A Generic Translation from Case Trees to Eliminators

Version of June 13, 2024

Kayleigh Zoë Lieverse

A Generic Translation from Case Trees to Eliminators

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Kayleigh Zoë Lieverse
born in Gouda, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

A Generic Translation from Case Trees to Eliminators

Author: Kayleigh Zoë Lieverse
Student id: 4844033

Abstract

Dependently-typed languages allow one to guarantee correctness of a program by providing formal proofs. The type checkers of such languages elaborate the user-friendly high-level surface language to a small and fully explicit core language. A lot of trust is put into this elaboration process, even though it is rarely verified. One such elaboration is elaborating dependent pattern matching to the low-level construction of eliminators. This elaboration is done in two steps. First, the function defined by dependent pattern matching is translated into a case tree, which can then be further translated to eliminators. We present a generic, well-typed implementation of this second step in Agda, without the use of metaprogramming and unsafe transformations, by providing a type-safe, correct-by construction, generic definition of case trees and an evaluation function that, given an interpretation of such a case tree and an interpretation of the telescope of function arguments, evaluates the output term of the function using only eliminators. We only allow case splits on variables from a fixed universe of data type descriptions, for which we use techniques like basic analysis and specialization by unification.

Thesis Committee:

Chair: Dr. J.G.H. Cockx, Faculty EEMCS, TU Delft
Committee Member: Dr. B. Özkan, Faculty EEMCS, TU Delft
Committee Member: L. Escot, Faculty EEMCS, TU Delft

Contents

Chapter 1

Introduction

Type systems are used in programming languages to prevent bugs by ensuring that functions are called with arguments of the correct type. Curry [16], and later Howard and de Bruijn [24], were the first to notice a correlation between programs and proofs, called the proposition-as-types principle, where propositions can be represented as a type, and a proof of the proposition as a term of that type. Martin-Löf extended this notion to create the first type system containing dependent types leading to a family of type theories called Martin-Löf type theory [26]. Dependent types are types whose definition depends on a value, thus enabling the programmer to assign types that restrain the set of possible values elements of that type can take. Languages based on dependent type theory, such as Agda [2], Coq [13], Idris [7], and Lean [17], allow us to provide precise specifications about the input and output of a program, and write proofs that the program matches its specification. The typechecker can then check these proofs automatically, thus guaranteeing that a program works as intended.

Combining programs and proofs requires the language to be expressive enough such that it is usable, but also simple enough to guarantee that it is sound. Therefore, these languages often have a high-level surface language, in which we code, and a core language that focuses on simplicity, which is often very difficult to manipulate by hand. The typechecker elaborates the high-level surface language into the low-level core language [32]. Errors in this elaboration process might lead to proofs or programs being accepted, while its meaning changed to something that was not intended [33]. For example, we might think we have proven a theorem, while in reality, this is not the case at all. A lot of trust is put into this elaboration process, even though it is rarely verified. To guarantee the correctness of these elaborations we need to verify each part of the elaboration process independently.

One example of an elaboration process is the transformation of definitions by dependent pattern matching [14], which provides a more user-friendly, high-level interface, to the low-level construction of eliminators that express the induction principle for a data type [27] [15]. Agda translates a function defined by dependent pattern matching into a case tree [9], which is a tree where each node corresponds to a case split of a variable and each leaf corresponds to a clause of the function. If a translation from a function defined by dependent pattern matching to a case tree is possible, then it is also possible to translate that function into a function that uses only eliminators [23]. Because of this, Agda does not target eliminators in the elaboration process. However, in Coq an implementation of this step is written in the `Equations` package [35], which translates a function defined by dependent pattern matching to a function that uses eliminators on a case-by-case basis. Coq's type checker then checks whether this translated function is type safe, meaning that there are no guarantees on whether this translation is sound.

In this thesis, we implement this translation in Agda in a correct-by-construction way by first giving a generic definition of case trees. The evaluation function takes any such case tree and evaluates to an output of the function type using only eliminators, ensuring that this evaluation is possible for every case tree that can be defined using this generic definition. This implementation should be considered an interpreter rather than a translation, as we immediately evaluate to an output value. We therefore do not rely on Agda's type checker to check anything other than the code we have written for the evaluation function.

