DescΣ
  rec-×_ : (xs : DescΣ) → DescΣ
⟦_⟧DescΣ : DescΣ → Set → Set
⟦ ι ⟧DescΣ X = ⊤
⟦ σ S xs ⟧DescΣ X = Σ S λ s → ⟦ xs s ⟧DescΣ X
⟦ rec-× xs ⟧DescΣ X = X × ⟦ xs ⟧DescΣ X
data μΣ (D : DescΣ) : Set where
  ⟨_⟩ : ⟦ D ⟧DescΣ (μΣ D) → μΣ D
```

Listing 3.8: Universe of Σ-descriptions

D : ConDesc/DatDesc	DΣ : DescΣ	⟦ D ⟧ and ⟦ DΣ ⟧
ι	ι	⊤
S ⊗ xs	σ S λ _ → xs	S × ⟦ xs ⟧ X
rec-⊗ xs	rec-× xs	X × ⟦ xs ⟧ X
x_0 ⊕ x_1 ⊕ ⋯ ⊕ x_{n-1} ⊕ ‘0	σ (Fin n) λ k → x_k	Σ (Fin n) λ k → ⟦ x_k ⟧ X

Table 3.1: Descriptions and their DescΣ counterparts

```
eitherDescΣ : (A B : Set) → DescΣ
eitherDescΣ A B = σ (Fin 2) λ
  { zero → σ A λ a → ι
  ; (suc zero) → σ B λ b → ι
  ; (suc (suc ())) }
eitherDescΣ-left : ∀ {A B} → A → μΣ (eitherDescΣ A B)
eitherDescΣ-left x = ⟨ 0 , x , tt ⟩
eitherDescΣ-right : ∀ {A B} → B → μΣ (eitherDescΣ A B)
eitherDescΣ-right x = ⟨ 1 , x , tt ⟩
```

The types which are encoded by our universe are a subset of those which can be encoded by Σ-descriptions. Table 3.1 shows how a ConDesc or DatDesc can be translated into a DescΣ with an equivalent semantics. Note how DatDesc needs multiple constructors to encode a sum where DescΣ uses just one σ. That is why we need the lookupCtor function to define the decoding and DescΣ does not.

In listing 3.9 we define ornaments for copying of ι, σ and rec-×_ and for insertion and deletion of σ's. They are similar to those defined by Dagand and McBride [6]. The insert-σ and delete-σ ornaments match our _+⊗_ and give-K ornaments.

As a quick example, we can now define descriptions of naturals and the ornamentation from naturals to lists. The ornament inserts a new σ in the description, and results in a description which can describe lists.

```
data OrnΣ : (D : DescΣ) → Set₁ where
  ι : OrnΣ ι
  σ : (S : Set) → ∀ { xs } (xs⁺ : (s : S) → OrnΣ (xs s)) → OrnΣ (σ S xs)
  rec-×_ : ∀ { xs } (xs⁺ : OrnΣ xs) → OrnΣ (rec-× xs)
  insert-σ : (S : Set) → ∀ { xs } → (xs⁺ : S → OrnΣ xs) → OrnΣ xs
  delete-σ : ∀ { S xs } → (s : S) → (xs⁺ : OrnΣ (xs s)) → OrnΣ (σ S xs)
ornToDescΣ : ∀ { D } → OrnΣ D → DescΣ
ornToDescΣ ι = ι
ornToDescΣ (σ S xs⁺) = σ S (λ s → ornToDescΣ (xs⁺ s))
ornToDescΣ (rec-× xs⁺) = rec-× (ornToDescΣ xs⁺)
ornToDescΣ (insert-σ S xs⁺) = σ S (λ s → ornToDescΣ (xs⁺ s))
ornToDescΣ (delete-σ s xs⁺) = ornToDescΣ xs⁺
```

Listing 3.9: Ornaments on Σ-descriptions

```
natDescΣ : DescΣ
natDescΣ = σ (Fin 2) λ
  { zero → ι
  ; (suc zero) → rec-× ι
  ; (suc (suc ())) }
nat→listΣ : (A : Set) → OrnΣ natDescΣ
nat→listΣ A = σ (Fin 2) λ
  { zero → ι
  ; (suc zero) → insert-σ A λ a → rec-× ι
  ; (suc (suc ())) }
```

Σ-descriptions are a way to describe sums of products using a very small number of components. All the types which can be encoded in our universe or in the regular types universe of listing 3.7 can be encoded with Σ-descriptions too. Additionally, because the tail description xs of a σ is a function of type S → DescΣ the full computational power of functions can be used. This results in the ability to encode rather exotic types. For instance a type which takes a number n and then n boolean values:

```
boolsDescΣ : DescΣ
boolsDescΣ = σ Nat rest
  where rest : Nat → DescΣ
    rest zero = ι
    rest (suc n) = σ Bool λ _ → rest n
boolsDescΣ-example : μΣ boolsDescΣ
boolsDescΣ-example = ⟨ 3 , true , false , true , tt ⟩
```

We can not use Σ-descriptions for our purposes, because they allow many types which do not correspond to Agda datatypes. However, their simpler semantics does provide a good vantage point to consider the more theoretical aspects of ornaments. While working with complicated descriptions like ours, it is often enlightening to take a step back and consider what things would look like with Σ-descriptions.

o : Con/DatOrn D	oΣ : OrnΣ DΣ	Original PF ⟦ D ⟧ and ⟦ DΣ ⟧	Ornamented PF ⟦ o ⟧ and ⟦ oΣ ⟧
ι	ι	⊤	⊤
- ⊗ xs⁺	σ S λ _ → xs⁺	S × ⟦ xs ⟧ X	S × ⟦ xs⁺ ⟧ X
S +⊗ xs⁺	insert-σ S λ _ → xs⁺	⟦ xs ⟧ X	S × ⟦ xs⁺ ⟧ X
give-K s xs⁺	delete-σ s xs⁺	S × ⟦ xs ⟧ X	⟦ xs⁺ ⟧ X
rec-⊗ xs⁺	rec-× xs⁺	X × ⟦ xs ⟧ X	X × ⟦ xs⁺ ⟧ X
rec-+⊗ xs⁺	-	⟦ xs ⟧ X	X × ⟦ xs⁺ ⟧ X
(give-rec?)	-	X × ⟦ xs⁺ ⟧ X	⟦ xs⁺ ⟧ X
$x_0{}^+$ ⊕ $x_1{}^+$ ⊕ ⋯ ⊕ ‘0	σ (Fin n) λ k → $x_k{}^+$	Σ (Fin n) λ k → ⟦ x_k ⟧ X	Σ (Fin n) λ k → ⟦ $x_k{}^+$ ⟧ X

Table 3.2: Ornaments and their OrnΣ counterparts

3.5.2 Finding the right ornaments

The ornaments in this chapter where presented without much justification, but there are in fact some choices to make here. By defining the forget function we have shown that these ornaments can really be called ornaments. But we can not show that all interesting ornaments are included.

The ornaments we use, defined in listing 3.5, are mostly adapted from the ornaments for Σ-descriptions in listing 3.9. Table 3.2 shows how they relate to each other. The exact meaning of each row is as follows: Given a description D and a Σ-description DΣ which both decode to *Original PF*, the application of o to D and oΣ to DΣ both result in descriptions which decode to *Ornamented PF*. We already knew that our descriptions and their corresponding Σ-descriptions had the same underlying pattern functors (Table 3.1), and now it turns out that the ornaments on our descriptions and the corresponding ornaments on Σ-descriptions perform the same operation on the underlying pattern functors as well. The overloaded notation ⟦_⟧ is used to mean ⟦ ornToDesc o ⟧ and ⟦ ornToDesc_Σ o ⟧ as well.

With the exception of rec-+⊗_, every ornament of ours has a OrnΣ counterpart. The rec-+⊗_ ornament is included because an ornamental algebra can be defined for it, and it does not cause any problems anywhere. One would then also expect an ornament which deletes recursive arguments—similar to give-K, the ornament would require a default value to be able reconstruct the right value in the ornamental algebra. The type of this value is however not known within the ornament declaration so we can not define it as far as we are aware.

```
-- Constructor for ConOrn
  give-rec : ∀ { xs } → ? → (xs⁺ : ConOrn xs) → ConOrn (rec-⊗ xs)
```

There are still some ornaments missing which we did have for Σ-descriptions. Because a chain of constructors x_0 ⊕ x_1 ⊕ ⋯ ⊕ ‘0 is similar to σ (Fin n) λ k → x_k, we would expect the ornaments insert-σ and delete-σ for constructors to have a counterpart in our ornaments. The main reason why we do not have those is because an equivalent ornament would have to insert or delete the whole constructor chain. The deletion of the chain would mean one ends up with a single constructor and the insertion would require a single constructor to start with. Our ornaments have to go from DatDesc to DatDesc, so these operations are both not valid.

Dagand and McBride [6] do use insert-σ to insert a constructor choice quite often though, while still keeping descriptions which make sense. The trick is to always let an

insert-σ be followed by delete-σ's. For instance, we can define the swapnatΣ ornament which swaps the constructors of natDesc_Σ. The insert-σ (Fin 2) says there are going to be two constructors and for each constructor we have to provide an ornament on the original natDesc_Σ. In the first constructor, delete-σ 1 means that we choose the second constructor of natDesc_Σ; the rec-× ι does nothing but copying.

```
swapnatΣ  :  OrnΣ natDescΣ
swapnatΣ  =  insert-σ (Fin 2) λ
   { zero → delete-σ 1 (rec-× ι)
   ; (suc zero) → delete-σ 0 ι
   ; (suc (suc ())) }
```

An insert-σ (Fin n) on the top-level is fine if the first thing we do for each constructor is to pick one of the constructors of the original type using delete-σ. We *can* implement a recons ornament for DatDesc which emulates this behavior. It requires the same information as the insert-σ with delete-σ's requires. The first thing we need is $\#c^+$; the number of constructors we want the ornamented type to have (while #c is the number of constructors of the original type). For each of the new constructors, i.e. for each possible value of a Fin $\#c^+$, two things have to be provided: A value k of type Fin #c to pick a constructor of the original type and an ornament on that constructor, together these emulate a delete-σ k xs^+. The recons constructor for DatOrn looks as follows:

```
recons  :
   ∀ #c⁺ { #c } { D  :  DatDesc #c } →
   (f  :  (k⁺  :  Fin #c⁺) → (Σ (Fin #c) λ k → ConOrn (lookupCtor D k))) →
   DatOrn { #c } D
```

With it, the swapnat ornament can be defined for our universe of descriptions. By comparing swapnat with swapnatΣ we see that, as expected, the user still has to provide all of the same information with the exception of the insert-σ and delete-σ's. That is,

```
swapnat  :  DatOrn natDesc
swapnat  =  recons 2 λ
   { zero → 1 , rec-⊗ ι
   ; (suc zero) → 0 , ι
   ; (suc (suc ())) }
```

The recons ornament is feasible to implement and makes sense in practice, as indicated by the use of the pattern of insert-σ and delete-σ's by Dagand and McBride. We do not implement them for our descriptions in the following chapters for entirely pragmatic reasons. The implementation is hard to get right, even for this simple universe. It also complicates other parts of the code because we can not assume that an ornament keeps the same number of constructors.

Chapter 4

Ornaments on dependently typed descriptions

The sum-of-products descriptions of chapter 3 can be extended to support dependent types. In the _⊗_ constructor we used a Set to indicate which type that argument has. To encode dependent types, we want to allow this type to depend on values and types in the context. Let us first establish some terminology:

- The term *context* will be used to indicate what variables are available and which types they have. Within the List datatype for example (as defined in chapter 2), the context consists of at least the type parameter A of type Set. In the second argument of the cons constructor, the variable x of type A is also in the context, though it is not used. If cons had more arguments after that, (xs : List A) would be in the context as well. In this thesis, contexts are usually indicated by a Γ, with Δ as an alternative. Contexts have the type Cx, and are defined in section 4.1.
- An *environment* is a specific instantiation of a context, containing inhabitants of the types which were indicated by the context. Environments are written as γ of type ⟦ Γ ⟧ or δ of type ⟦ Δ ⟧. The meaning of their types is explained in section 4.1 as well.

In a description, the types of arguments were specified with a Set. Arguments with dependent types are encoded as a function from an environment (γ : ⟦ Γ ⟧) to a Set is used. To maintain the old behavior, an argument can simply ignore the environment. With the definitions in the upcoming sections, the description of lists will be written as follows:

```
listDesc : (A : Set) → DatDesc 2
listDesc A = ι ⊕ (λ γ → A) ⊗ rec-⊗ ι ⊕ ‘0
```

A typical use case for dependent types is in the usage of predicates. For instance, if the IsLessThan7 predicate states that a given number is lower than 7, the type Lt7 contains a natural which is lower than seven:

```
IsLessThan7 : Nat → Set
IsLessThan7 n = n < 7

data Lt7 : Set where
  lt7 : (n : Nat) → IsLessThan7 n → Lt7
```

The constructor of Lt7 uses the value of the first argument to determine the type of the second argument. This can be encoded as a description, where top γ is used to refer to the only value in the environment γ.

```
lt7Desc : DatDesc 1
lt7Desc = (λ γ → Nat) ⊗ (λ γ → IsLessThan7 (top γ)) ⊗ ι ⊕ ‘0
```

More often than not, we will be writing the arguments in point-free style if we can. In the definition of lt7Desc, the functions const and _∘_ can be used to get rid of a lot of parentheses.

```
lt7Desc′ : DatDesc 1
lt7Desc′ = const Nat ⊗ IsLessThan7 ∘ top ⊗ ι ⊕ ‘0
```

Of course, an environment can contain more than one value. The environment is basically a stack of values (more precisely, a snoc-list), where pop and top can be used to refer to a value in the context, in the style of DeBruijn indices [8]. So top γ means variable 0, top (pop γ) means variable 1, top (pop (pop γ)) means variable 2 and so forth.

In the following example we assume that a predicate IsShorter of type List A → Nat → Set exists which says that some list is shorter than some length. A datatype Shorter can be defined which contains a list, a length, and a proof that the list is shorter than that length:

```
IsShorter : {A : Set} → List A → Nat → Set
IsShorter = ⋯
data Shorter (A : Set) : Set where
  shorter : (xs : List A) → (n : Nat) → IsShorter xs n → Shorter A
```

The description shorterDesc describes the Shorter datatype. In the third argument of the constructor, top (pop γ) is used to refer to the list and top γ refers to the natural.

```
shorterDesc : ∀ {A} → DatDesc 1
shorterDesc {A} = const (List A) ⊗ const Nat ⊗
  (λ γ → IsShorter (top (pop γ)) (top γ)) ⊗ ι ⊕ ‘0
```

Note that the third argument of the constructor can not be written point-free with just _∘_ and const. It *is* possible with an S-combinator, as McBride [18] demonstrates. Applicative functors are a generalisation of SKI combinators [20], so one might even choose to write that argument of shorterDesc in applicative style as IsShorter <$> top ∘ pop <*> top. While that style works well for expressions like these, it quickly breaks down for more complicated ones.

In the next section, we will start by showing how environments are exactly implemented. Descriptions will be revised to support the propagation of environments in section 4.2. When descriptions support dependent types, ornaments must do so as well—they will be revised in section 4.3.

4.1 Contexts and environments

An environment γ must contain a stack of values, but what is the type of γ? The type has to mention the types of all the variables and each of those types should be able to use the

```
record _▶_ {a b} (A : Set a) (B : A → Set b) : Set (a ⊔ b) where
  constructor _,_
  field
    pop : A
    top : B pop
```

Listing 4.1: Definition of _▶_

```
mutual
  data Cx : Set₁ where
    _▷_ : (Γ : Cx) (S : (γ : ⟦ Γ ⟧Cx) → Set) → Cx
    ε : Cx
  ⟦_⟧Cx : Cx → Set
  ⟦ Γ ▷ S ⟧Cx = ⟦ Γ ⟧Cx ▶ S
  ⟦ ε ⟧Cx = ⊤′
_▷′_ : (Γ : Cx) → Set → Cx
Γ ▷′ S = Γ ▷ const S
```

Listing 4.2: Cx definition and semantics

values of the previous variables. For the purpose of building types of environments we define _▶_, which is a left-associative version of Σ where fst is renamed to pop and snd to top (Listing 4.1). The unit type ⊤′ can be used as the empty environment, and types are added to the right of it by using _▶_. In each type, an environment γ containing values for all variables to the left of it can be used. For example, if we want to write the type of an environment containing the variables (xs : List A), (n : Nat) and (p : IsShorter xs n), we could write it like this:

```
ShorterEnv : {A : Set} → Set
ShorterEnv {A} = ⊤′ ▶ const (List A) ▶ const Nat ▶
  (λ γ → IsShorter (top (pop γ)) (top γ))
```

The basic types _▶_ and ⊤′ can contain an environment, but they can not be used for pattern matching. There is no way to inspect a value of type Set to see if it is a _▶_ or ⊤′. For this purpose a universe of contexts Cx is built. The Cx decodes to _▶_'s and ⊤′'s. The definition is given in listing 4.2. This is quite a common approach to encode contexts [7, 18]. While we are at it, we also define _▷′_ as a shortcut when a type does not need to use the environment. With these definitions we can create a context which, when decoded, is equal to the ShorterEnv type we defined before.