To furthermore ensure that the implementation is type-safe, we do not make use of unsafe transformations, which we enforce by using the `--safe` flag in Agda. This ensures that some features in Agda are prohibited (e.g. accepting unfinished proofs or non-terminating programs), which could lead to inconsistencies in your code. These inconsistencies might allow one to (accidentally) prove false, which impacts the soundness of the code as we can conclude anything we want. We also do not allow the use of metaprogramming, which is type checked only on a case-by-case basis. Our approach is guaranteed to succeed for any well-typed case tree. This implementation is a step forward in reducing the trusted base for dependently-typed languages. The source code of this thesis can be found online. [1]

1.1 Overview

The problem is split into three phases. We start by solving the problem for simple data types, which here means data types that are not indexed on other data types. Then, we solve the problem for one-indexed data types (data types that are indexed by simple data types). Lastly, we solve the problem for fully-indexed data types (data types that are indexed by other indexed data types). Each chapter discusses one aspect of the implementation:

- Chapter 2 provides a background on elaborating dependent pattern matching to eliminators. It introduces the concepts of dependent pattern matching, eliminators, unification, and case trees.
- Chapter 3 introduces simple data types and their generic representation in the code. We furthermore show that we can create a generic elimination principle and well-founded recursion.
- Chapter 4 introduces telescopes, which are used to generically represent a series of input arguments of a function and a series of constructor arguments. We furthermore define operations on these telescopes.
- Chapter 5 provides the first contribution of this research: a generic representation of a case tree for simple data types. We show how to define functions by pattern matching using this representation.
- Chapter 6 shows the evaluation function on a case tree given by the generic representation discussed in Chapter 5 and discusses the soundness of this evaluation.
- Chapter 7 shows the representation of four unification rules (solution, deletion, conflict, and injectivity) for simple data types, which is needed to deal with pattern matching on indexed data types.
- Chapter 8 introduces one-indexed data types. We extend the previously defined elimination principle, well-founded recursion, and telescope of constructor arguments to work on these data types and update the case tree and evaluation function.

[1] `https://github.com/klieverse/case-trees-to-eliminators`

- Chapter 9 introduces indexed data types. We extend everything discussed in the previous sections to work for these data types and show that it is possible to solve functions that make use of higher-dimensional unification [10].

In chapter 10 we discuss the strengths and limitations of the current implementation and in chapter 11 we discuss related efforts in translating pattern matching to eliminators.

Chapter 2

Background: Dependent Pattern Matching and Eliminators

This chapter introduces the concepts of dependent pattern matching, eliminators, unification, and case trees. Readers already familiar with these concepts are recommended to only skim through this chapter.

2.1 Pattern Matching

Data types declare types that are defined by a list of constructors that produce the type. In Agda [2] we can declare data types using the `data` keyword. For example, natural numbers are defined as such:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ -> ℕ
```

The first line states that we are introducing a new type called ℕ, which we define in the type `Set`. The type of types in Agda is `Set`, which means that it is a type whose members themselves are again type. After the `where` keyword, every line explains how to construct values of type ℕ. Those are called the constructors of the data type. ℕ has two constructors:

- Constructor `zero` takes no argument and produces a natural number.
- Constructor `suc` takes one natural number as parameter and constructs a new natural number: its successor.

This is known as the Peano encoding: natural numbers are encoded in unary. Every natural number n is encoded as n applications of constructor `suc` to constructor `zero`. Agda allows us to use regular numerical notation to write those natural numbers. For example, 0 maps to `zero`, 1 maps to `suc zero`, and 2 maps to `suc (suc zero)`.

Now that we have introduced a new data type, we can define operations on it. We do so using pattern matching. For example, we can define the addition function for natural numbers as follows:

```
add : ℕ -> ℕ -> ℕ
add zero    m = m
add (suc n) m = suc (add n m)
```

The first line states that we are defining a function named `add`, which takes 2 natural numbers as arguments and produces a natural number as output. The remaining lines are function clauses. On the left-hand side of the equal sign we have patterns. If the input matches a

given pattern, then the function returns the value on the right-hand side of the associated clause.

- The first clause, or base case, states that by adding 0 to a number m, we can return that number m.
- The second clause, or inductive case, states that by adding the successor of a number n to another number m, we can return the successor of adding n and m.

This is an example of a recursive function, where the recursive call happens in the second clause on a smaller input: the first argument decreases.

An example evaluation of the function `add` on the natural numbers 2 and 1 is as follows:

```
add 2 1 = add (suc (suc zero)) (suc zero)
        = suc (add (suc zero) (suc zero))
        = suc (suc (add zero (suc zero)))
        = suc (suc (suc zero))
        = 3
```

In the first line, the expression `add (suc (suc zero)) (suc zero)` matches the second clause of the `add` function, reducing to the second line. The expression `add (suc zero) (suc zero)` in the second line again matches against the second clause, reducing to the expression to `suc (add zero (suc zero))` in the third line. The expression `add zero (suc zero)` in the third line matches the first clause, reducing the expression to `suc zero`. There are no more expressions that contain calls to the function `add`, so it evaluates to 3.

Data types can have parameters, which remain the same in the types of the constructors. They are declared after the name of the data type, but before the colon. For example, we can represent a data type of a list of arguments of type A as such:

```
data List (A : Set) : Set where
  []   : List A
  _::_ : A -> List A -> List A
```

In Agda, we can use underscores to denote where arguments go. For example, in the case of the non-empty list, if we have two elements $(a : A)$ and $(as : \texttt{List}\ A)$, we can call its constructor using $a :: as$.

Data types can also have indices that can differ from constructor to constructor, in constrast to parameters. These are declared after the colon. For example, we can represent a data type of a vector, which is a list of objects with a determined length, as such:

```
data Vec (A : Set) : ℕ -> Set where
  nil  : Vec A 0
  cons : (n : ℕ) -> A -> Vec A n -> Vec A (suc n)
```

The parameter A represents the type of the objects in the vector and the index represents the length of the vector. The first line $\texttt{Vec}\ (A : \texttt{Set}) : \mathbb{N} \to \texttt{Set}$ tells us that for any type $(A : \texttt{Set})$ and any natural number n, we are declaring a new type $\texttt{Vec}\ A\ n$, which belongs to the type of `Set`. Hence, we define a family of inductive types, rather than a single type [19].

- Constructor `nil` takes no arguments and produces the empty vector, where we denote the length with 0.
- Constructor `cons` takes a natural number n, an element of type A, and a vector of length n as parameters, and constructs a new vector where the length is the successor of n.

This is an example of a dependent data type: the set of values that an element of a certain type can take is dependent on the interpretation of the indices of that type. This is important, as it gives extra precision at compile-time about the correctness and safety of certain operations. For example, if we want to take the first element of a vector, we can make sure that we only perform such an operation on a non-empty vector, hence always producing a value:

```
head : {A : Set} {n : ℕ} -> Vec A (suc n) -> A
head (cons n a as) = a
```

Here, A and n are implicit arguments. Implicit arguments are denoted by $\{...\}$ and allow us to call the function without that argument, provided that the type checker can figure the term out by itself. If we perform pattern matching on an element of type `Vec` A (`suc` n), we have the additional constraint that the vector must be non-empty, hence we cannot match on constructor `nil`. Therefore, we only get one clause that states that we can get the head of a vector, that contains a and the remaining vector as, by returning a.

2.2 Eliminators

Dependent pattern matching provides a more user friendly, high-level interface, to the low-level construction of eliminators. In dependent type theory, each data type comes with an elimination principle, or eliminator, which expresses the induction principle for that data type, and enables the definition of functions operating on this data type. For example, if we want to prove a property P on natural number, we can proceed by induction. First we prove that the property holds for `zero`, and then we prove that, if the property holds for some natural number k, it holds for its successor `suc` k. Then we can guarantee that the property holds for any natural number:

$$\forall\ P \to (P\ \texttt{zero}) \to (\forall\ k \to P\ k \to P\ (\texttt{suc}\ k)) \to \forall\ n \to P\ n$$

We can define eliminators by hand in Agda using pattern matching. For example, the eliminator for natural numbers is defined as such:

```
ℕ-elim : (P : ℕ -> Set)
  -> (pZero : P zero)
  -> (pSuc : ∀ k -> P k -> P (suc k))
  -> ∀ n -> P n
ℕ-elim P pZero pSuc zero    = pZero
ℕ-elim P pZero pSuc (suc n) = pSuc n (ℕ-elim P pZero pSuc n)
```

We can create a function that expresses the induction principle for a predicate P on a natural number n by supplying a proof `pZero`, where predicate P holds for `zero`, and a proof `pSuc`, where predicate P holds for `suc` k given that predicate P holds for k.

- If n is zero, we know that P `zero` holds by `pZero`.
- If n is `suc` n, we know that P (`suc` n) holds, by calling `pSuc` with n. We can infer that $P\ n$ holds by calling the eliminator recursively for n.

We can use this eliminator to define functions on natural numbers. For example, we can define the addition function using eliminators by performing induction on the first argument, just like we did with the function defined by pattern matching:

```
add : ℕ -> ℕ -> ℕ
add = ℕ-elim (λ n -> ℕ -> ℕ) (λ m -> m) (λ n h m -> suc (h m))
```

The predicate takes a natural number n that it performs induction on and another natural number m that we want to add to n, and returns a natural number.

- For `pZero` we have that n is `zero`, thus we return m.
- For `pSuc` we have a natural number n, a hypothesis h that takes a natural number m and returns $P\ n\ m$, and a natural number m, thus we return the successor of m applied to the hypothesis.

The eliminator for vectors expresses the induction principle for a predicate P on a vector xs, which holds if we can supply a proof `pNil`, where predicate P holds for `nil`, and we can supply a proof `pCons`, where predicate P holds for `cons` $n\ a\ xs$, given P holds for xs:

```
Vec-elim : (P : {n : ℕ} -> Vec A n -> Set)
  -> (pNil : P nil)
  -> (pCons : ∀ {n} a xs -> P xs -> P (cons n a xs))
  -> ∀ {n} (xs : Vec A n) -> P xs
```

Defining the `head`-function using only eliminators requires different techniques:

```
head :  {A : Set} {n : ℕ} -> Vec A (suc n) -> A
head {n = n} xs = Vec-elim (λ {n'} xs' -> n' ≡ suc n -> A)
  (λ e -> ⊥-elim (conflict e))
  (λ a xs H e -> a) xs refl
```

- First, when we eliminate the vector xs, we require that the index n' of the vector we replace it by is equivalent to the index `suc` n of the eliminated vector. In Agda, we represent this equivalence using the identity type, which contains only one constructor `refl` that holds when two terms are equal. However, if we have two element $(x, y : A)$ that are provably unequal, then $x \equiv y$ is an empty type:

  ```
  data _≡_ {A : Set} (x : A) : A -> Set where
    refl : x ≡ x
  ```

  To ensure that the indices of both vectors are equivalent, we use a technique called basic analysis [28], where we add an additional constraint to the predicate P and fill in `refl` whenever the constraint is satisfied.

- We then use a technique called specialization by unification [23] (denoted by `conflict`) on the equivalence e in the case of `pNil` to derive that the type `zero` $\equiv$ `suc` n is empty. On the empty type we can call $\bot -$ `elim` to conclude anything we want (in this case that an element of type A exists).
- In the case of `pCons` we simply fill in a.

Working with eliminators by hand can thus quickly become unmanageable and unreadable. Therefore, some dependently-typed languages, like Agda, provide the high-level interface of dependent pattern matching.

2.3 From Pattern Matching to Case Trees

When elaborating a function defined by dependent pattern matching, the function is translated into a case tree [4] [9]. A case tree is a tree where each node corresponds to a case split and each leaf corresponds to a clause of the function. For example, the case tree of the addition function would look as follows, where the brackets [] represent the variables used in the patterns and $n\ m$ represent the patterns. We underline the variables that a case split is performed on:

$$[(n : \mathbb{N})(m : \mathbb{N})]\ \underline{n}\ m \begin{cases} [(m : \mathbb{N})]\ \texttt{zero}\ m \mapsto m \\ [(n' : \mathbb{N})(m : \mathbb{N})]\ (\texttt{suc}\ n')\ m \mapsto \texttt{suc}\ (\texttt{add}\ n'\ m) \end{cases}$$

We start with a node containing two natural numbers n and m that represent the input arguments of the addition function. A case split is performed on n, resulting in two new case trees (one for each constructor of n):

- If n maps to `zero`, the first clause of the addition function is reached, and thus we get a leaf node containing the right-hand side of the clause.
- If n maps to $\texttt{suc}\ n'$ for a fresh variable n', the second clause of the addition function is reached, and thus we get a leaf node containing the right-hand side of the clause.

In the case where we split on a variable of an indexed data type, we need to check whether a constructor is allowed to replace the variable based on the indices in the type of that variable and the indices of the type of the constructor. For example, the case tree of the `head`-function would be defined as follows:

$$[(n : \mathbb{N})(\underline{xs} : \texttt{Vec}\ A\ (\texttt{suc}\ n))]\ xs \begin{cases} [(n : \mathbb{N})(a' : A)(xs' : \texttt{Vec}\ A\ n)]\ (\texttt{cons}\ n\ a'\ xs') \mapsto a' \end{cases}$$

That is, a case split is performed on variable xs of type $\texttt{Vec}\ A\ (\texttt{suc}\ n)$, leading to two possible case trees (one for each constructor of xs):

- If xs maps to `nil`, we have to check whether `zero` (the index of `nil`) is equivalent to $\texttt{suc}\ n$ (the index of xs). We know that this is never possible and thus there is no case tree for this mapping.
- If xs maps to $\texttt{cons}\ n'\ a\ xs'$ for fresh variables n', a, and xs', we have to check whether $\texttt{suc}\ n'$ is equivalent to $\texttt{suc}\ n$. If this is possible, we can also replace each occurence of n' by n, thereby removing n' from the variables that are used in the patterns. Now, the clause of the head function is reached, and thus we get a leaf node containing the right-hand side of said clause.

To check whether a constructor can be used when we split on elements of indexed datatypes, we perform a process called unification.

2.4 Unification

When deriving a case tree, Agda performs unification on the data type indices. It constructs a unification problem, which is a list of equations (e.g. $(n : \mathbb{N})(\texttt{suc}\ n \stackrel{?}{=} \texttt{zero})$ or $(n\ m : \mathbb{N})(\texttt{suc}\ n \stackrel{?}{=} \texttt{suc}\ m)$) and applies the following unification transitions to simplify the problem step by step:

- The solution rule solves an equation $x \stackrel{?}{=} y$ if one side is a variable and x does not occur free in y, e.g.:
$$(n\ m : \mathbb{N})(n \stackrel{?}{=} m) \simeq (n : \mathbb{N})$$
- The deletion rule removes an equation $x \stackrel{?}{=} x$, e.g.:
$$(n : \mathbb{N})(n \stackrel{?}{=} n) \simeq (n : \mathbb{N})$$