```
ShorterCx : {A : Set} → Cx
ShorterCx {A} = ε ▷′ List A ▷′ Nat ▷ (λ γ → IsShorter (top (pop γ)) (top γ))
test-ShorterCx : ∀ {A} → ⟦ ShorterCx {A} ⟧ ≡ ShorterEnv {A}
test-ShorterCx = refl
```

4.2 Descriptions

For now we will be assuming that all DatDescs are closed, i.e. they do not refer to free variables. The ConDescs which are directly contained within the DatDesc have to be

```
data ConDesc (Γ : Cx) : Set₁ where
  ι : ConDesc Γ
  _⊗_ : (S : (γ : ⟦ Γ ⟧) → Set) → (xs : ConDesc (Γ ▷ S)) → ConDesc Γ
  rec-⊗_ : (xs : ConDesc Γ) → ConDesc Γ
data DatDesc : Nat → Set₁ where
  ‘0 : DatDesc 0
  _⊕_ : ∀ {#c} (x : ConDesc ε) (xs : DatDesc #c) → DatDesc (suc #c)
```

Listing 4.3: Descriptions with contexts

```
⟦_⟧conDesc : ∀ {Γ} → ConDesc Γ → ⟦ Γ ⟧ → Set → Set
⟦ ι ⟧conDesc γ X = ⊤
⟦ S ⊗ xs ⟧conDesc γ X = Σ (S γ) λ s → ⟦ xs ⟧conDesc (γ , s) X
⟦ rec-⊗ xs ⟧conDesc γ X = X × ⟦ xs ⟧conDesc γ X
```

Listing 4.4: Semantics of ConDesc with contexts

closed too, so a DatDesc is essentially a list of closed ConDescs. Not all ConDescs have to be closed though, because within a constructor new types are added to the context. The context is chained through from left to right and whenever a _⊗_ operator is encountered, the specified type is added to the context of the ConDesc which forms the tail.

In listing 4.3 we see how this works. The DatDesc datatype specifies that each constructor starts with an empty context ε. In the type of _⊗_ we see that a S of type ⟦ Γ ⟧ → Set must be given. The value of S specifies a type which can depend on the current context Γ. The context Γ is extended with S, and this forms the context for the ConDesc in the remainder of the ornament xs.

Ideally, we would also add recursive arguments to the context, but this is fundamentally impossible with our current implementation. This problem will be discussed in section 7.2.

The semantics of ConDesc now requires an environment before a pattern functor can be delivered. The new semantics is given in listing 4.4. For the _⊗_ constructor, the environment is applied to S to obtain the definitive type of that argument. The semantics of DatDesc is only changed slightly to pass the empty environment tt to $⟦_⟧_{conDesc}$.

Example 4.2.1. We can now describe all the types from the introduction of this chapter. To gain some insight in how the contexts are propagated and extended we will also give a step-by-step example of how dependent pairs (the Σ type) are described. We start by specifying the type, which is parameterised by A and B, as the Σ type always is:

```
pairDesc : (A : Set) (B : A → Set) → DatDesc 1
```

By using Agda's refine command, the _⊕_ and ε are automatically filled in. In the remaining hole, a closed ConDesc is expected.

```
pairDesc₁ A B = ?0 ⊕ ‘0
-- ?0 : ConDesc ε
```

When we add an argument of type A with _⊗_, the context of the rest of the constructor is extended with the type A. Remember that ε ▷′ A is defined as ε ▷ const A.

```
pairDesc₂ A B = const A ⊗ ?1 ⊕ ‘0
-- ?1 : ConDesc (ε ▷′ A)
```

We refine the hole with the _⊗_ constructor and ι to finish the list of arguments. Leaving us with the hole for the type of the second argument. The required type tells us that the local context is ε ▷′ A. When the semantics is expanded we get the corresponding type ⊤′ ▶ const A, which is the type of the environment which contains the value of the first argument.

```
pairDesc₃ A B = const A ⊗ ?2 ⊗ ι ⊕ ‘0
-- ?2 : ⟦ ε ▷′ A ⟧ → Set
-- ?2 : ⊤′ ▶ const A → Set
```

Finally, we give B ∘ top as the implementation of the hole, resulting in a description of dependent pairs.

```
pairDesc A B = const A ⊗ B ∘ top ⊗ ι ⊕ ‘0
```

According to section 3.1, an isomorphism between Σ A B and μ (pairDesc A B) should be given to be certain that this is the right description. Doing that is straightforward, so we will only show that the definition is not *entirely* wrong by giving one half of the embedding-projection pair (one of the four functions in the isomorphism).

```
pair-to : {A : Set} {B : A → Set} → Σ A B → μ (pairDesc A B)
pair-to (x , y) = ⟨ 0 , x , y , tt ⟩
```

△

In the previous chapter, conDescmap and datDescmap (Listing 3.3) were defined as the functorial map on the semantics of descriptions. For a given description D and a function from X to Y, they turned a ⟦ D ⟧ X into a ⟦ D ⟧ Y. With contexts built-in, the semantics of ConDesc requires an environment and the type of conDescmap is updated accordingly to accomodate all contexts and all environments. The type of datDescmap does not change and the implementations of both functions still look the same.

```
datDescmap : ∀ {#c X Y} (f : X → Y) (D : DatDesc #c) →
  (v : ⟦ D ⟧ X) → ⟦ D ⟧ Y
conDescmap : ∀ {Γ X Y} (f : X → Y) (D : ConDesc Γ) →
  ∀ {γ} → (v : ⟦ D ⟧ γ X) → ⟦ D ⟧ γ Y
```

The types and definitions of fold and Alg do not change at all, though they do make use of the new conDescmap through datDescmap.

```
Alg : ∀ {#c} → DatDesc #c → Set → Set
Alg D X = ⟦ D ⟧ X → X
fold : ∀ {#c} {D : DatDesc #c} {X} (α : Alg D X) → μ D → X
fold {D = D} α ⟨ xs ⟩ = α (datDescmap (fold α) D xs)
```

4.3 Ornaments

Ornaments have to be revised to use the new contexts. Particularly, the argument insertion ornament _+⊗_ should be able to use the environment to determine the type

it wants to insert. One also has to consider how the insertion or removal of arguments changes the context of the remainder of the constructor. When a new argument is inserted, the rest of the ornament should be able to use it.

The changing of contexts is encoded by two parameters of the DatOrn datatype, a starting context Γ and an output context Δ. These parameters tell us that the ornament goes from a ConDesc Γ to a ConDesc Δ. To implement the ornamental algebra later on, we also have to be able to calculate the original environment from an environment of the ornamented type. That is, a function from (δ : ⟦ Δ ⟧) to ⟦ Γ ⟧ is required which we will call the *environment transformer*. We will be working with functions between environments a lot, so an alias Cxf Γ Δ is defined to mean ⟦ Γ ⟧ → ⟦ Δ ⟧. The environment transformer is a parameter of ConDesc as well. This gives us the following types:

```
Cxf : (Γ Δ : Cx) → Set
Cxf Γ Δ = ⟦ Γ ⟧ → ⟦ Δ ⟧
DatOrn : ∀ {#c} (D : DatDesc #c) → Set₂
ConOrn : ∀ {Γ Δ} (f : Cxf Δ Γ) (D : ConDesc Γ) → Set₂
```

The environment transformer seems to go backwards here, from an environment of the ornamented type back to an environment of the original type. This is a result of the fact that every element of the ornamented type has a unique corresponding element in the original type.

We are not aware of any previous work which implemented ornaments on datatypes with contexts like these. The treatment of them is however very similar to how indices are treated usually in ornaments [19, 6], specifically in how the environment transformer works and is used as a parameter on the ornament. In fact, the next chapter will show how indices can be implemented using the same components in a very similar way.

The new definition of ornaments is given in full in listing 4.5. An ornament is always told from the outside what its environment transformer c is. This is seen in the _⊕_ constructor, where id is used as the environment transformer for ConOrn. The first arguments to _+⊗_ and give-K can both depend on an environment, just like the first argument of the _⊗_ description. Both of them use the new environment Δ.

Every ornament is responsible for providing the right transformer to its children. Ornaments like rec-⊗_ do not change the context of the rest of its tail and do not introduce additional changes to the environment, so c is simply passed along. The _+⊗_ ornament extends the context with S, meaning that the tail ornament has to go from context Γ to Δ ▷ S. The tail ornament must be given an environment transformer of type Cxf (Δ ▷ S) Γ, while we already have c of type Cxf Δ Γ. This transformer is given by cxf-forget in listing 4.6. The other ornaments which update the context use similar functions to produce environment transformers for their tails.

The rest of the definitions relating to ornaments do not differ much from the previous chapter. The interpretation function of ornaments on constructors, conOrnToDesc, now works for all contexts and environment transformers. It used to go from ConOrn D to ConDesc, now the signature becomes:

```
conOrnToDesc : ∀ {Γ Δ} {c : Cxf Δ Γ} {D : ConDesc Γ} →
  ConOrn c D → ConDesc Δ
```

The implementation of conOrnToDesc is the same as before, except that the environment transformer is used in the -⊗_ ornament. When we try to produce a description

```
data ConOrn {Γ Δ} (c : Cxf Δ Γ) : (D : ConDesc Γ) → Set₂ where
  ι : ConOrn c ι
  -⊗_ : ∀ {S xs} → (xs⁺ : ConOrn (cxf-both c) xs) → ConOrn c (S ⊗ xs)
  rec-⊗_ : ∀ {xs} → (xs⁺ : ConOrn c xs) → ConOrn c (rec-⊗ xs)
  _+⊗_ : ∀ {xs} → (S : (δ : ⟦ Δ ⟧) → Set) →
    (xs⁺ : ConOrn (cxf-forget c S) xs) → ConOrn c xs
  rec-+⊗_ : ∀ {xs} → (xs⁺ : ConOrn c xs) → ConOrn c xs
  give-K : ∀ {S xs} → (s : (γ : ⟦ Δ ⟧) → S (c γ)) →
    (xs⁺ : ConOrn (cxf-inst c s) xs) → ConOrn c (S ⊗ xs)
data DatOrn : ∀ {#c} (D : DatDesc #c) → Set₂ where
  ‘0 : DatOrn ‘0
  _⊕_ : ∀ {#c x xs} →
    (x⁺ : ConOrn id x) (xs⁺ : DatOrn xs) → DatOrn {suc #c} (x ⊕ xs)
```

Listing 4.5: Ornaments with contexts

```
Cxf : (Γ Δ : Cx) → Set
Cxf Γ Δ = ⟦ Γ ⟧ → ⟦ Δ ⟧
cxf-both : ∀ {Γ Δ S} → (c : Cxf Δ Γ) → Cxf (Δ ▷ (S ∘ c)) (Γ ▷ S)
cxf-both c δ = c (pop δ) , top δ
cxf-forget : ∀ {Γ Δ} → (c : Cxf Δ Γ) → (S : ⟦ Δ ⟧ → Set) → Cxf (Δ ▷ S) Γ
cxf-forget c S δ = c (pop δ)
cxf-inst : ∀ {Γ Δ S} → (c : Cxf Δ Γ) → (s : (γ : ⟦ Δ ⟧) → S (c γ)) → Cxf Δ (Γ ▷ S)
cxf-inst c s δ = c δ , s δ
```

Listing 4.6: Environment transformers

with -⊗_, we have to give a function ⟦ Δ ⟧ → Set representing a type within the ornamented context Δ. The ornament stored the type within the original context: S of type ⟦ Γ ⟧ → Set. The environment transformer (c : Cxf Δ Γ) helps us to transform types within the context Γ to types with the context Δ:

```
conOrnToDesc {c = c} (-⊗_ {S = S} xs⁺) = S ∘ c ⊗ conOrnToDesc xs⁺
```

We have seen in section 3.4 how the ornamental algebra for an ornament o on a description D was built using a natural transformation from the pattern functor of o to the pattern functor of D. With contexts, an environment has to be provided before we get a pattern functor for a description of a constructor, so the natural transformation must go from the pattern functor ⟦ conOrnToDesc o ⟧ δ to ⟦ D ⟧ γ for a suitable environment γ. By assuming that we know the environment δ, we can calculate the right γ by applying the environment transformer c to δ:

```
conForgetNT : ∀ {Γ Δ} {c : Cxf Δ Γ} {D : ConDesc Γ} →
  (o : ConOrn c D) →
  ∀ {δ X} → ⟦ conOrnToDesc o ⟧ δ X → ⟦ D ⟧ (c δ) X
```

Example 4.3.1. As a somewhat contrived example, we will define an ornament on the lt7Desc′ description from the introduction of this chapter. The definition is reiterated

here for convenience. The ornament inserts an argument of type IsOdd n which, obviously, says that n must be odd.

```
lt7Desc′ : DatDesc 1
lt7Desc′ = const Nat ⊗ IsLessThan7 ∘ top ⊗ ι ⊕ ‘0
postulate
  IsOdd : Nat → Set
lt7odd : DatOrn lt7Desc′
lt7odd = – ⊗ IsOdd ∘ top +⊗ – ⊗ ι ⊕ ‘0
```

Looking at the description of the ornamented type, we can see how the argument of IsLessThan7 has been updated to use the second DeBruijn index instead of the first. The call to cxf-forget in the type of the _+⊗_ constructor has caused the insertion of a pop.

```
test-lt7odd : ornToDesc lt7odd ≡
  (const Nat ⊗ IsOdd ∘ top ⊗ IsLessThan7 ∘ top ∘ pop ⊗ ι ⊕ ‘0)
test-lt7odd = refl
```

If we postulate proofs that 3 is lower than 7 and that 3 is odd, we can create an element of lt7odd for the number 3. The forget function gives the expected result.

```
postulate
  proof-that-3<7 : (3 ofType Nat) < 7
  proof-that-3-is-odd : IsOdd 3
ex-lt7odd : μ (ornToDesc lt7odd)
ex-lt7odd = ⟨ 0 , 3 , proof-that-3-is-odd , proof-that-3<7 , tt ⟩
forget-lt7odd : forget lt7odd ex-lt7odd ≡ ⟨ 0 , 3 , proof-that-3<7 , tt ⟩
forget-lt7odd = refl
```

△

Chapter 5

Ornaments on families of datatypes

Datatype parameters and indices will be added to our descriptions in this chapter. They form the final components to be able to describe a large portion of Agda datatypes. Ornaments will be revised once more to work with these descriptions. The addition of indices allows the implementation of some of the theory surrounding ornaments.

Parameters are a natural extension of contexts within descriptions—the only difference is that a full type does not need to be closed. Where we previously always started with an empty context ε for each constructor, now the whole datatype description can have a context. Within the constructors, the parameters are available as variables in the environment and they can be used with top and pop. Though the contexts are declared during the definition of a description, the interpretation of a description to a Set requires the user to pass an environment containing the parameters. This is similar to how the parameters of an Agda datatype have to be declared during the declaration of the datatype, and they have to be applied before we get a Set.

Indices are added to our descriptions as well. When indices are used, we are not just describing a single type but an inductive family of types [10]. A recursive call within a type can refer to any of the family members, so in every rec_⊗_ we must specify an index to pick a type within the family. Additionally, every type (family member) must tell us which index it has. This is done by requiring an index to be specified in the ι constructor as well. The way we implement indices is a lot like McBride's approach [19], though we make use of our Cx datatype to allow multiple indices.

Parameters and indices will both be declared using a Cx as a parameter on the DatDesc type. A type like Vec, which has one parameter of type Set and one index of type Nat, is described with the type DatDesc (ε ▷′ Nat) (ε ▷′ Set) 2.

5.1 Descriptions

Descriptions of constructors where already parameterised by a (Γ : Cx), now we also add a parameter (I : Cx). The declared indices I stay constant across all constructors. The context Γ is initially equal to the parameters, but can be extended within the constructors like in the previous chapter.

Listing 5.1 shows the new definitions for ConDesc and DatDesc. The interesting changes are in the ι and rec_⊗_ constructors, which both have gained a new argument.

```
data ConDesc (I : Cx) (Γ : Cx) : Set₁ where
  ι : (o : (γ : ⟦ Γ ⟧) → ⟦ I ⟧) → ConDesc I Γ
  _⊗_ : (S : (γ : ⟦ Γ ⟧) → Set) → (xs : ConDesc I (Γ ▷ S)) → ConDesc I Γ
  rec_⊗_ : (i : (γ : ⟦ Γ ⟧) → ⟦ I ⟧) → (xs : ConDesc I Γ) → ConDesc I Γ
data DatDesc (I : Cx) (Γ : Cx) : (#c : Nat) → Set₁ where
  ‘0 : DatDesc I Γ 0
  _⊕_ : ∀ {#c} (x : ConDesc I Γ) (xs : DatDesc I Γ #c) →
    DatDesc I Γ (suc #c)
```

Listing 5.1: Descriptions of families of datatypes

```
data DescTag : Set₂ where
  isCon : DescTag
  isDat : (#c : Nat) → DescTag
Desc : (I : Cx) → (Γ : Cx) → DescTag → Set₁
Desc I Γ (isCon) = ConDesc I Γ
Desc I Γ (isDat #c) = DatDesc I Γ #c
```

Listing 5.2: Definition of DescTag and Desc

In the ι constructor, the user can use the local environment (γ : ⟦ Γ ⟧) to specify an index of type ⟦ I ⟧. The rec_⊗_ constructor also requires the specification of an index of type ⟦ I ⟧, and here too the local environment can be used.

Before the semantics for ConDesc and DatDesc are defined, we will take a slight detour. In previous chapters many functions for ConDesc and DatDesc were defined separately. Now that ConDesc and DatDesc have some overlapping parameters, it will become bothersome to have to write many of the same function signatures for both of them. Writing the same thing twice is a bad programming habit, so this is circumvented by defining a small universe in listing 5.2. Using DescTag and Desc, we can refer to ConDesc I Γ as Desc I Γ isCon and to DatDesc I Γ #c as Desc I Γ (isDat #c). Functions which have to be defined on both ConDesc and DatDesc can now be defined on Desc dt for all dt. All functions that use DescTag can also be defined as one or more functions that do not use it, but the homogeneous treatment of all descriptions will provide some benefit later.

The semantics of descriptions is one of those functions which have the same type for both ConDesc and DatDesc. The type is quantified over all dt, so it takes the following form:

```
⟦_⟧desc : ∀ {I Γ dt} → Desc I Γ dt → ?
```

The type which goes in the hole ? is a bit more involved than what we have previously seen. The semantics of DatDesc in the previous chapters gave an endofunctor on Set. Dybjer [11] has shown how an inductive family (with indices) can be described using an endofunctor on I′ → Set. We use ⟦ I ⟧ instead of I′, to encode the idea of having a telescope of indices instead of just one. By interpreting the description as an endofunctor on ⟦ I ⟧ → Set, the recursive positions are allowed to pick an index of type ⟦ I ⟧ in return for a Set. An environment for the current context has to be passed in as well, but this is not part of the endofunctor. This results in the following type:

```
⟦_⟧desc : ∀ {I Γ dt} → Desc I Γ dt → ⟦ Γ ⟧ → (⟦ I ⟧ → Set) → (⟦ I ⟧ → Set)
```

```
⟦_⟧desc : ∀ { I Γ dt } → Desc I Γ dt → ⟦ Γ ⟧ → (⟦ I ⟧ → Set) → (⟦ I ⟧ → Set)
⟦_⟧desc { dt = isCon } (ι o′) γ X o = o′ γ ≡ o
⟦_⟧desc { dt = isCon } (S ⊗ xs) γ X o = Σ (S γ) λ s → ⟦ xs ⟧desc (γ , s) X o
⟦_⟧desc { dt = isCon } (rec i ⊗ xs) γ X o = X (i γ) × ⟦ xs ⟧desc γ X o
⟦_⟧desc { dt = isDat #c } D γ X o = Σ (Fin #c) λ k → ⟦ lookupCtor D k ⟧desc γ X o
data μ { I Γ #c } (D : DatDesc I Γ #c) (γ : ⟦ Γ ⟧) (o : ⟦ I ⟧) : Set where
  ⟨_⟩ : ⟦ D ⟧ γ (μ D γ) o → μ F γ o
```

Listing 5.3: Semantics of datatype families

The full definition of ⟦_⟧$_{desc}$ is given in listing 5.3. In every clause, we get a local environment γ just like we did in ⟦_⟧$_{conDesc}$ in section 4.2—this time including the parameters. The value X of type ⟦ I ⟧ → Set is used in the clause for rec i ⊗ xs to pick one of the members of the inductive family. The o is what we are being told what the index of the type *should* be. In ι o′, the description *says* that the index is o′ γ. Therefore, the interpretation of ι o′ is an equality constraint o′ γ ≡ o. The use of the indices is similar to McBride's definitions [19], but here the indices are determined with help of the local environment γ.

By partially applying the fixpoint datatype μ with a description D and the parameters γ, we get μ D γ of type ⟦ I ⟧ → Set. This is exactly the type we need to pass to ⟦_⟧$_{desc}$ within the definition of the ⟨_⟩ constructor. The parameters, i.e. the starting environment, are passed along unchanged to all the recursive children.

Example 5.1.1. The List type has no indices and one parameter of type Set, so the description is of type DatDesc ε (ε ▷′ Set) 2. The ι and rec_⊗_ constructors both require the user to specify an index—this can only be tt as that is the only inhabitant of ⟦ ε ⟧. The description of List can now be defined as follows:

```
listDesc : DatDesc ε (ε ▷′ Set) 2
listDesc = ι (const tt) ⊕
  top ⊗ rec (const tt) ⊗ ι (const tt) ⊕
  ‘0
```

Comparing this to how listDesc was defined in section 3.1 and the introduction of chapter 4, we see that the use of the parameter has been internalised. Where it used to say A, we now see a top.

The fixpoint of descriptions is aware of parameters and indices, and it is required to instantiate them before we obtain a Set. The type of μ listDesc is now ⟦ ε ▷′ Set ⟧ → ⟦ ε ⟧ → Set. Ignoring the fluff, one can see how this is similar to the type of List: Set → Set. The function list-to shows how μ listD can be used:

```
list-to : ∀ { A } → List A → μ listD (tt , A) tt
list-to [] = ⟨ 0 , refl ⟩
list-to (x ∷ xs) = ⟨ 1 , x , list-to xs , refl ⟩
```

With the descriptions of chapter 3 and chapter 4, the type of list-to would have been ∀ { A } → List A → μ (listDesc A). △

Example 5.1.2. One of the simplest inductive families we can make is those of finite naturals. It has the normal zero and suc constructors of naturals, but it is indexed by a Nat which limits how high the value can be. The set Fin n contains naturals which are lower than n. It is usually defined as follows:

```
data Fin : Nat → Set where
  zero : ∀ { n } → Fin (suc n)
  suc : ∀ { n } (i : Fin n) → Fin (suc n)
```

The Fin type can be described by a DatDesc (ε ▷′ Nat) ε. Our descriptions do not allow implicit arguments, so we make them explicit. We can easily write down part of the description, leaving open the holes where all the indices have to be specified:

```
finDesc : DatDesc (ε ▷′ Nat) ε 2
finDesc = const Nat ⊗ ι ?1
  ⊕ const Nat ⊗ rec ?2 ⊗ ι ?3
  ⊕ ‘0
```

All the open holes happen to have a local context where just one Nat exists, so they all require a term of type ⟦ ε ▷′ Nat ⟧ → ⟦ ε ▷′ Nat ⟧. The ⟦ ε ▷′ Nat ⟧ that goes in is the environment, while the returned ⟦ ε ▷′ Nat ⟧ contains the index of the type being constructed (for the ι's) or of the type that is required (for rec). This situation is comparable to what we would get if we defined Fin using one parameter of type ⟦ ε ⟧ and one index of type ⟦ ε ▷′ Nat ⟧. In the following alternative definition of Fin, all the holes have (A : ⟦ ε ⟧) and (n : Nat) in the context, and have to be of type ⟦ ε ▷′ Nat ⟧.

```
data FinTT (A : ⟦ ε ⟧) : ⟦ ε ▷′ Nat ⟧ → Set where
  zero : (n : Nat) → FinTT A ?1
  suc : (n : Nat) (i : FinTT A ?2 ) → FinTT A ?3
```

The definition of FinTT could be completed by filling in (tt , suc n), (tt , n) and (tt , suc n) in the holes. To complete the definition of finDesc, we merely need to replace n with top γ and add the lambda-abstractions:

```
finDesc : DatDesc (ε ▷′ Nat) ε 2
finDesc = const Nat ⊗ ι (λ γ → tt , suc (top γ))
  ⊕ const Nat ⊗ rec (λ γ → tt , top γ) ⊗ ι (λ γ → tt , suc (top γ))
  ⊕ ‘0
```

An element of type μ finDesc tt (tt , 10) is a natural which is lower than 10. Let us build such an element which represents the value zero. This is done by immediately picking the first constructor (⟨ 0 , ⋯ ⟩). Then we need to give a Nat, which is the implicit n in the definition of Fin, and a proof that the created element has the right index.

```
fin-zero′ : μ finDesc tt (tt , 10)
fin-zero′ = ⟨ 0 , ?0 , ?1 ⟩
-- ?0 : Nat
-- ?1 : (tt , suc ?0) ≡ (tt , 10)
```

The obvious definitions for the holes ?0 and ?1 are 9 and refl, completing the definition of fin-zero.

```
fin-zero : μ finDesc tt (tt , 10)
fin-zero = ⟨ 0 , 9 , refl ⟩
```

△

```
_→ⁱ_ : {I : Set} → (I → Set) → (I → Set) → Set
X →ⁱ Y = ∀ {i} → X i → Y i
```

Listing 5.4: Arrows in the I → Set category

```
descmap : ∀ {I Γ dt X Y} (f : X →ⁱ Y) (D : Desc I Γ dt) →
  ∀ {γ} → ⟦ D ⟧ γ X →ⁱ ⟦ D ⟧ γ Y
descmap {dt = isCon} f (ι o) refl = refl
descmap {dt = isCon} f (S ⊗ xs) (s , v) = s , descmap f xs v
descmap {dt = isCon} f (rec i ⊗ xs) (s , v) = f s , descmap f xs v
descmap {dt = isDat _} f xs (k , v) = k , descmap f (lookupCtor xs k) v
```

Listing 5.5: Map for pattern functors with indices

We claimed that $⟦_⟧_{desc}$ gave us a functor on ⟦ I ⟧ → Set, so we should be able to define a functorial map. Within functional programming, we often assume that functors are from the Set category to the Set category, with functions as the arrows. When working in the ⟦ I ⟧ → Set category, one has to reconsider what the arrows are. The arrows are characterised as functions which respect indexing, they are defined as _→ⁱ_ in listing 5.4[1].

A map function for a functor F in the I → Set category should lift an arrow X →ⁱ Y to an arrow F X →ⁱ F Y. By instantiating ⟦ D ⟧ γ to F we get the type for descmap, which is fully defined in listing 5.5. The implementation is straightforward, but it is listed for completeness.

The definitions of the Alg and fold in listing 5.6 are lifted to the ⟦ I ⟧ → Set category in a similar way, by replacing some of the arrows _→_ with _→ⁱ_. An environment ⟦ Γ ⟧ has to be passed to Alg, because an algebra might only work for a specific environment. For example; an algebra to calculate the sum of a list would be of type Alg listDesc (tt , Nat) (const Nat), where (tt , Nat) instantiates the parameter of type Set to Nat. An algebra like the one to calculate the length will work for any parameter, so it will have the type ∀ {A} → Alg listDesc (tt , A) (const Nat). The code below demonstrates how these algebras can be defined:

```
sumAlg : Alg listDesc (tt , Nat) (const Nat)
sumAlg (zero , refl) = 0
sumAlg (suc zero , x , xs , refl) = x + xs
sumAlg (suc (suc ()) , _)

lengthAlg : ∀ {A} → Alg listDesc (tt , A) (const Nat)
lengthAlg (zero , refl) = 0
lengthAlg (suc zero , x , xs , refl) = suc xs
lengthAlg (suc (suc ()) , _)
```

Example 5.1.3. An algebra can be used to define the raise function for Fin. It takes a natural m and lifts a Fin n into Fin (n + m) while the represented number stays the same. It has the following type:

```
raise : ∀ {n} → (m : Nat) → Fin n → Fin (n + m)
```

[1] Actually, the _→ⁱ_ definition works for a more general category A → Set, of which ⟦ I ⟧ → Set is an instance.

```
Alg : ∀ {I Γ dt} → Desc I Γ dt → ⟦ Γ ⟧ → (⟦ I ⟧ → Set) → Set
Alg {I} D γ X = ⟦ D ⟧ γ X →ⁱ X
fold : ∀ {I Γ #c} {D : DatDesc I Γ #c} {γ X} (α : Alg D γ X) → μ D γ →ⁱ X
fold {D = D} α ⟨ xs ⟩ = α (descmap (fold α) D xs)
```

Listing 5.6: Generic fold

An algebra on finDesc which calculates this value should have μ finDesc tt (tt , n + m) as its return type, where n is the index of the Fin that was given. This can be represented with a ⟦ I ⟧ → Set function, as is required by Alg. To refer to n we take the top of the indices, giving us the following type for raiseAlg:

```
raiseAlg : (m : Nat) → Alg finDesc tt (λ i → μ finDesc tt (tt , top i + m))
```

The rest of the definition is a straightforward case split on the constructors. In the pattern for zero we build a new zero, and in the pattern for suc we build a new suc. In both cases the index changes from n to n + m.

```
raiseAlg : (m : Nat) → Alg finDesc tt (λ i → μ finDesc tt (tt , top i + m))
raiseAlg m (zero , n , refl) = ⟨ 0 , n + m , refl ⟩
raiseAlg m (suc zero , n , k , refl) = ⟨ 1 , n + m , k , refl ⟩
raiseAlg m (suc (suc ()) , _)
```

By folding raiseAlg m we get a function that takes a representation of a Fin n to a representation of a Fin (n + m). The top i in the type of the algebra is correctly translated to the index n of the Fin n that goes in.

```
'raise : ∀ {n} → (m : Nat) → μ finDesc tt (tt , n) → μ finDesc tt (tt , n + m)
'raise m = fold {D = finDesc} (raiseAlg m)
```

If we want to, the 'raise function can be adapted to work on real Fins by using the embedding-projection pair defined below. The raise function simply chains the fin-to, 'raise and fin-from together.

```
fin-to : ∀ {n} → Fin n → μ finDesc tt (tt , n)
fin-to zero = ⟨ 0 , _ , refl ⟩
fin-to (suc k) = ⟨ 1 , _ , fin-to k , refl ⟩
fin-from : ∀ {n} → μ finDesc tt (tt , n) → Fin n
fin-from ⟨ zero , _ , refl ⟩ = zero
fin-from ⟨ suc zero , _ , k , refl ⟩ = suc (fin-from k)
fin-from ⟨ suc (suc ()) , _ ⟩
raise : ∀ {n} → (m : Nat) → Fin n → Fin (n + m)
raise m = fin-from ∘ 'raise m ∘ fin-to
```

△

5.2 Ornaments

The definition of ornaments on descriptions with parameters and indices is mostly based on the constructor ornaments of section 4.3 (Listing 4.5). The same context transformers (Listing 4.6) are used, but this time on both the indices and the context/parameters. Many of the parts relating to indices are based on McBride's ornaments [19].

```
module _ {a b} {A : Set a} {B : Set b} where
  -- f ⁻¹ y contains an x such that f x ≡ y
  data _⁻¹_ (f : A → B) : (y : B) → Set (a ⊔ b) where
    inv : (x : A) → f ⁻¹ (f x)

  uninv : {f : A → B} {y : B} → f ⁻¹ y → A
  uninv (inv x) = x

  inv-eq : {f : A → B} {y : B} → (invx : f ⁻¹ y) → f (uninv invx) ≡ y
  inv-eq (inv x) = refl
```

Listing 5.7: Inverse image of functions

Using our DescTag codes, a single datatype for ornaments can be defined which contains ornaments for both ConDesc and DatDesc. Like ConOrn of section 4.3, it contains the starting context Γ, the result context Δ and an environment transformer (u : Cxf Δ Γ) as parameters. The indices are added in a similar way using a starting index I, result index J and a transformer between indices (u : Cxf J I). The Orn datatype gets the following signature:

```
data Orn {I} J (u : Cxf J I) {Γ} Δ (c : Cxf Δ Γ) :
  ∀ {dt} (D : Desc I Γ dt) → Set₁
```

Remember that the type Cxf J I of the index transformer u expands to ⟦ J ⟧ → ⟦ I ⟧. It is *meant* to allow the mapping from indices of elements in the ornamented type, back to indices of the original type. The existence of such a function ensures that the index J is more informative than I, and that the extra information can be forgotten. The index transformer u does *not* ensure that an ornamented index maps back to the *same* original index. So when a ι i is ornamented to a ι j, where (i : ⟦ I ⟧) and (j : ⟦ J ⟧), the j could be mapped back to a (i′ : ⟦ I ⟧) which is different than the original i. For the correct behavior of ornaments, we need to know that u j gives the original i—j must lie in the *inverse image* of i for the function u.

Like McBride [19], we use a datatype to define the inverse image of a function (Listing 5.7). The only constructor of f ⁻¹ y says that the index y must be f x, so a value of type f ⁻¹ y always contains an x such that f x ≡ y. The function uninv extracts the x from inv, to avoid having to pattern match in other places. The inv-eq function delivers the f x ≡ y equality, which will proof useful later.

Note also that a small module is used with for the parameters a, b, A and B. These arguments are shared between all three functions. Because the module is nameless (it is named _), the module is transparent to function calls—Meaning that outside the module, these functions can be called as if they had been defined outside of the module, with the module parameters as function arguments.

In listing 5.8 the new ornaments are defined. All the constructors now fit in a single Orn datatype, and the contexts are now propagated in the ‘0 and _⊕_ ornaments as well. The rec_+⊗_ ornament gains a simple extension: an index j of type ⟦ J ⟧ can be chosen by making use of the ornamented environment δ.

In the ι and rec_⊗_ copy ornaments, a new index must be given which—for the function u—lies in the inverse image of the original index. The ornamented environment δ may be used to determine this index. The original index i can only be determined using the original environment, which is reconstructed by applying the environment transformer c to the ornamented environment δ.

```
data Orn {I} J (u : Cxf J I)
  {Γ} Δ (c : Cxf Δ Γ) : ∀ {dt} (D : Desc I Γ dt) → Set₁ where
  ι : ∀ {i} → (j : (δ : ⟦ Δ ⟧) → u ⁻¹ (i (c δ))) → Orn _ u _ c (ι i)
  -⊗_ : ∀ {S xs} → (xs⁺ : Orn _ u _ (cxf-both c) xs) → Orn _ u _ c (S ⊗ xs)
  rec_⊗_ : ∀ {i xs} → (j : (δ : ⟦ Δ ⟧) → u ⁻¹ (i (c δ))) →
    (xs⁺ : Orn _ u _ c xs) → Orn _ u _ c (rec i ⊗ xs)
  _+⊗_ : ∀ {xs : ConDesc I Γ} → (S : (δ : ⟦ Δ ⟧) → Set) →
    (xs⁺ : Orn _ u _ (cxf-forget c S) xs) → Orn _ u _ c xs
  rec_+⊗_ : ∀ {xs : ConDesc I Γ} → (j : (δ : ⟦ Δ ⟧) → ⟦ J ⟧) →
    (xs⁺ : Orn _ u _ c xs) → Orn _ u _ c xs
  give-K : ∀ {S xs} → (s : (δ : ⟦ Δ ⟧) → S (c δ)) →
    (xs⁺ : Orn _ u _ (cxf-inst c s) xs) → Orn _ u _ c (S ⊗ xs)
  ‘0 : Orn _ u _ c ‘0
  _⊕_ : ∀ {#c x} {xs : DatDesc I Γ #c} →
    (x⁺ : Orn _ u _ c x) (xs⁺ : Orn _ u _ c xs) → Orn _ u _ c (x ⊕ xs)
```

Listing 5.8: Ornaments for families of datatypes

```
module _ {I J u} where
  ornToDesc : ∀ {Γ Δ c dt} {D : Desc I Γ dt} →
    (o : Orn J u Δ c D) → Desc J Δ dt
  ornToDesc (ι j) = ι (uninv ∘ j)
  ornToDesc {c = c} (-⊗_ {S = S} xs⁺) = S ∘ c ⊗ ornToDesc xs⁺
  ornToDesc (rec j ⊗ xs⁺) = rec (uninv ∘ j) ⊗ ornToDesc xs⁺
  ornToDesc (_+⊗_ S xs⁺) = S ⊗ ornToDesc xs⁺
  ornToDesc (rec_+⊗_ j xs⁺) = rec j ⊗ ornToDesc xs⁺
  ornToDesc (give-K s xs⁺) = ornToDesc xs⁺
  ornToDesc ‘0 = ‘0
  ornToDesc (x⁺ ⊕ xs⁺) = ornToDesc x⁺ ⊕ ornToDesc xs⁺
```

Listing 5.9: Interpretation of ornaments

The semantics of ornaments is listed in listing 5.9. A small nameless module is used to put the quantification over I, J and u outside of the ornToDesc function. Module parameters in Agda remain constant between calls within the module, so this emphasises that the indices are the same within every part of a description.

The combination of ConDesc and DatDesc into a single Desc type works very well here: A single function is required to define the semantics of ornaments on both constructors and datatypes and there is no mention of DescTags within the clauses. The term uninv ∘ j occurs twice, in the clauses for ι j and for rec j ⊗ xs⁺. The function j gives the new index, of type u ⁻¹ i (c δ), when it is applied to an ornamented environment. The index is then extracted from using uninv.

Example 5.2.1. Let us get some practice with the new ornaments by refining lists to Vecs. Recall the definition of the Vec type:

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A 0
  _::_ : ∀ {n} → (x : A) → (xs : Vec A n) → Vec A (suc n)
```

The Vec type has one index of type Nat and one parameter of type Set. By using

listDesc as the starting point, the type of the ornament to be defined has the following structure:

```
list→vec : Orn (ε ▷′ Nat) ?0 (ε ▷′ Set) ?1 listDesc
```

The hole ?0 is the index transformer and must have be of type ⟦ ε ▷′ Nat ⟧ → ⟦ ε ⟧. The result type has only one inhabitant, so the implementation is easily given as λ j → tt. The parameter transformer ?1 must have type ⟦ ε ▷′ Set ⟧ → ⟦ ε ▷′ Set ⟧. We do not want the parameters to change—if A is the parameter for the list, A should be the parameter for the Vec—so we use the identity function id. Structurally, the ornament should only insert one argument; the n in the _::_ constructor with which the index is determined. Skipping the indices, the ornament looks as follows.

```
list→vec : Orn (ε ▷′ Nat) (λ j → tt) (ε ▷′ Set) id listDesc
list→vec = ι ?0
  ⊕ const Nat +⊗ - ⊗ rec ?1 ⊗ ι ?2
  ⊕ ‘0
-- ?0 : (δ : ⊤′ ▶ const Set) → (λ j → tt) ⁻¹ tt
-- ?1 : (δ : ⊤′ ▶ const Set ▶ const Nat ▶ top ∘ pop) → (λ j → tt) ⁻¹ tt
-- ?2 : (δ : ⊤′ ▶ const Set ▶ const Nat ▶ top ∘ pop) → (λ j → tt) ⁻¹ tt
```

The first hole to be filled in asks for a (λ j → tt) ⁻¹ tt, this means we can fill in inv x where x must be of type ⟦ ε ▷′ Nat ⟧ (the type of the new indices) and (λ j → tt) x must be equal to tt. The second requirement is met trivially, so any value with the right type will do. In this case the length index should be zero, so tt , 0 is filled in.

For the holes ?1 and ?2 the situations are similar, any index of type ⟦ ε ▷′ Nat ⟧ can be chosen. Note that both holes can use the ornamented environment, including the new argument of type Nat, to determine the index. The holes ?1 and ?2 are filled in to use n and suc n respectively as the length index.

```
list→vec : Orn (ε ▷′ Nat) (λ _ → tt) (ε ▷′ Set) id listDesc
list→vec = ι (λ δ → inv (tt , 0))
  ⊕ const Nat +⊗ - ⊗ rec (λ δ → inv (tt , top (pop δ)))
  ⊗ ι (λ δ → inv (tt , suc (top (pop δ))))
  ⊕ ‘0
```

As a quick verification that this ornament results in a type which does the same thing as Vec, part of the embedding-projection pair is given:

```
vecDesc : DatDesc (ε ▷′ Nat) (ε ▷′ Set) 2
vecDesc = ornToDesc list→vec

vec-to : ∀ {A n} → Vec A n → μ vecDesc (tt , A) (tt , n)
vec-to [] = ⟨ 0 , refl ⟩
vec-to (x :: xs) = ⟨ 1 , _ , x , vec-to xs , refl ⟩
```

△

The ornamental algebra is an extension of the ornamental algebras we have seen before. The full listing is given in listing 5.10. The index types I and J and the index transformer u are module parameters to emphasise that they remain the same between forget, forgetAlg and forgetNT.

```
module _ {I J u} where
  forgetNT : ∀ {Γ Δ c dt} {D : Desc I Γ dt} (o : Orn J u Δ c D) →
    ∀ {δ X j} → ⟦ ornToDesc o ⟧ δ (X ∘ u) j → ⟦ D ⟧ (c δ) X (u j)
  forgetNT (ι j) {δ} refl rewrite inv-eq (j δ) = refl
  forgetNT (-⊗ xs⁺) (s , v) = s , forgetNT xs⁺ v
  forgetNT (rec j ⊗ xs⁺) {δ} {X} (s , v) rewrite inv-eq (j δ) = s , forgetNT xs⁺ v
  forgetNT (_+⊗_ S xs⁺) (s , v) = forgetNT xs⁺ v
  forgetNT (rec_+⊗_ j xs⁺) (s , v) = forgetNT xs⁺ v
  forgetNT (give-K s xs⁺) {δ} v = s δ , forgetNT xs⁺ v
  forgetNT ‘0 (() , _)
  forgetNT (x⁺ ⊕ xs⁺) (zero , v) = zero , forgetNT x⁺ v
  forgetNT (x⁺ ⊕ xs⁺) (suc k , v) = (suc *** id) (forgetNT xs⁺ (k , v))

  forgetAlg : ∀ {Γ Δ c #c} {D : DatDesc I Γ #c} (o : Orn J u Δ c D) →
    ∀ {δ} → Alg (ornToDesc o) δ (μ D (c δ) ∘ u)
  forgetAlg o = ⟨_⟩ ∘ forgetNT o

  forget : ∀ {Γ Δ c #c} {D : DatDesc I Γ #c} (o : Orn J u Δ c D) →
    ∀ {δ j} → μ (ornToDesc o) δ j → μ D (c δ) (u j)
  forget o = fold (forgetAlg o)
```

Listing 5.10: Ornamental algebras

The type of forgetNT is similar to that of conForgetNT in section 4.3, in that they go from the functor ⟦ ornToDesc o ⟧ δ to the functor ⟦ D ⟧ (c δ). The *ornamented* environment is passed as an argument to forgetNT, and the environment for the original type is obtained by applying the environment transformer c. Both functors require a new argument containing an index (to get to a Set), these are handled similarly to the environments. The index j of the ornamented type is transformed to the index for the original type by applying u. An X of type ⟦ I ⟧ → Set can be combined with u (of type ⟦ J ⟧ → ⟦ I ⟧ to get the other arguments for the functors. The resulting types are similar to those of McBride [19], with the addition of environments.

The cases of forgetNT for ι j and rec j ⊗ xs⁺ both require the proof inv-eq (j δ). In both clauses j δ is of type u ⁻¹ i (c δ), so it contains a value in the inverse-image of i (c δ). The proof inv-eq (j δ) says that u (uninv (j δ)) ≡ i (c δ), confirming that applying u on the ornamented index (j δ) results in the original index (i (c δ)). Alternatively, we might state: The index j under the environment δ lies in the inverse-image for u of i under the environment c δ. Rewriting with the proof unifies enough variables that the proof obligation for the ι j case becomes i (c δ) ≡ i (c δ), allowing us to write refl as the term. The type of s in the rec j ⊗ xs⁺ case is rewritten to X (i (c δ)), which is exactly what is needed on the right side.

5.3 Algebraic ornaments

Now that indices are supported, *algebraic ornaments* can be defined. When an algebra is given for a description D, it induces an algebraic ornament on D which adds the results of the algebra as an index. The type of algOrn below shows how an algebra which results in a value of type R gives an ornament which goes from a Desc I Γ dt to a Desc (I ▷ R) Γ dt.

```
algOrn : ∀ {Γ dt} (D : Desc I Γ dt) →
  ({γ : ⟦ Γ ⟧} → Alg D γ R) → Orn (I ▷ R) pop Γ id D
```

Interestingly, algebraic ornaments only work when the algebra is polymorphic in the datatype parameters. So lengthAlg for lists could be used, but sumAlg could not. During the definition of an ornament we do not know which environment will be used, so it should work for any environment. To produce an index of type R for any environment, the algebra must work for any environment. One quickly gets stuck when trying to define algOrn for a fixed environment.

What exactly should an algebraic ornament do? Consider the Vec datatype. We would like to get a descriptions of Vec by using the algebraic ornament of the length algebra for lists. We reiterate the definitions of Vec and lengthAlg below. By comparing them, one may note that the result indices 0 and suc n of the [] and _::_ constructors match with the right sides of the first and second clause of lengthAlg. In an algebra, every recursive argument is matched with the result of the algebra on that argument; this can be used to write the right hand side. In the resulting datatype (Vec) the result for the recursive argument xs is kept in a new argument n. We will call n the index-holding argument for xs.

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A 0
  _::_ : ∀ {n} → (x : A) → (xs : Vec A n) → Vec A (suc n)
lengthAlg : {A : Set} → Alg listDesc (tt , A) (const Nat)
lengthAlg (zero , refl) = 0
lengthAlg (suc zero , x , n , refl) = suc n
lengthAlg (suc (suc ()) , _)
```

We will generalise the observations on vectors to get the formula for building algebraic ornaments: For every *recursive* argument, the result of the algebra will be held in a new index-holding argument that is inserted right before it. The index-holding arguments are passed to the algebra to compute the result indices.

Algebraic ornaments are implemented in listing 5.11. The implementation itself is in algOrn′, which has a slightly different type than algOrn. Because new arguments are being inserted, the recursive calls may have a modified context. The algOrn′ function supports context changes by having two additional arguments Δ and c. The type of algOrn is a bit more convenient to use in practice—It helps with some type inference.

The algebra is used up piece by piece while recursing over the description. Though the algebra α only requires one argument, this argument is a product type in most cases (for all descriptions but ι), so curry can be used to instantiate part of the product. The case for S ⊗ xs shows it clearly: An α of type Alg (S ⊗ xs) _ _, which normalises to Σ (S _) _ → R _, is curried to get a function S _ → Alg xs _ _. The top γ is of the correct type S _, so with curry α (top γ) we get an algebra which works for the tail of the description xs.

The case for rec i ⊗ xs shows how the index-holding argument R ∘ i ∘ c is inserted. Here c transforms the ornamented environment of type ⟦ Δ ⟧ into an environment of type ⟦ Γ ⟧, i tells us the index that was used for the recursive argument under that environment, and R gives the type of the result under that index. The recursive argument is copied, but with a new index consisting of two parts: i (c (pop δ)) and top δ. The first part is effectively the old index, but calculated by using the pop δ environment (the ornamented environment excluding the newly inserted argument). The second part top δ is the value of the newly inserted argument.

```
module _ {I R} where
  algOrn′ : ∀ {Γ Δ dt} {c : Cxf Δ Γ} (D : Desc I Γ dt) →
    (∀ {δ : ⟦ Δ ⟧} → Alg D (c δ) R) → Orn (I ▷ R) pop Δ c D
  algOrn′ {dt = isCon} {c = c} (ι o) α = ι (λ δ → inv (o (c δ) , α refl))
  algOrn′ {dt = isCon} (S ⊗ xs) α = - ⊗ (algOrn′ xs (λ {γ} → curry α (top γ)))
  algOrn′ {dt = isCon} {c = c} (rec i ⊗ xs) α = R ∘ i ∘ c +⊗
    rec (λ δ → inv (i (c (pop δ)) , top δ)) ⊗
    algOrn′ xs (λ {δ} → curry α (top δ))
  algOrn′ {dt = isDat _} ‘0 α = ‘0
  algOrn′ {dt = isDat _} (x ⊕ xs) α = algOrn′ x (curry α 0)
    ⊕ algOrn′ xs (α ∘ (suc *** id))
  algOrn : ∀ {Γ dt} (D : Desc I Γ dt) →
    ({γ : ⟦ Γ ⟧} → Alg D γ R) → Orn (I ▷ R) pop Γ id D
  algOrn = algOrn′
```

Listing 5.11: Algebraic ornaments

Example 5.3.1. The Vec type can be described with algOrn. The length algebra can be used to do this. The signature of list→vec′ is the same as that for the previously defined list→vec. A new index of type const Nat is introduced, because that is the result type of lengthAlg.

```
list→vec₁ : Orn (ε ▷ const Nat) pop (ε ▷′ Set) id listDesc
list→vec₁ = algOrn listDesc lengthAlg
```

The ornament results in a description of Vec. Do note that the order of the arguments of the _::_ constructor is slightly different, because the new argument n is being inserted right before the recursive argument.

```
vecDesc₁ : DatDesc (ε ▷′ Nat) (ε ▷′ Set) 2
vecDesc₁ = ι (const (tt , 0))
  ⊕ top
  ⊗ const Nat
  ⊗ rec (λ γ → tt , top γ)
  ⊗ ι (λ γ → tt , suc (top γ))
  ⊕ ‘0
test-list→vec₁ : ornToDesc list→vec₁ ≡ vecDesc₁
test-list→vec₁ = refl
```

△

5.4 Discussion

This chapter has shown how descriptions the descriptions with contexts can be extended to support both parameters and indices. Parameters are a fairly simple addition, but indices required some rethinking of what the types of our functors had to be (the change from Set to ⟦ I ⟧ → Set). Existing literature on ornaments adapts well to this universe, and most importantly we were able to implement the ornamental algebra. Additionally, algebraic ornaments were implemented.

Some interesting functionality from McBride's [19] work relating to algebraic ornaments has not yet been implemented due to a lack of time. One is the remember function, which is the inverse of forget for algebraic ornaments. For example, if one has a list and its length algebra, it may be used to convert lists to Vecs. The type will be stated here, but it has not been implemented. McBride uses a general induction principle to define remember, which has not (yet) been implemented either.

```
remember : ∀ { I R Γ #c } (D : DatDesc I Γ #c) →
  (α : ∀ { γ } → Alg D γ R) →
  ∀ { γ i } → (x : μ D γ i) → μ (ornToDesc (algOrn D α)) γ (i , (fold α x))
```

The recomputation lemma states: When an algebraic ornament is forgotten, folding the same algebra over the result recomputes the index of the original value. It is stated as follows:

```
recomputation : ∀ { I R Γ #c } (D : DatDesc I Γ #c) →
  (α : { γ : ⟦ Γ ⟧ } → Alg D γ R) →
  ∀ { γ j } → (x : μ (ornToDesc (algOrn D α)) γ j) →
  fold α (forget (algOrn D α) x) ≡ top j
```

Example 5.4.1. The remember and recomputation functions have not been implemented for this thesis. If one were to define them, some interesting results could be obtained. Consider the length algebra for lists. It is used to define the length′ function and the Vec type:

```
length′ : ∀ { A } → μ listDesc (tt , A) tt → Nat
length′ = fold lengthAlg
vecDesc′ : DatDesc (ε ▷′ Nat) (ε ▷′ Set) 2
vecDesc′ = ornToDesc (algOrn listDesc lengthAlg)
```

Like any ornament, the length algebraic ornament can be forgotten to convert any Vec back to a list:

```
vec-to-list : ∀ { A n } → μ vecDesc′ (tt , A) (tt , n) →
  μ listDesc (tt , A) tt
vec-to-list = forget (algOrn listDesc lengthAlg)
```

The remember function would allow the list-to-vec function to be defined in terms of lengthAlg. The length index is computed with fold lengthAlg, which we have defined as length′.

```
list-to-vec : ∀ { A } → (x : μ listDesc (tt , A) tt) →
  μ vecDesc′ (tt , A) (tt , length′ x)
list-to-vec = remember listDesc lengthAlg
```

One would expect that when a Vec A n in converted to a list, the length of that list would be n. The recomputation lemma would help to prove this fact:

```
length-recomputation : ∀ { A n } → (x : μ vecDesc′ (tt , A) (tt , n)) →
  length′ (vec-to-list x) ≡ n
length-recomputation x = recomputation listDesc lengthAlg x
```

△

5.4.1 Separating parameters from contexts

One of the limitations of the current implementation of indices and parameters is that indices can not use the parameters. For instance in the description of the following datatype Silly, one gets stuck when trying to give the type of the index. The hole ?0 must be of type Nat, while only γ of type ⊤′ is available.

```
data Silly (n : Nat) : Fin n → Set where
  silly : (k : Fin n) → (b : Bool) → Foo n k

sillyDesc : DatDesc (ε ▷ (λ γ → Fin ?0 )) (ε ▷′ Nat) 1
sillyDesc = ···
```

It is not easy to make the indices depend on the parameters within the current implementation, because parameters are not a separate thing within the descriptions. The datatype parameters are merely the initial context, which is being expanded in the constructors.

Consider the type for the argument of the current ι constructor: (γ : ⟦ Γ ⟧) → ⟦ I ⟧. A value of this type gives an index of type ⟦ I ⟧ under a given environment. The environment γ contains both the values for the datatype parameters and for other variables in the constructor. If indices could depend on the parameters, the result type (currently ⟦ I ⟧) should depend on the parameter part of γ. Other local variables must not be used to determine the index type, because the types of the indices in datatypes are declared in the signature (before the where) where only the parameters can be used. Right now, there is no way to just take the parameter part of an environment.

The fundamental problem here is that parameter types are in the same Cx as the *internal* contexts that contain earlier arguments in the current constructor. Internal contexts can not be used everywhere where the parameter types can be used, but obtaining subsets of environments is not a trivial problem. The choice to have internal contexts build upon the Cx from the parameters seemed reasonable at the time because it allows the local parts of the contexts (i.e. arguments) to use the parameters. For many purposes it works well, but this approach is not suited when indices must depend on parameters.

There is a promising solution to this problem. Descriptions can have a separate Cx just for the parameters, let us call it (P : Cx), and the indices and internal contexts take the form of functions form ⟦ P ⟧ to Cx. One might say: indices and internal contexts are both *contexts under the parameter environment*, meaning that the parameters can be used to determine these contexts.

Descriptions using such a separate parameter context are defined in listing 5.12. The P and I are module parameters because they stay constant within the whole description, consistent with how the declared parameters and indices of real datatypes are the same throughout the datatype definition. For all practical purposes P and I work as if they were datatype parameters for both ConDesc and DatDesc. Places where an internal environment could be used (the functions which had (γ : ⟦ Γ ⟧) as input) can now use both the parameter values (p : ⟦ P ⟧) and the environment (γ : ⟦ Γ p ⟧). When an index has to be specified, the type to be given is ⟦ I p ⟧, so the type of the indices can depend on the parameter values.

The DatDesc type does not pass a context around, but it starts every ConDesc off with an empty context (const ε). Note how similar this is to the descriptions of chapter 4 (Listing 4.3). Once again, constructor descriptions have their own environments which datatype descriptions do not need and a full constructor is always closed, in the sense that Γ is const ε.

```
module _ (P : Cx) (I : (p : ⟦ P ⟧) → Cx) where
  data ConDesc (Γ : (p : ⟦ P ⟧) → Cx) : Set where
    ι : (o : (p : ⟦ P ⟧) (γ : ⟦ Γ p ⟧) → ⟦ I p ⟧) → ConDesc Γ
    _⊗_ : (S : (p : ⟦ P ⟧) (γ : ⟦ Γ p ⟧) → Set) →
      (xs : ConDesc (λ p → Γ p ▷ S p)) → ConDesc Γ
    rec_⊗_ : (i : (p : ⟦ P ⟧) (γ : ⟦ Γ p ⟧) → ⟦ I p ⟧) →
      (xs : ConDesc Γ) → ConDesc Γ
  data DatDesc : (#c : Nat) → Set where
    ‘0 : DatDesc 0
    _⊕_ : ∀ {#c} (x : ConDesc (const ε)) →
      (xs : DatDesc #c) → DatDesc (suc #c)
```

Listing 5.12: Descriptions with separate parameters

```
⟦_⟧conDesc : ∀ {P I Γ} → ConDesc P I Γ →
  (p : ⟦ P ⟧) → (γ : ⟦ Γ p ⟧) → (X : ⟦ I p ⟧ → Set) → ((o : ⟦ I p ⟧) → Set)
⟦ ι o ⟧conDesc p γ X o′ = o p γ ≡ o′
⟦ S ⊗ xs ⟧conDesc p γ X o = Σ (S p γ) λ s → ⟦ xs ⟧conDesc p (γ , s) X o
⟦ rec i ⊗ xs ⟧conDesc p γ X o = X (i p γ) × ⟦ xs ⟧conDesc p γ X o
⟦_⟧datDesc : ∀ {P I #c} → DatDesc P I #c →
  (p : ⟦ P ⟧) → (X : ⟦ I p ⟧ → Set) → ((o : ⟦ I p ⟧) → Set)
⟦_⟧datDesc D p X o = Σ (Fin _) λ k → ⟦ lookupCtor D k ⟧conDesc p tt X o

data μ {P I #c} (D : DatDesc P I #c) (p : ⟦ P ⟧) (o : ⟦ I p ⟧) : Set where
  ⟨_⟩ : ⟦ D ⟧ p (μ D p) o → μ D p o
```

Listing 5.13: Semantics of descriptions with separate parameters

The semantics in listing 5.13 show how descriptions with separate parameters are interpreted. It is a straightforward derivation from the semantics in listing 5.3 and listing 4.4. While the interpretation of ConDesc requires parameter values (p : ⟦ P ⟧) and a local environment (γ : ⟦ Γ p ⟧), the interpretation of DatDesc does not need a local environment. Notice how both result in an endofunctor on ⟦ I p ⟧ → Set, of which μ is the fixpoint.

The Silly datatype of the beginning of this section can now be described. The index type uses top p to refer to the parameter of type Nat. Argument types and indices can be specified using both the parameter values p and the local environment γ.

```
SillyDesc : DatDesc (ε ▷′ Nat) (λ p → ε ▷′ Fin (top p)) 1
SillyDesc = (λ p γ → Fin (top p)) ⊗ (λ p γ → Bool)
  ⊗ ι (λ p γ → tt , top (pop γ)) ⊕ ‘0

silly-test : μ SillyDesc (tt , 10) (tt , 3)
silly-test = ⟨ 0 , 3 , true , refl ⟩
```

Another interesting, less silly, datatype which can be described is the equality type. The index of our equality datatype MyEq uses the value A from the parameters to determine its type. The description EqDesc is quite simple, and the embedding-projection pair is given to show that it is correct.

```
data MyEq {A : Set} (x : A) : A → Set where
  refl : MyEq x x

EqDesc : DatDesc (ε ▷′ Set ▷ top) (λ p → ε ▷′ top (pop p)) 1
EqDesc = ι (λ p γ → tt , top p) ⊕ ‘0

to-eq : ∀ {A x y} → MyEq x y → μ EqDesc ((tt , A) , x) (tt , y)
to-eq refl = ⟨ 0 , refl ⟩
from-eq : ∀ {A x y} → μ EqDesc ((tt , A) , x) (tt , y) → MyEq x y
from-eq ⟨ zero , refl ⟩ = refl
from-eq ⟨ suc () , _ ⟩
```

This way of encoding parameters separately from contexts seems to be a better approximation of Agda datatypes than the descriptions of listing 5.1. This particular encoding was found in the late stages of writing this thesis, so no further efforts have been made regarding the implementation of ornaments and related functionality. For future research, this encoding might be promising. It would be interesting to see if everything works out.

Chapter 6

Generic programming with descriptions

The previous chapter defined descriptions and ornaments with all the core features we wish to have. In this chapter, the last minor changes are made to descriptions to make them store names of arguments. The surrounding functionality as introduced in chapter 2 is presented to get a true generic programming library which allows the derivation of descriptions and embedding-projection pairs from user-defined datatypes.

One of the major goals of this thesis is to allow quoting of datatypes. We use the term *quoting* in general for the conversion of *code* to *data*. More concretely, actual definitions and terms in your Agda program may be *quoted* to representations. In the case of the quoting of datatypes, it primarily means that a description is being calculated for a user-defined datatype.

Once a datatype has been quoted, you may want to derive an embedding-projection pair to translate between the original type and the representation. The term 'derive' is used in the way that it is in Haskell, where certain record instances like Show, Read and Generic can be automatically derived from datatypes by the deriving keyword. We will be using the deriveHasDesc function to perform a similar process:

```
deriveHasDesc : (‘quotedDesc ‘hasDesc ‘dt : Name) → TC ⊤
```

The result of deriveHasDesc is a *meta-program*, contained in the TC monad. The TC monad is built into Agda, and meta-programs within TC can be run by using keywords like unquoteDecl. The meta-program can access types in the context, define new functions, perform unification of types, normalise types, and more. Essentially, it is a way to directly control the type-checker. A meta-program is run during the type-checking at the exact point where it was called, and type-checking will only continue once the result has been computed. Type errors can occur during the execution, for instance because one tries to unify two types which can not be unified, or because an error is thrown manually.

The deriveHasDesc function requires three values of the Name type. The Name type is also built-in, and is a reference to a definition in the program. *Every* Name is directly connected to a function, datatype, record or other kind of definition. Agda makes sure that this is always the case[1]. We will be using two ways to create Names:

[1] Within a TC computation, new Names can be created which are not necessarily bound to a definition. There is, however, no way for these Names to escape the TC monad.

- With the quote keyword, a Name is given for an existing definition. So the expression quote Vec results in a Name which refers to the Vec datatype. The same notation with the quote keyword is used to *show* names as well.
- A statement like unquoteDecl x_1 x_2 ⋯ x_n = m. The expression m must be of type TC ⊤, and *must* declare functions with the names x_1 ⋯ x_n. Within m, these names are of type Name. After the unquoteDecl statement n new definitions have been created.

Our deriveHasDesc function is used in combination with the unquoteDecl keyword, such that 'quotedDesc and 'hasDesc are functions which must be declared by deriveHasDesc. The 'dt argument is the name of the datatype which must be quoted. This is most easily explained with an example—Assume that the Vec datatype has been defined as follows:

```
data Vec (A : Set) : Nat → Set where
  nil : Vec A 0
  cons : ∀ n → (x : A) → (xs : Vec A n) → Vec A (suc n)
```

Remark 6.1. *The n argument of cons is visible, not hidden as it usually is. Hidden arguments are currently not supported by the library. This is why the constructor is named cons instead of _::_. With 3 visible arguments, the infix notation would not give the intended result.*

We use deriveHasDesc on the Vec datatype and run the meta-program by using unquoteDecl. This process defines two functions for us: quotedVec and VecHasDesc. If the name does not match a datatype, or if the datatype can not be described by our descriptions, a type error is thrown.

```
-- Quote the Vec datatype
unquoteDecl quotedVec VecHasDesc =
  deriveHasDesc quotedVec VecHasDesc (quote Vec)
-- Two new functions have been defined:
-- quotedVec : QuotedDesc
-- VecHasDesc : {A : Set} {n : Nat} → HasDesc (Vec A n)
```

The quotedVec function is of type QuotedDesc. It contains, among other things, the generated description and will be defined in section 6.2. The VecHasDesc function returns a HasDesc (Vec A n) for any A and n. The HasDesc record contains the derived embedding-projection pair and is further explained in section 6.3.

The execution of deriveHasDesc on a datatype Dt will often be called *the quoting of Dt*. So when we talk about 'after Vec has been quoted', we mean after deriveHasDesc has been executed by unquoteDecl like in the code above. By convention, we will always use names like quotedDt and DtHasVec for the results of the quoting of a specific datatype Dt.

6.1 Descriptions and ornaments

When quoting datatypes, the library can see what the names of arguments are within the constructors. A fairly small change to descriptions allows each argument to contain such

```
data ConDesc (I : Cx) (Γ : Cx) : Set₁ where
  ι : (o : (γ : ⟦ Γ ⟧) → ⟦ I ⟧) → ConDesc I Γ
  _/_⊗_ : (nm : String) → (S : (γ : ⟦ Γ ⟧) → Set) →
    (xs : ConDesc I (Γ ▷ S)) → ConDesc I Γ
  _/rec_⊗_ : (nm : String) → (i : (γ : ⟦ Γ ⟧) → ⟦ I ⟧) →
    (xs : ConDesc I Γ) → ConDesc I Γ
data DatDesc (I : Cx) (Γ : Cx) : (#c : Nat) → Set₁ where
  ‘0 : DatDesc I Γ 0
  _⊕_ : ∀ {#c} → (x : ConDesc I Γ) →
    (xs : DatDesc I Γ #c) → DatDesc I Γ (suc #c)
```

Listing 6.1: Descriptions with names

a name, of type String. This is the *only* change relative to the descriptions of chapter 5. The full definition of descriptions is given in listing 6.1. The argument names have been added in front of _⊗_ and rec_⊗_, separated by a forward slash. The rest of the definition and semantics are exactly like in section 5.1. The Vec type can now be described in the following way:

```
vecDesc : DatDesc (ε ▷′ Nat) (ε ▷′ Set) 2
vecDesc = ι (const (tt , 0))
  ⊕ "n" / const Nat ⊗
    "x" / top ∘ pop ⊗
    "xs" /rec (λ γ → tt , top (pop γ)) ⊗
    ι (λ γ → tt , suc (top (pop γ)))
  ⊕ ‘0
```

Ornaments are changed accordingly. The copying operators _/-⊗_ and _/rec_⊗_ require a name, which will overwrite the old name of the argument. The insertion operators _/_+⊗_ and _/rec_+⊗_ need a name as well for the argument being inserted. The names are the only change compared to the ornaments in section 5.2. The new definition of ornaments is in listing 6.2.

The ornToDesc function has been slightly updated to make sure that the description gets the names as specified in the ornament. Some other functions, like forget simply ignore the names. All the changes required to support the new descriptions and ornaments with names are trivial, and they will not be listed here.

6.2 Quoting datatypes

The quoting of a datatype gives a DatDesc I Γ #c for *some* I, Γ and #c which are not known in advance. Additionally, the name of the datatype and a list of names of the constructors can be read during the quoting operation, so we would like to store them as well. The QuotedDesc record (Listing 6.3) can contain all the information which we can extract from a datatype definition including the indices, parameters, constructor count, names and description.

The Name type has been used to store the names of constructors and of the datatype. As explained in the introduction of this chapter, this means that ‘datatypeName is connected to a real datatype, and each of the ‘constructorNames is tied to a real

```
data Orn {I} J (u : Cxf J I)
  {Γ} Δ (c : Cxf Δ Γ) : ∀ {dt} (D : Desc I Γ dt) → Set₁ where
  ι : ∀ {i} → (j : (δ : ⟦ Δ ⟧) → u ⁻¹ (i (c δ))) → Orn _ u _ c (ι i)
  _/-⊗_ : ∀ {nm S xs} (nm′ : String) →
    (xs⁺ : Orn _ u _ (cxf-both c) xs) → Orn _ u _ c (nm / S ⊗ xs)
  _/rec_⊗_ : ∀ {nm i xs} (nm′ : String) → (j : (δ : ⟦ Δ ⟧) → u ⁻¹ (i (c δ))) →
    (xs⁺ : Orn _ u _ c xs) → Orn _ u _ c (nm /rec i ⊗ xs)

  _/_+⊗_ : {xs : ConDesc I Γ} (nm : String) (S : (δ : ⟦ Δ ⟧) → Set)
    (xs⁺ : Orn _ u _ (cxf-forget c S) xs) → Orn _ u _ c xs
  _/rec_+⊗_ : {xs : ConDesc I Γ} (nm : String) (j : (δ : ⟦ Δ ⟧) → ⟦ J ⟧)
    (xs⁺ : Orn _ u _ c xs) → Orn _ u _ c xs
  give-K : ∀ {S xs nm} → (s : (δ : ⟦ Δ ⟧) → S (c δ)) →
    (xs⁺ : Orn _ u _ (cxf-inst c s) xs) → Orn _ u _ c (nm / S ⊗ xs)

  ‘0 : Orn _ u _ c ‘0
  _⊕_ : ∀ {#c x} {xs : DatDesc I Γ #c}
    (x⁺ : Orn _ u _ c x) (xs⁺ : Orn _ u _ c xs) → Orn _ u _ c (x ⊕ xs)
```

Listing 6.2: Ornaments with names

```
record QuotedDesc : Set₂ where
  constructor mk
  field
    {I} : Cx
    {Γ} : Cx
    {#c} : Nat
    ‘datatypeName : Name
    ‘constructorNames : Vec Name #c
    desc : DatDesc I Γ #c
```

Listing 6.3: Quoted descriptions

constructor[2].

One may note that datatype/constructor names and argument names are handled very differently. Argument names are not bound to definitions—they are always merely a String. It is still easy to write a new description by hand. If the programmer does not have a name for an argument they can always resort to writing " _". If constructor names were included in DatDesc, which is definitely possible, one would not be able to write new descriptions without needing to grab a Name from somewhere. A newly written description is not bound to a real datatype, so it does not make sense to have to connect the constructors to definitions. A quotedDesc *is* bound to a real datatype by the Names of the datatype and constructors.

The quoting of a datatype (by the use of deriveHasDesc) will result in a QuotedDesc being defined. As an example we quote the Vec datatype, just like in the introduction of this chapter. We can verify that (quotedVec : QuotedDesc) is correct with a simple equality, and we note that the datatype name and the constructor names match with those of Vec and that the description contained in the QuotedDesc is exactly the vecDesc of

[2] Strictly speaking, these names could be connected to any definition. So ‘datatypeName could just as well be the name of a function.

```
record HasDesc (A : Set) : Set₂ where
  constructor mk
  field
    {I Γ} : Cx
    {#c} : Nat
    desc : DatDesc I Γ #c
    {γ} : ⟦ Γ ⟧
    {i} : ⟦ I ⟧
    to′ : A → μ desc γ i
    from′ : μ desc γ i → A
```

Listing 6.4: HasDesc definition

the previous section. The following code will work whenever the Vec datatype has been defined and our library module has been opened.

```
unquoteDecl quotedVec VecHasDesc =
  deriveHasDesc quotedVec VecHasDesc (quote Vec)
quotedVec-check : quotedVec ≡
  mk (quote Vec) (quote Vec.nil :: quote Vec.cons :: []) vecDesc
quotedVec-check = refl
```

Alternatively, the desc field of the QuotedDesc record can be extracted by the function QuotedDesc.desc:

```
vecDesc-check : QuotedDesc.desc quotedVec ≡ vecDesc
vecDesc-check = refl
```

6.3 Deriving an embedding-projection pair

No generic programming framework is complete without having some way to derive an embedding-projection pair for a given datatype. Looking at the type of for instance the vec-to function below, we see that it is parameterised over the parameters (A) and indices (n) of the datatype. In a sense, what we called an embedding-projection pair is actually a *family* of embedding-projection pairs (similar to how Vec is a family of types), with a family member for each combination of parameters and indices.

```
vec-to : ∀ {A n} → Vec A n → μ vecDesc (tt , A) (tt , n)
vec-from : ∀ {A n} → μ vecDesc (tt , A) (tt , n) → Vec A n
```

The HasDesc record, as defined in listing 6.4, can contain one member of the family of embedding-projection pairs. It has a type parameter A and the contained pair converts values between A and μ desc γ i. The fields desc, γ and i together represent a fully applied type, so the type A must be fully applied as well. One could have a HasDesc (Vec Nat 10) or a HasDesc (Fin 7), but HasDesc Vec is not a correct type.

To fully cover the use cases of the family of embedding-projections defined by vec-to and vec-from, one would have to define a family of HasDesc records. The signature of the family of HasDescs for Vec is straightforward, simply parameterise by the A and n:

```
instance
  VecHasDesc : ∀ {A n} → HasDesc (Vec A n)
```

This is exactly the signature of the definition that is generated by the quoting of Vec. The instance keyword allows the VecHasDesc definition to be used for instance searching. We are effectively treating HasDesc as a Haskell typeclass [9], and VecHasDesc provides a HasDesc instance for Vec A n. If a function requires an instance argument ⦃ r : HasDesc B ⦄, Agda will consider VecHasDesc when trying to build a record of type HasDesc B. Of course, VecHasDesc will only be able to return a result of the right type if B is Vec A n for some A and n.

Outside of the record, HasDesc.to′ and HasDesc.from′ are of type {A : Set} (r : HasDesc A) → ⋯. They require a HasDesc record to be passed explicitly. We expect HasDesc records to be defined as instances, so we would be better off by using instance search for these functions. The functions to and from are the versions of to′ and from′ which use instance search to find the right HasDesc record[3]:

```
to : {A : Set} ⦃r : HasDesc A⦄ →
  A → μ (HasDesc.desc r) (HasDesc.γ r) (HasDesc.i r)
to ⦃r⦄ = HasDesc.to′ r
from : {A : Set} ⦃r : HasDesc A⦄ →
  μ (HasDesc.desc r) (HasDesc.γ r) (HasDesc.i r) → A
from ⦃r⦄ = HasDesc.from′ r
```

Any time after Vec has been quoted, one may use to or from for Vecs and the right HasDesc will be found automatically. The definition of vec-to and vec-from thus becomes trivial:

```
vec-to : ∀ {A n} → Vec A n → μ vecDesc (tt , A) (tt , n)
vec-to = to
vec-from : ∀ {A n} → μ vecDesc (tt , A) (tt , n) → Vec A n
vec-from = from
```

6.4 Generic functions

Now that embedding-projection pairs are readily available in their HasDesc instances, generic programming with actual datatypes becomes possible. A typical example is the fold function. Remember the signature of fold in listing 5.6:

```
fold : ∀ {I Γ #c} {D : DatDesc I Γ #c} {γ X}
  (α : Alg D γ X) → μ D γ →ⁱ X
```

One may expand the _→ⁱ_ and reorder some variables to get the following equivalent signature:

```
fold : ∀ {I Γ #c γ i} {desc : DatDesc I Γ #c} → ∀ {X} →
  (α : Alg desc γ X) → μ D γ i → X i
```

The μ D γ i which goes in is the generic representation of some type. If we want to define a gfold which works for real types, the μ D γ i must be replaced by some A. The

[3] The same effect could be achieved by open HasDesc ⦃...⦄ with the proper qualifiers.

A is required to be representable with a description, so a HasDesc A is expected. This HasDesc contains all the values for I, Γ, #c, desc, γ and i—all these variables can be removed from the signature. We end up with the following signature for gfold, where the notation var_R is used to take the field var from the record R[4].

```
gfold : ∀ { A } ⦃ R : HasDesc A ⦄ → ∀ { X } →
  Alg (desc_R) (γ_R) X → A → X (i_R)
gfold α = fold α ∘ to
```

Now gfold can be used to calculate, for instance, the sum of a Vec. If we assume that some algebra vecSumAlg exists, it can simply be gfolded over a Vec and the HasDesc record is found automatically.

```
vecSumAlg : Alg vecDesc (tt , Nat) (λ i → Nat)
vecSumAlg = ⋯
vec-example : Vec Nat 4
vec-example = cons _ 3 (cons _ 1 (cons _ 5 (cons _ 6 nil)))
vec-example-sum : gfold vecSumAlg vec-example ≡ 15
vec-example-sum = refl
```

Other functions on descriptions can be transformed to generic functions as well, including some which work with ornaments. The gforget function is a version of forget which works on datatypes which have been quoted. Let us quote the List datatype and create a list→vec ornament:

```
unquoteDecl quotedList ListHasDesc =
  deriveHasDesc quotedList ListHasDesc (quote List)
listDesc : DatDesc ε (ε ▷′ Set) 2
listDesc = QuotedDesc.desc quotedList
list→vec : Orn (ε ▷′ Nat) (λ i → tt) (ε ▷′ Set) id listDesc
list→vec = ι (λ δ → inv (tt , 0))
  ⊕ "n" / const Nat +⊗
    "x" /-⊗
    "xs" /rec (λ δ → inv (tt , top (pop δ))) ⊗
    ι (λ δ → inv (tt , suc (top (pop δ))))
  ⊕ ‘0
```

The forget function for the list→vec ornament goes from μ vecDesc (tt , A) (tt , n) to μ listDesc (tt , A) tt. With the right definition of gforget, forget can be used to transform a Vec A n into a List A. The following works:

```
vec-example-forget :
  gforget list→vec vec-example ≡ 3 ∷ 1 ∷ 5 ∷ 6 ∷ []
vec-example-forget = refl
```

The signature gforget is rather unwieldy and uncovers some problems with the current structure of the records. This problem and how a solution would improve the type of gforget is discussed in section 6.7.

[4] In Agda, var_R can be written as var R *if* the HasDesc module has been opened. Without opening the module one must write HasDesc.var R.

6.5 Unquoting descriptions

The functionality defined in section 6.2, section 6.3 and section 6.4 is similar to what generic deriving [15] does for Haskell. A description is calculated for a given datatype and an embedding-projection pair is generated. Generic functions like gfold and gdepth can be implemented. Our descriptions are carefully engineered to *always* be convertible to a real datatype, which is what we will do in this section.

The process of generating a datatype based on a description will be called *unquoting*. Agda (version 2.5.1) does not yet support the declaration of datatypes from the reflection mechanism, so it can not be fully automated. We *can* write the skeleton of a datatype, and unquote the types of the constructors and of the datatype, as Pierre-Evariste Dagand pointed out to me. So if one has a finDesc of type DatDesc (ε ▷′ Nat) ε 2 which describes the Fin type, the user would at least have to write the following:

```
data Fin : ··· where
  zero : ···
  suc : ···
```

Two macros are responsible for generating the types for the ...'s: unquoteDat and unquoteCon:

```
macro
  unquoteDat : {I Γ #c} (D : DatDesc I Γ #c) → Tactic
  unquoteCon : {I Γ #c} (D : DatDesc I Γ #c) →
    (k : Fin #c) → (‘self : Term) → Tactic
```

Macros result in a Tactic which is executed at the spot where the macro is called. The tactic *must* place a value in that same spot. This means that unquoteDat finDesc is not of type Tactic, but it is the *result* of that tactic—in the case of unquoteDat finDesc the result is Set of type Set_1. With these macros the following definition of Fin can be built:

```
data Fin : unquoteDat finDesc where
  zero : unquoteCon finDesc 0 Fin
  suc : unquoteCon finDesc 1 Fin
```

Now we have the Fin datatype and a description finDesc, but no HasDesc record connecting the two. There is no QuotedDesc record for this datatype either. The usual quoting operation deriveHasDesc can create these records, but it calculates a description as well. We want to make use of the description that is already available, and for this purpose deriveHasDescExisting has been implemented. It is similar to deriveHasDesc, but takes an additional description and will ensure that it matches with the generated description. If it does not, an error will occur. Figure 6.1 shows how deriveHasDescExisting fits in. After the following call to deriveHasDescExisting, the quotedFin and FinHasDesc functions will be defined:

```
unquoteDecl quotedFin FinHasDesc =
  deriveHasDescExisting quotedFin FinHasDesc
  (quote Fin) finDesc
```

We have now used a description to first unquote a datatype semi-automatically. After that, we derived the QuotedDesc and HasDesc records. Ideally, one would merge

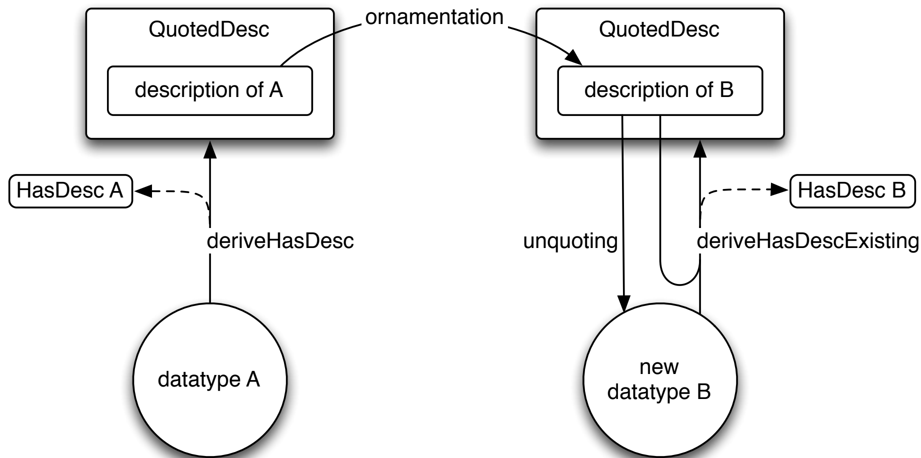

Figure 6.1: The process of quoting and unquoting

these operations into a single call (it would make fig. 6.1 a lot prettier), but that is not possible in the current version of Agda (2.5.1). Even if the unquoting of datatypes were possible, one would still need to give the names for all the constructors. If tactics would support the definition of datatypes, but the existing unquoteDecl would have to be used, it might look as follows:

```
-- Speculative:
unquoteDecl quotedFin FinHasDesc Fin zero suc =
  unquoteDatatype quotedFin FinHasDesc
  finDesc Fin (zero ∷ suc ∷ [])
```

6.6 Higher-level ornaments

Writing ornaments with the Orn datatypes is verbose and requires a decent understanding of how descriptions work. The ornaments do not do a particularly good job of communicating ideas like 'add a parameter' or 'rename these constructors'. For instance, it is not obvious at first sight that the following ornament only renames 'x' to 'y' and 'xs' to 'ys':

```
list-rename₁ : Orn ε id (ε ▷′ Set) id listDesc
list-rename₁ = ι (λ δ → inv tt)
  ⊕ "y" /-⊗
    "ys" /rec (λ δ → inv tt) ⊗
    ι (λ δ → inv tt)
  ⊕ ‘0
```

The Orn datatype provides a good low-level language which guarantees that the ornament induces a forget function. For actual programming, higher-level abstractions may be easier to work with. These abstractions take the form of functions that generate ornaments, and we have already seen algebraic ornaments as an example. In this section we give some more examples of operations like that. We try to bring ornaments closer to how programmers think about the relations between datatypes.

6.6.1 Structure-preserving ornaments

To start with, we will talk about ornaments which preserve the *structure* of the description. That is, it keeps all the ι's, _⊗_'s and rec_⊗_'s in the same place. The most obvious example of such an ornament is the identity ornament, which does nothing. A more general version allows changes to parameters and indices. We define it as reCx:

```
reCx : ∀ {I J u Γ Δ c dt} {D : Desc I Γ dt} →
  (f : ∀ i → u ⁻¹ i) → Orn J u Δ c D
reCx {c = c} {isCon} {ι o} f = ι (f ∘ o ∘ c)
reCx {c = _} {isCon} {nm / S ⊗ xs} f = nm /-⊗ reCx f
reCx {c = c} {isCon} {nm /rec i ⊗ xs} f = nm /rec f ∘ i ∘ c ⊗ reCx f
reCx {c = _} {isDat _} {‘0} f = ‘0
reCx {c = _} {isDat _} {x ⊕ xs} f = reCx f ⊕ reCx f
```

For every ornament, a copy ornament is created which updates the indices and context. In the case for ι, an index has to be given using an ornamented environment. We get an unornamented environment with c, see what the old index is under that environment with o and use f to convert an old index into a new index. Intuitively, the function f should be seen as a function of type ⟦ I ⟧ → ⟦ J ⟧, with the extra requirement that u is the inverse of this function.

The reCx function can be specialised in three ways: by only allowing updates to indices, by only allowing updates to the context/parameters or by not allowing either. This gives the functions reindex, reparam and idOrn:

```
reindex : ∀ {I J u Γ dt} {D : Desc I Γ dt} →
  (f : ∀ i → u ⁻¹ i) → Orn J u Γ id D
reindex = reCx

reparam : ∀ {I Γ Δ c dt} {D : Desc I Γ dt} → Orn I id Δ c D
reparam = reCx inv

idOrn : ∀ {I Γ dt} {D : Desc I Γ dt} → Orn I id Γ id D
idOrn = reCx inv
```

6.6.2 Ornament composition

Ornament composition is defined as _≫⁺_. The function _≫⁺_ takes two ornaments and results in a new ornament which combines the two:

```
module _ {I J J′} {u : Cxf J I} {v : Cxf J′ J} where
  _≫⁺_ : ∀ {Γ Δ Δ′ c d dt} {D : Desc I Γ dt} →
    (o : Orn J u Δ c D) → Orn J′ v Δ′ d (ornToDesc o) →
    Orn J′ (u ∘ v) Δ′ (c ∘ d) D
```

For the definition of _≫⁺_, a case split is done on both ornaments. The first ornament o determines what the input for the second ornament is, which limits the number of cases to a workable amount. For instance, if o is a ι copy ornament, the input for the second ornament must be a ι so only another ι copy ornament or an insertion ornament can occur:

```
_≫⁺_ (ι j) (ι k) = ι (λ _ → inv-∘ (j _) (k _))
_≫⁺_ (ι j) (nm / T +⊗ ys⁺) = nm / T +⊗ (_≫⁺_ (ι j) ys⁺)
_≫⁺_ (ι j) (nm /rec k +⊗ ys⁺) = nm /rec k +⊗ (_≫⁺_ (ι j) ys⁺)
```

The full definition of composition is quite long and very straightforward, so it is not listed here. To prove that composition of ornaments is correctly defined, ≫⁺-coherence says that ornToDesc of the composed ornament is the same as ornToDesc of the second ornament, which in turn is an ornament on ornToDesc of the first ornament. The descriptions contain higher order terms (terms depending on environments) which are not intensionally equal. We can however prove that they are pointwise equal, for each environment they give the same result. A small module is used wherein the extensionality axiom ((∀ x → f x ≡ g x) → f ≡ g) is available, effectively making the normal equality _≡_ extensional (within the module).

```
module _ (ext : ∀ {a b} → Extensionality a b) where
  ≫⁺-coherence : ∀ {Γ Δ Δ′ c d dt} {D : Desc I Γ dt} →
    (o : Orn J u Δ c D) → (p : Orn J′ v Δ′ d (ornToDesc o)) →
    (ornToDesc (o ≫⁺ p)) ≡ ornToDesc p
```

6.6.3 More ornaments

Programmers may only want to ornament one of the constructors of a datatype. This idea is expressed by updateConstructor. The programmer can specify which of the constructors to update with a Fin #c, and must only give an ornament for that constructor. The identity ornament is used for the rest of the constructors.

```
updateConstructor : ∀ {I Γ #c} {D : DatDesc I Γ #c} →
  (k : Fin #c) → Orn I id Γ id (lookupCtor D k) →
  Orn I id Γ id D
updateConstructor {D = ‘0} () o
updateConstructor {D = x ⊕ xs} zero o = o ⊕ idOrn
updateConstructor {D = x ⊕ xs} (suc k) o =
  idOrn ⊕ updateConstructor k o
```

The ornament from Nat to List adds a type parameter, and then inserts an argument of that type in the suc constructor. The addParameterArg ornament does exactly that. The new parameter is specified by the (Γ ▷′ Set) in the type, and reparam is used to modify the whole description to work with the new context, without really using the newly added type. After that, by using composition, updateConstructor inserts one new argument in the kth constructor. Because the argument is added at the start of the constructor, the parameter can be referred to with top. It is currently hard to insert the argument somewhere else in the constructor, but it would probably be easy when the separation of parameters discussed in section 5.4.1 is implemented.

```
addParameterArg : ∀ {I Γ #c} {D : DatDesc I Γ #c} →
  Fin #c → String → Orn I id (Γ ▷′ Set) pop D
addParameterArg k str = reparam
  ≫⁺ updateConstructor k (str / top +⊗ reparam)
```

Another common operation when ornamenting datatypes is the renaming of arguments. While the names do not influence the functioning of a datatype, they will be visible when a datatype is unquoted. The renaming of arguments in a specific constructor is done by the renameArguments function. The user picks a constructor with (k : Fin #c) and gives a list of Maybe String's, one for each argument in the constructor.

If a Nothing is given for an argument, the old name is kept. The conRenameArguments function is a variant which works directly on a constructor.

```
renameArguments : ∀ {I Γ #c} {D : DatDesc I Γ #c} →
  (k : Fin #c) →
  Vec (Maybe String) (countArguments (lookupCtor D k)) →
  Orn I id Γ id D
conRenameArguments : ∀ {I Γ} {D : ConDesc I Γ} →
  Vec (Maybe String) (countArguments D) →
  Orn I id Γ id D
```

These are some examples of functions that create ornaments. Small components like idOrn, reparam, reindex, reCx, renameArguments and updateConstructor can be combined easily. Chapter 2 already showed an example of what nat→list could look like:

```
nat→list′ : Orn _ _ _ _ natDesc
nat→list′ = renameArguments 1 (just "xs" :: [])
  >>+ addParameterArg 1 "x"
```

This definitely does a better job at communicating the meaning of the changes than the low-level ornament:

```
nat→list : Orn ε id (ε ▷′ Set) (λ _ → tt) natDesc
nat→list = ι (λ δ → inv tt)
  ⊕ "x" / top +⊗
    "xs" /rec (λ δ → inv tt) ⊗
    ι (λ δ → inv tt)
  ⊕ '0
```

These are two extremes, where the former is very high-level and the later is very low-level. There is nothing wrong with something in between:

```
nat→list″ : Orn ε id (ε ▷′ Set) (λ _ → tt) natDesc
nat→list″ = reparam
  ⊕ "x" / top +⊗
  (reparam >>+ conRenameArguments (just "xs" :: []))
  ⊕ '0
```

6.6.4 Reornaments

The *ornamental algebra* of an ornament is an algebra that forgets the extra information introduced by the ornament. So the nat→list ornament induced a length algebra. *Algebraic ornaments* (section 5.3) used an algebra to create an ornament that added the results of the algebra as an index. For instance, the length algebra for lists could be used to obtain vectors. The first creates an algebra from an ornament, while the second creates an ornament from an algebra. These can be combined to create *reornaments*[19].

The reornament function implements reornamentation using composition and the algebraic ornament of the ornamental algebra. An index is added which can contain elements of the original description (μ D tt (u j)). No parameters are allowed for the original description, so the environment can be instantiated with tt.

```
reornament : ∀ {I J u Δ} {c : Cxf Δ ε} {#c} {D : DatDesc I ε #c} →
  (o : Orn J u Δ c D) → Orn (J ▷ μ D tt ∘ u) (u ∘ pop) Δ c D
reornament o = o >>⁺ (algOrn _ (λ {δ} → forgetAlg o {δ}))
```

Example 6.6.1. We will construct the reornament of nat→list. Let us assume that we have the following definitions for the description of natural numbers, the constructors for that description, and the ornament from natural numbers to lists:

```
natDesc : DatDesc ε ε 2

natDesc-zero : μ natDesc tt tt
natDesc-suc : μ natDesc tt tt → μ natDesc tt tt

nat→list : Orn ε id (ε ▷′ Set) (λ δ → tt) natDesc
```

By applying reornament to nat→list, one obtains a ornament from natural numbers to Vec. Contrary to list→vec from section 6.4, which added a Nat as an index, this one uses a μ natDesc tt tt. These are isomorphic, so it should not be a problem.

```
nat→vecᵣ : Orn (ε ▷′ μ natDesc tt tt) (λ j → tt) (ε ▷′ Set) (λ δ → tt) natDesc
nat→vecᵣ = reornament nat→list
```

The resulting description is very similar to the one created by algOrn lengthAlg. The only differences are that Nat has been replaced with μ natDesc tt tt, 0 with natDesc-zero and suc with natDesc-suc.

```
vecDescᵣ : DatDesc (ε ▷′ μ natDesc tt tt) (ε ▷′ Set) 2
vecDescᵣ = ι (const (tt , natDesc-zero))
  ⊕ "x" / top
  ⊗ "_" / const (μ natDesc tt tt)
  ⊗ "xs" /rec (λ γ → tt , top γ)
  ⊗ ι (λ γ → tt , natDesc-suc (top γ))
  ⊕ ‘0

test-nat→vec : ornToDesc nat→vecᵣ ≡ vecDescᵣ
test-nat→vec = refl
```

△

With the current descriptions, reornaments on descriptions with parameters can not be supported in general. While writing a type for a reornament′ operation which does support it, we get stuck when trying to give the environment for the index. The hole ?0 is of type ⟦ Γ ⟧, an environment for the original description. Such an environment could be built using the ornamented environment of type ⟦ Δ ⟧ and the environment transformer c, but there is no ⟦ Δ ⟧ available in the place of the hole.

```
reornament′ : ∀ {I J u Δ Γ} {c : Cxf Δ Γ} {#c} {D : DatDesc I Γ #c} →
  (o : Orn J u Δ c D) → Orn (J ▷ μ D ?0 ∘ u) (u ∘ pop) Δ c D
```

The problem lies in the fact that descriptions do not allow indices to be dependent on parameters, as was discussed in section 5.4.1. Right now, reornament can not work around it, but this may be possible if the solution proposed in that section was implemented.

6.7 Discussion

This chapter presented the final iteration of descriptions, which is suited to describe a fairly large class of datatypes. We showed how datatypes can be quoted to these descriptions, and how descriptions can be unquoted to datatypes. Some generic functions were defined, which work on actual datatypes once their embedding-projection pairs are derived (which is automatically done when quoting a datatype). Finally, some higher-level ornaments were defined.

With all these components together, we hope to have made the barrier to start working with descriptions and ornaments low enough. A user does not necessarily need to know much about the theory to quote a datatype, make some basic modifications, and unquote it to a new datatype. At the very least, it is easy to figure out what the description for a certain datatype should be by simply quoting it. These abstractions are still leaky—if one does not write everything just right, they will get errors which can not be understood without a deeper understanding of these descriptions and ornaments.

6.7.1 Embedding-projection instances

The HasDesc record is indexed by the represented type A, this allows for easy instance searching *by type*. When one has a value of type A this works well, for instance in the use of to. In the following example, the HasDesc (Vec Nat 4) instance is found which contains the description, γ and i. The type of ?0 is inferred as μ vecDesc (tt , Nat) (tt , 4).

```
vec-example-rep : ?0 -- μ vecDesc (tt , Nat) (tt , 4)
vec-example-rep = to vec-example
```

The other way around is not so easy. If one only knows γ, i and the description itself, Agda can not search for a HasDesc instance. This means that the type of from vec-example-rep, the hole in the following example, can not be inferred. If the result type is given by the user, or known in some other way, that can be used to find the HasDesc. So the following does type check:

```
vec-example′ : Vec Nat 4
vec-example′ = from vec-example-rep
```

While to and from seem to behave the same, this is only the case when all the types are known. The difference is that to uses its input to find the record, and from requires the result type before a record can be found. One of the consequences is that the signature of gforget becomes very complicated:

```
gforget : ∀ {A} ⦃ AR : HasDesc A ⦄ {B} ⦃ BR : HasDesc B ⦄ →
  ∀ {u c} (o : Orn (I_BR) u (Γ_BR) c (desc_AR)) →
  ⦃ ieq : i_AR ≡ u (i_BR) ⦄
  ⦃ γeq : γ_AR ≡ c (γ_BR) ⦄
  ⦃ #ceq : #c_AR ≡ #c_BR ⦄
  ⦃ deq : transport (DatDesc (I_BR) (Γ_BR)) #ceq (ornToDesc o)
    ≡ desc_BR ⦄ →
  B → A
```

The ornament goes from A to B, so the forget function goes from B to A. The HasDesc B instance can be searched for, which is necessary to transform the B into

```
record EmbeddingProjection (A : Set) {I Γ #c}
  (desc : DatDesc I Γ #c) (γ : ⟦ Γ ⟧) (i : ⟦ I ⟧) : Set₂ where
  constructor mk
  field
    to′ : A → μ desc γ i
    from′ : μ desc γ i → A

record Embeddable (A : Set) : Set₂ where
  constructor mk
  field
    {I Γ} : Cx
    {#c} : Nat
    desc : DatDesc I Γ #c
    γ : ⟦ Γ ⟧
    i : ⟦ I ⟧
    ep : EmbeddingProjection A desc γ i

record Projectable {I Γ #c}
  (desc : DatDesc I Γ #c) (γ : ⟦ Γ ⟧) (i : ⟦ I ⟧) : Set₂ where
  constructor mk
  field
    A : Set
    ep : EmbeddingProjection A desc γ i

to : ∀ {A} ⦃ R : Embeddable A ⦄ → A → μ (desc_R) (γ_R) (i_R)
to ⦃ mk desc γ i ep ⦄ = to′ ep

from : ∀ {I Γ #c desc γ i} ⦃ R : Projectable {I} {Γ} {#c} desc γ i ⦄ →
  μ desc γ i → A_R
from ⦃ mk A ep ⦄ = from′ ep
```

Listing 6.5: Alternative embedding-projection records

a μ ($desc_{BR}$) ($γ_{BR}$) (i_{BR}). By ornamenting with o we get a μ (ornToDesc o) (c ($γ_{BR}$)) (u (i_{BR})). In the current implementation, the result type A is used to find a HasDesc A instance, which gives us a way to transform a μ ($desc_{AR}$) ($γ_{AR}$) (i_{AR}) into an A. The types μ (ornToDesc o) (c ($γ_{BR}$)) (u (i_{BR})) and μ ($desc_{AR}$) ($γ_{AR}$) (i_{AR}) do not line up, which is why all the equalities are required.

A solution to this problem is to the HasDesc record into several records. Time did not permit to implement this in the framework, but listing 6.5 shows how it might work. The embedding-projection pair is in a separate record parameterised by A, desc, γ and i. The Embeddable record takes over the role of HasDesc and is suitable for instance search by type, while the Projectable record enables searching by description. Using these records, the to and from functions can be implemented in a way where type information always flows from the input to the output, so the result types of to and from can be inferred.

The signature of the generic forget function becomes a lot simpler. The embedding-projection pair of the result is obtained by searching a Projectable with the calculated environment and indices, so there is no need to check afterwards that the types line up. The ornToDesc o ≡ desc_{BR} equality is still required to make sure that the given ornament matches with the input type B.

```
gforget′ : ∀ { B } ⦃ BR : Embeddable B ⦄ →
  ∀ { AI AΓ Adesc u c } (o : Orn { AI } (I_BR) u { AΓ } (Γ_BR) c Adesc) →
  ⦃ deq : ornToDesc o ≡ desc_BR ⦄ →
  ⦃ AR : Projectable Adesc (c (γ_BR)) (u (i_BR)) ⦄ →
  B → A_AR
```

Chapter 7

Discussion

The structure of our descriptions matches closely with the structure of actual datatype declarations. We have chosen to split them up into constructor descriptions and datatype descriptions, and to have a first-order structure to determine which arguments each constructor has. Functions are only allowed within parts where arbitrary terms could occur in real datatypes. Our descriptions have strict control over what can and what can not be influenced by the context.

Descriptions encode indexed functors that are of the form (I → Set) → (O → Set). There are many ways to encode indexed functors, including ways that build on the Σ-descriptions of section 3.5.1. Indexed containers[2] can also be used, but for our purposes they have the same problems as Σ-descriptions: They can be used to define a lot of exotic types that do not correspond to an Agda datatype. Indexed Σ-descriptions and, even more so, indexed containers serve well as *semantical* models of inductive families, but they do not provide an accurate *syntactical* representation of Agda datatypes.

7.1 Explicit parameter use

In our descriptions, starting with those in chapter 4, 'types within a context' were represented with a function of type ⟦ Γ ⟧ → Set. This allows any type to be represented and the type may depend on a local environment. While this is a very powerful approach if one only cares about representing types, it is not very helpful when the representation needs to be *inspected*. For instance, one can not decide whether a given term uses a certain parameter. More precisely, the following definition of isTop can not be completed. For an arbitrary S of type ⟦ ε ▷′ Set ⟧ → Set, we can neither prove that it is top or that it is not top.

```
data Dec (P : Set) : Set where
  yes : P → Dec P
  no  : ¬ P → Dec P
isTop : (S : ⟦ ε ▷′ Set ⟧ → Set) → Dec (∀ γ → S γ ≡ top γ)
isTop S = ?0
```

This quickly becomes a problem when writing generic functions. A common function in generic programming frameworks is flatten; it takes a value of a type with a type parameter A, and converts it into a List A. Another is the parametric map function pmap

which maps a function (f : A → B) over elements in a structure. With the descriptions of chapter 5, these functions would have the following type:

```
flatten : ∀ { #c } (D : DatDesc ε (ε ▷′ Set) #c) →
  ∀ { A } → μ D (tt , A) tt → List A
pmap : ∀ { #c } (D : DatDesc ε (ε ▷′ Set) #c) →
  ∀ { A B } → (f : A → B) → μ D (tt , A) tt → μ D (tt , B) tt
```

The implementation of both flatten and pmap is impossible with our descriptions, because it can not be decided where parameters are being used. Other generic programming frameworks often do not have this problem, because they have a separate description for parameter use. For instance, a subset of the universe of PolyP (where a single parameter is allowed) can be encoded in Agda as follows [14, 16]:

```
data PolyPDesc : Set where
  ι : PolyPDesc
  rec : PolyPDesc
  par : PolyPDesc
  _⊕_ : (F G : PolyPDesc) → PolyPDesc
  _⊗_ : (F G : PolyPDesc) → PolyPDesc
```

The decoding for this universe is of type PolyPDesc → (P : Set) → (X : Set) → Set, where the decoding of par results in the parameter type P. With simple pattern matching, the usage of the parameter can be detected. This same idea can be made to work for multiple parameters in a Cx. We use Γ ∋Set as proofs that a Set is specified in the context, and $⟦_⟧_{∋Set}$ to lookup the type (a value of type Set) in an environment γ. Note that _∋Set and $⟦_⟧_{∋Set}$ are specifically meant to lookup *types* in the environment. The same can be done to lookup values in the environment, but other definitions are needed [18].

```
data _∋Set : (Γ : Cx) → Set₁ where
  top′ : ∀ { Γ } → (Γ ▷′ Set) ∋Set
  pop′ : ∀ { Γ S } → Γ ∋Set → (Γ ▷ S) ∋Set
⟦_⟧∋Set : ∀ { Γ } → Γ ∋Set → (γ : ⟦ Γ ⟧) → Set
⟦ top′ ⟧∋Set (γ , t) = t
⟦ pop′ i ⟧∋Set (γ , s) = ⟦ i ⟧∋Set γ
```

With these definitions, the PolyP universe can be modified to support multiple parameters. Listing 7.1 defines the descriptions and semantics of the new universe. The semantics are mostly business as usual—the parameters are decoded with $⟦_⟧_{∋Set}$.

A binary tree type, with data of type A in the leaves, is defined as Tree A. The new universe can describe this type, where par top′ is used to refer to the parameter. Note that constructors in this universe do not have to be terminated with a ι. We show that the definition makes sense by defining the tree-to function, to convert real trees into represented trees.

```
data Tree (A : Set) : Set where
  leaf : A → Tree A
  node : Tree A → Tree A → Tree A
```

```
data Desc (Γ : Cx) : Set₁ where
  ι : Desc Γ
  rec : Desc Γ
  par : (i : Γ ∋Set) → Desc Γ
  _⊕_ : Desc Γ → Desc Γ → Desc Γ
  _⊗_ : Desc Γ → Desc Γ → Desc Γ
⟦_⟧desc : ∀ {Γ} → Desc Γ → ⟦ Γ ⟧Cx → Set → Set
⟦ ι ⟧desc γ X = ⊤
⟦ rec ⟧desc γ X = X
⟦ par i ⟧desc γ X = ⟦ i ⟧∋Set γ
⟦ A ⊕ B ⟧desc γ X = Either (⟦ A ⟧desc γ X) (⟦ B ⟧desc γ X)
⟦ A ⊗ B ⟧desc γ X = ⟦ A ⟧desc γ X × ⟦ B ⟧desc γ X
data μ {Γ} (D : Desc Γ) (γ : ⟦ Γ ⟧) : Set where
  ⟨_⟩ : ⟦ D ⟧desc γ (μ D γ) → μ D γ
```

Listing 7.1: Descriptions with parameter lookup

```
treeDesc : Desc (ε ▷′ Set)
treeDesc = par top′ ⊕ rec ⊗ rec
tree-to : ∀ {A} → Tree A → μ treeDesc (tt , A)
tree-to (leaf v) = ⟨ left v ⟩
tree-to (node xs ys) = ⟨ right (tree-to xs , tree-to ys) ⟩
```

So far so good, we can do what we already could in chapter 5. To show that we have indeed made progress, we will define the flatten function, for which parameter use needs to be recognised. Using straightforward definitions for Alg and fold, a flattenAlg algebra can be defined which works for any description. Notice how we can simply pattern match on par top′ to extract the value of type A.

```
flattenAlg : ∀ {Γ} (D : Desc (Γ ▷′ Set)) →
  ∀ {γ A} → Alg D (γ , A) (List A)
flattenAlg ι tt = []
flattenAlg rec x = x
flattenAlg (par top′) x = [ x ]
flattenAlg (par _) x = []
flattenAlg (A ⊕ B) (left x) = flattenAlg A x
flattenAlg (A ⊕ B) (right x) = flattenAlg B x
flattenAlg (A ⊗ B) (x , y) = flattenAlg A x ++ flattenAlg B y
```

Finally, the flatten algebra can be folded over a tree to retrieve a list of all the elements:

```
tree-example : Tree Nat
tree-example = node (leaf 7) (node (node (leaf 5) (leaf 3)) (leaf 1))
test-flatten : fold (flattenAlg treeDesc) (tree-to tree-example)
  ≡ 7 :: 5 :: 3 :: 1 :: []
test-flatten = refl
```

In summary, it is possible to explicitly encode parameter references in descriptions when the parameters are declared with a Cx. Of course, this would be nice to integrate into our descriptions of 5 or 6. There are two things holding us back:

- With the simple universe in this section the use of the last parameter *always* looks like par top′, so a simple pattern match suffices. In our descriptions, contexts do not remain constant but depend on where in a constructor we are. The descriptions of section 5.4.1, where the parameters are separated from internal contexts, do not have this problem.

- The possibility of false negatives. If one introduces a new constructor for parameter argumenst while keeping the old _⊗_ constructor with the ⟦ Γ ⟧ → Set argument, both can be used to encode the use of a parameter. One can then still not say with certainty that an argument is *not* the simple use of a parameter.

When the ability to describe as many types as possible is less important, one could get rid of the old ⟦ Γ ⟧ → Set arguments altogether. Instead, a language of types could be used like McBride's [18]. McBride defines a type-is-representable predicate _⋆_ in the style of Crary et al [5]. The predicate ensures that types are only built using the language as defined in the constructors of _⋆_. It is indexed by a ⟦ Γ ⟧ → Set function, telling us what the expected behavior is. As an example, we define the type language to have three types: natural numbers, sets, and types from the context.

```
data _⋆_ (Γ : Cx) : (⟦ Γ ⟧ → Set) → Set₁ where
  'Nat  : Γ ⋆ const Nat
  'Set  : Γ ⋆ const Set
  'TypeVar : (i : Γ ∋Set) → Γ ⋆ ⟦ i ⟧∋Set
```

Limited experimentation shows that the ⟦ Γ ⟧ → Set function in the _⊗_ constructor of our descriptions can be replaced by Γ ⋆ S. So the constructor type is changed from (S : ⟦ Γ ⟧ → Set) → ⋯ to {S : ⟦ Γ ⟧ → Set} → Γ ⋆ S → ⋯. It remains to be seen how ornaments will work out with such descriptions.

7.2 Induction-recursion and strict positivity

Our descriptions are able to describe a practical subset of the *inductive types*. Dybjer and Setzer [12] describe *ordinary inductive definitions* of types with a finite number of constructors:

```
conᵢ : Φᵢ U → U
```

The Φ_i are strictly positive functors. If dependent types are allowed, strictly positive functors can be constructed by a number of rules (according to Dybjer and Setzer):

- nil: The constant functor Φ X = ⊤ is strictly positive.
- nonind: If A is a type and Ψ_x is a strictly positive functor depending on (a : A), then Φ X = Σ A λ a → Ψ_a X is strictly positive.
- ind: If Ψ is strictly positive, then Φ X = X × Ψ X is strictly positive.

The rules nil, nonind and ind correspond exactly to the semantics of ι, σ and rec-×_ of the Σ-descriptions in listing 3.8 on page 21, while the introduction rule con_i : Φ_i U → U corresponds to the constructor of the fixpoint datatype μ_Σ. We have shown in table 3.1 (section 3.5.1) that our ConDesc/DatDesc universe of that chapter can describe a subset of those Σ-descriptions, so by the rules stated above this means that that universe describes a subset of the ordinary inductive types.

In later chapters we have extended the universe in several ways, but the same logic still holds. One can confirm that the semantics for families of datatypes in listing 5.3 can be generated with similar rules as above, though slightly modified to allow indices. The ConDesc/DatDesc universes with indices describe inductive families.

Note that the ind rule does not allow later arguments to depend on the value of an inductive argument. So within a datatype D, an arguments in a constructor can not depend on earlier arguments of type D. Our ConDesc/DatDesc universes reflect this fact by not including the S of a rec S × xs in the context for xs. We know that Agda datatypes do not have such restrictions—They are not just inductive types.

Inductive-recursive types are a generalisation of inductive types, where a simultaneously defined recursive function of type D → ⋯ can be used within the definition of the type D. A simple example is our Cx type in listing 4.2 on page 27, which is mutually defined with $⟦_⟧_{Cx}$. Dybjer and Setzer [12] have given an axiomatisation of inductive-recursive types that can be implemented in Agda easily.

Compared to inductive types, the pattern functors of inductive-recursive types can use an extra argument T that represents a recursively defined function[1]. Now in the case of an inductive argument a, later arguments can depend on T a (not on *just* a). For example; when describing the type Cx the function T would represent $⟦_⟧_{Cx}$. When (Γ : Cx) is an (inductive) argument, the rest of the arguments could depend on $⟦ Γ ⟧_{Cx}$. This is sufficient to encode the _▷_ constructor, which is of type (Γ : Cx) → ($⟦ Γ ⟧_{Cx}$ → Set) → Cx.

Separately from inductive-recursive types, ordinary inductive types can also be extended in another way–To *generalised* inductive types. Generalised inductive types are the same as ordinary inductive types, but with the inclusion of an inductive premise in the ind rule, giving the following rule:

- ind: If Ψ is strictly positive and A is a type, then Φ X = (A → X) × Ψ X is srictly positive. If one instantiates A to ⊤, one obtains the ordinary inductive types (up to isomorphism).

To summarise, there are three ways to expand on ordinary inductive types:

- By adding indices, so *inductive families* can be described. This was done for our universe in chapter 5.

- By allowing inductive premises, to get *generalised* inductive types.

- By passing a recursive function to the pattern functors, to implement *inductive-recursive* types.

The ConDesc/DatDesc universe has not been adapted to implement the latter two, but this may well be possible. Dybjer and Setzer have presented *indexed inductive-recursive* types[13], combining the combination of these three expansions. Indexed inductive-recursive types are a good approximation of the datatypes that are implemented by Agda. It would be interesting to see if our universes, ornaments, and generic programming framework could be rebuilt with indexed inductive-recursive types as their foundation.

[1] Actually, as Dybjer and Setzer note, the functors are not really functors anymore in the category theory sense of the word.

Chapter 8

Conclusion

In this thesis we have presented a generic programming framework for Agda. Datatypes can be transformed into new datatypes by a process of quoting, ornamenting and unquoting.

A universe of descriptions has been implemented that can describe inductive families. Dependent types are supported, so types of arguments can depend on other arguments. Universes like these usually allow a lot of types that could not be implemented with a single Agda datatype. Our descriptions are uniquely capable of describing inductive families and dependent types while still keeping to a *subset* of Agda datatypes.

Work on representing dependent types in type theory [18] is combined with work on generic programming with dependent types [4], resulting in descriptions that pass along contexts internally. This idea restricts the use of arguments to exactly those places where they could be used in an Agda datatype. Section 3.5.1 and section 7.2 have shown that these universes with contexts are still implementing inductive types (for those in chapter 4) or inductive families (in chapter 5). Parameters and indices are represented as contexts as well, to effectively allow multiple parameters and multiple indices that can depend on earlier parameters or indices respectively.

Ornaments were adapted to our universe of descriptions as well. We already knew that types where closed under ornamentation (because every ornament results in a description). Our descriptions describe a subset of Agda datatypes and each ornament results in a description. This leads us to an interesting conclusion: With ornaments as we have defined them, *Agda datatypes are closed under ornamentation*. This means that ornamentation of *actual* datatypes can be implemented in a way that will not produce types that can not be represented as a datatype.

The low-level ornaments were shown te be well-suited for abstraction. We were able to implement higher-level ornaments for concepts like 'renaming arguments' and 'adding a parameter'. This shows that it is possible to provide an easier interface to ornamentation that requires only a limited understanding of ornaments.

Datatypes can be quoted to descriptions, and descriptions can be unquoted to datatypes. This allows users to obtain descriptions without having to write them, and to use descriptions without having to work without having to use representations of values such as `⟨ 1 , x , xs , refl ⟩`. Generic programming frameworks like Haskell's generic deriving have already shown the strength of approaches like these.

8.1 Future work

Some limitations of the current implementation were already explained throughout this thesis. We provide some directions for future research based on these limitations. Some of these directions involve fundamental changes to how the descriptions and their ornaments work. Others are more practically oriented, to make the library easier to use.

The *separation of parameters from local contexts*, explained in section 5.4.1, is an obvious candidate for implementation. We already know that the descriptions themselves can be implemented in this way. This would make it possible to implement reornaments on descriptions with parameters (section 6.6.4) and the `addParameterArg` function of section 6.6 could add the parameter argument in other places than at the front of a constructor.

Ornaments on *inductive-recursive* types are not well researched yet. We do not know whether ornaments, quoting and unquoting work for inductive-recursive types. If one wishes to support all types that can be represented by Agda datatypes, this would certainly need to be researched further.

Mutually recursive datatypes usually fall under the inductive families, because inductive arguments can use indices to indicate which of the types is being referred to. Our `ConDesc`/`DatDesc` descriptions do not support mutually recursive types, but there does not seem to be a fundamental reason why this would not be possible. One could add another layer of descriptions to describe bunches of `DatDesc`s and use indexed Σ-descriptions to inform the semantics and ornaments of them. If one description can describe multiple datatypes, it may even be possible to split one datatype into multiple datatypes by refining the index that picks the datatype from the bunch.

Our universe of descriptions does not allow many generic operations. In section 7.1 we show how parameter use can be made explicit, so functions like `flatten` or `sum` could be implemented generically. The same idea of making the meaning of arguments more explicit can be expanded upon, to the point of implementing a full *language describing dependent types* (a la McBride [18]). Quoting and unquoting of arguments to/from such a representation is a whole new issue.

The *unquoting of datatypes* is not entirely automatic yet (section 6.5). We did solve all the problems regarding the unquoting of the *types of* the constructors and of the datatype itself. It would be nice if Agda supported the unquoting of datatypes within the `TC` monad. As an alternative solution, one might implement functionality within the development environment that helps with code generation. One can imagine how the user could instruct the environment to write a datatype definition based on a description, and that the user is then prompted to provide the name for each constructor.

Some simple superficial improvements could be made to the descriptions that are being quoted. They can be modified to allow hidden arguments, such that a constructor like `cons` for `Vec` does not need to have the length index as a visible argument. Contexts could be made to support hidden arguments and names as well, then parameters and indices can contain hidden arguments and the arguments are named properly when the datatype is unquoted. Other more practically oriented modifications could be made to the `HasDesc` record. The structure with `Embeddable` and `Projectable` records as proposed in section 6.7.1 would probably be better suited for generic programming.

Acknowledgements

I would like to thank my supervisor Wouter Swierstra for getting me interested in dependent types, for his helpful comments and for many interesting discussions. His positive attitude and personal mentoring never failed to cheer me up when i had a rough time. His engagement during the whole process is truly appreciated. I also thank Johan Jeuring for taking the time to read and assess this work.

Furthermore i want to thank the developers of Agda for the continued development of a great dependently typed programming language, Andres Löh for making writing easier with lhs2TeX, Pierre-Évariste Dagand for a good discussion about ornaments, the software-technology reading club for lots of interesting sessions, and my university buddies for lots of *fun*.

I am grateful to Maartje for her love and support, i could not have done this without her. She is my motivation and inspiration to do the best i can.

Bibliography

[1] Thorsten Altenkirch, Conor McBride, and Peter Morris. Generic programming with dependent types. In *Proceedings of the 2006 International Conference on Datatype-generic Programming*, SSDGP'06, pages 209–257, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] Thorsten Altenkirch and Peter Morris. Indexed containers. In *Proceedings of the 2009 24th Annual IEEE Symposium on Logic In Computer Science*, LICS '09, pages 277–285, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Richard Bird and Lambert Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer Berlin Heidelberg, 1998.

[4] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 3–14, New York, NY, USA, 2010. ACM.

[5] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 301–312, New York, NY, USA, 1998. ACM.

[6] Pierre-Évariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of Functional Programming*, 24:316–383, 2014.

[7] Nils Anders Danielsson. *Types for Proofs and Programs: International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, chapter A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family, pages 93–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[8] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[9] Dominique Devriese and Frank Piessens. On the bright side of type classes: Instance arguments in agda. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 143–155, New York, NY, USA, 2011. ACM.

[10] Peter Dybjer. Inductive sets and families in martin-löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[11] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

[12] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications*, TLCA '99, pages 129–146, London, UK, UK, 1999. Springer-Verlag.

[13] Peter Dybjer and Anton Setzer. Indexed induction-recursion. In *Proceedings of the International Seminar on Proof Theory in Computer Science*, PTCS '01, pages 93–113, London, UK, UK, 2001. Springer-Verlag.

[14] Patrik Jansson and Johan Jeuring. Polyp—a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 470–482, New York, NY, USA, 1997. ACM.

[15] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for haskell. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 37–48, New York, NY, USA, 2010. ACM.

[16] José Pedro Magalhães and Andres Löh. A formal comparison of approaches to datatype-generic programming. In James Chapman and Paul Blain Levy, editors, Proceedings Fourth Workshop on *Mathematically Structured Functional Programming,* Tallinn, Estonia, 25 March 2012, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 50–67. Open Publishing Association, 2012.

[17] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*. Studies in proof theory. Bibliopolis, Napoli, 1984.

[18] Conor McBride. Outrageous but meaningful coincidences: Dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming*, WGP '10, pages 1–12, New York, NY, USA, 2010. ACM.

[19] Conor McBride. Ornamental algebras, algebraic ornaments. unpublished, dated 23 January 2011, 2011.

[20] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[21] Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, WGP '08, pages 13–24, New York, NY, USA, 2008. ACM.

[22] Ulf Norell and Patrik Jansson. Polytypic programming in haskell. In *Proceedings of the 15th International Conference on Implementation of Functional Languages*, IFL'03, pages 168–184, Berlin, Heidelberg, 2004. Springer-Verlag.