- The injectivity rule simplifies an equation where both sides use the same constructor, by equating all constructor arguments instead, e.g.:
$$(n\ m : \mathbb{N})(\texttt{suc}\ n \stackrel{?}{=} \texttt{suc}\ m) \simeq (n\ m : \mathbb{N})(n \stackrel{?}{=} m)$$

- The conflict rule shows that it is not possible to have an equation where both sides use different constructors, e.g.:

$$(n : \mathbb{N})(\mathsf{suc}\ n \stackrel{?}{=} \mathsf{zero}) \simeq \bot$$

- The cycle rule shows that it is not possible to have an equation where one side occurs in a constructor argument on the other side, e.g.:

$$(n : \mathbb{N})(n \stackrel{?}{=} \mathsf{suc}\ n) \simeq \bot$$

If the unification process ends in $\bot$ (which denotes the empty type), we can eliminate that branch of the case tree. Hence, we can eliminate the case where xs maps to `nil` in the previous example, as the unification problem $(n : \mathbb{N})(\mathsf{suc}\ n \stackrel{?}{=} \mathsf{zero})$ ends in $\bot$ (following the conflict rule). If the unification process ends in success, we can update the variables in that branch of the case tree. For example, in the case where xs maps to $\mathsf{cons}\ n'\ a'\ xs'$ in the previous example, we have the following unification problem:

$$\begin{aligned}(n\ n' : \mathbb{N})(a' : A)(xs' : \mathsf{Vec}\ A\ n')(\mathsf{suc}\ n' \stackrel{?}{=} \mathsf{suc}\ n) \\ \simeq_{\text{Injectivity}} (n\ n' : \mathbb{N})(a' : A)(xs' : \mathsf{Vec}\ A\ n')(n' \stackrel{?}{=} n) \\ \simeq_{\text{Solution}} (n : \mathbb{N})(a' : A)(xs' : \mathsf{Vec}\ A\ n)\end{aligned}$$

Hence, we get that xs maps to $\mathsf{cons}\ n\ a'\ xs'$ instead of $\mathsf{cons}\ n'\ a'\ xs'$.

2.5 From Case Trees to Eliminators

Agda does not translate a case tree back to a function using only eliminators, because on paper it is proven that functions representable by a case tree can be translated to data type eliminators [23], thus admitting pattern matching in traditional type theory, which is equipped only with the elimination principle.

In this thesis, we put this translation to the test by creating an evaluation function that takes a case tree of a function and the input arguments of that function and evaluates the function for these input arguments using only eliminators. This will give more confidence in the handwritten proof and reduce the trusted code base of Agda (and other proof assistents that translate functions defined by pattern matching to case trees, but not to eliminators).

Chapter 3

Generic Simple Data Types

Because we want to implement a generic translation, that works for every data type, we need a way to manipulate data type definitions using regular programs. This is knows as data type-generic programming, where we use a universe of data type descriptions [5]. In this chapter we focus on giving this representation for simple data types.

Simple data types are data types without indices. Examples of simple data types include natural numbers and (parameterized) lists, which we both have seen in chapter 2. An example of a data type that is **not** simple is the indexed data type of vector, which we have also seen in chapter 2. What makes this data type not simple is that it is indexed.

In this chapter, we define a universe of data type descriptions (section 3.1) to give a generic representation of simple data types in Agda (section 3.2). Then, we show the definition of the elimination principle for an element of such a data type (section 3.3). Lastly, functions defined by pattern matching often contain recursive calls. Eliminators only allow for structural recursion, but we also want to allow the case tree to work for functions that use well-founded recursion. Hence, we add a data type that captures all possible recursive calls for an element of an arbitrary data type (section 3.4).

3.1 Universe of Data Type Descriptions

The generic representation of simple data types should contain all the information about the structure of the data types: how many constructors they have, with how many arguments, and types thereof. On these data types we want to define generic functions that allows us to reason about them (e.g. elimintation principle, unification rules). If we use a fixed universe of data type descriptions, we can derive these generic functions that work for every variable whose type is a data type from this fixed universe [5]. Chapman [8] shows an implementation of this universe for simple data types:

```
data Desc : Set₁ where
  one'   : Desc
  Σ'     : (S : Set) (D : S -> Desc) -> Desc
  ind×' : (D : Desc) -> Desc
```

A description of a data type in this fixed universe of data type descriptions for simple data types consists of three constructs:

- `one'` describes a constructor expecting no further arguments. An example is the empty description for the case of `[]` for lists.

- `Σ' S D` denotes branching constructors on values of type S, where D denotes a mapping from each element in S to a constructor description. This allows for encoding both distinct constructors (e.g. the data type declaration for lists where S is the set containing two elements, one for each constructor) and a single constructor with the type of remaining arguments that depends on the first arguments (e.g. $(a : A)$ in the case of `a::as` for a list).
- `ind × ' D` denotes a constructor expecting a recursive arguments of the data type that we are describing (e.g. $(as$: `List` $A)$ in the case of `a::as` for a list) and the remaining constructor description `D`.

From these three constructs, we can describe any simple data type. For example, we can describe the data type `ListD`, which is parameterized on a type A:

```
ListD : (A : Set) -> Desc
ListD A = Σ' (Fin 2) (λ where
  f0 -> one'
  f1 -> Σ' A (λ a -> ind×' one'))
```

- We use `Σ'` to branch on the constructors of `ListD`. The `Fin` n data type contains n objects; we denote them as `f0`, `f1`, ..., `f(n − 1)`. Using this in `Σ'` allows us to create exactly n branches, hence we can use it to branch on the constructors of the list data type. As the list data type contains two constructors, we take `Fin 2`, which only contains the elements `f0` and `f1`.
- We perform anonymous pattern matching on `Fin 2` to provide the constructor descriptions using the `where` syntax, where `f0` denotes the constructor `[]` and `f1` denotes the constructor `_::_`.
- Constructor `[]` takes no arguments, so we take the empty description `one'`.
- Constructor `_::_` takes an element $(a : A)$, for which we use `Σ'`, and a recursive element, for which we use `ind × '`.

We now have a way to describe the structure of simple data types. We still need an interpretation of these descriptions as actual Agda types. The action $[\![D]\!]X$ can be interpreted as adding "one layer" of the data type, where every inductive reference refers to type X. For example, elements of type $[\![$`ListD` $A]\!]X$ are either `[]` or `a::X`:

```
⟦_⟧ : Desc -> Set -> Set
⟦ one'    ⟧ X = ⊤
⟦ Σ' S D  ⟧ X = Σ[ s ∈ S ] (⟦ D s ⟧ X)
⟦ ind×' D ⟧ X = X × (⟦ D ⟧ X)
```

- In the case where D is the empty constructor `one'` we return $\top$, which is a type that contains only one element `tt`.
- If D is $\Sigma'\ S\ D$, we want an element of the dependent sum type, which means that we have a pair (s, xs) where $(s : S)$ and $(xs : [\![D\ s]\!]X)$
- If D contains a recursive call, we want an element of the product type, which is the type of non-dependent pairs, which means that we have a pair (x, xs) where $(x : X)$ and $(xs : [\![D]\!]X)$.

Now, we define the fixpoint of the action, meaning that we add infinitely many layers of the action, which allows us to define a regular Agda type given a description:

```
data μ (D : Desc) : Set where
  ⟨_⟩ : (d : ⟦ D ⟧ (μ D)) -> μ D
```

For example, we can define an empty list as an element of type μ (`List` A) by calling the constructor of μ with `f0`:

```
[]' : μ (ListD A)
[]' = ⟨ f0 , tt ⟩
```

We can also define a non-empty list as an element of type μ (`List` A), provided we have an element of type A and another list, by calling the constructor of μ with `f1`:

```
_::'_ : A -> μ (ListD A) -> μ (ListD A)
x ::' xs = ⟨ f1 , x , xs , tt ⟩
```

Agda furthermore allows us to define pattern synonyms using the `pattern` syntax, which are declarations that can be used on the left-hand side of a function (when pattern matching), as well as on the right-hand side (in expressions). This allows us to define functions on elements of this universe of data type descriptions using regular pattern matching. For examples, we can declare the following patterns for the constructors of elements of type μ (`List` A):

```
pattern []'      = ⟨ f0 , tt ⟩
pattern x ::' xs = ⟨ f1 , x , xs , tt ⟩
```

We can moreover illustrate that μ (`ListD` A) behaves just like the data type `List` A from chapter 2 by showing that we can translate from an element of type μ (`ListD` A) to an element of type `List` A, using the pattern synonyms `[]'` and x ::$'$ xs:

```
ListD->List : μ (ListD A) -> List A
ListD->List []'        = []
ListD->List (x ::' xs) = x :: (ListD->List xs)
```

We can furthermore translate from an element of type `List` A back to an element of type μ (`ListD` A), and show that these functions are inverses of each other.

3.2 Simple Data Type Representation

The representation of Chapman uses Σ' to denote both the different constructors and constructor arguments. This makes it impossible, given a description, to know exactly how many constructors a data type is made of. But to perform a case split on a variable whose type is a data type from this universe of data type descriptions, we need to be able to derive certain properties. First, we need to know the number of different constructors to decide the number of branches the case tree splits into. Secondly, we need to be able to derive all constructor arguments for a specific constructor to decide which fresh variables are added in a new branch. To allow for this, we seperate the description in a data type description (`DataDesc`) and a constructor description (`ConDesc`) and add indices c_n and a_n that denote the number of constructors and constructor arguments, respectively [3].

The constructor description remains more or less the same as the previous description, except that we have added an extra index $\mathtt{a}_n$ that enforces a constant number of arguments for a given constructor:

```
data ConDesc : ℕ -> Set₁ where
  one'   : ConDesc zero
  Σ'     : (S : Set)(D : S -> ConDesc aₙ) -> ConDesc (suc aₙ)
  ind×' : (D : ConDesc aₙ) -> ConDesc (suc aₙ)
```

The data type description takes a natural number c_n that enforces a constant number of constructors for a given data type, and requests a mapping from each element in `Fin` c_n to a constructor description:

```
DataDesc : ℕ -> Set₁
DataDesc cₙ = Fin cₙ -> Σ ℕ ConDesc
```

We can now describe the data type of natural numbers in this universe as a data type description that contains 2 constructors. We then need a mapping from `Fin 2` to a dependent sum that contains the number of arguments for a constructor and its description. For the `zero` constructor (at `f0`), we have the empty constructor, so we have 0 arguments and `one'` as constructor description. For the `suc` constructor (at `f1`), we have 1 (recursive) arguments, so we take `ind × ' one'` as constructor description:

```
NatD : DataDesc 2
NatD f0 = 0 , one'
NatD f1 = 1 , ind×' one'
```

The fixpoint does not require any changes, but we update the action of a data type description D on a set X to be a dependent sum type from an constructor index c_i of type `Fin` c_n, where c_n is the number of constructors, to the action of the ith constructor description on the set X, which we get by getting the second element (π_2) of c_i applied to D. The action of a set X on a constructor description C is denoted by $[\![C]\!]c\ X$ which is the same function as $[\![D]\!]\ X$ of the previous section:

```
data μ (D : DataDesc cₙ) : Set where
⟨_⟩ : (d : ⟦ D ⟧ (μ D)) -> μ D

⟦_⟧ : DataDesc cₙ -> Set -> Set
⟦_⟧ {cₙ} D X = Σ[ cᵢ ∈ Fin cₙ ] (⟦ π₂ (D cᵢ) ⟧c X)
```

We can again define pattern synonyms for the constructors `zero` and `suc`:

```
pattern zero'  = ⟨ f0 , tt ⟩
pattern suc' n = ⟨ f1 , n , tt ⟩
```

We can now define the addition function for two natural numbers using regular pattern matching on the universe of data type descriptions for natural numbers:

```
_+_ : μ NatD -> μ NatD -> μ NatD
zero'  + m = m
suc' n + m = suc' (n + m)
```

3.3 The Elimination Principle

When we evaluate a case tree, we want to be able to reason about arbitrary elements in the defined universe of data type descriptions. For this, we define the elimination principle that works on a predicate P for any element of type $\mu\ D$ for an arbitrary data type description D. The elimination principle for an arbitrary data type has a fixed structure:

```
elim-μ : (D : DataDesc cₙ)(P : μ D -> Set)
  -> (p : (d : ⟦ D ⟧ (μ D)) -> ? -> P ⟨ d ⟩)
  -> (x : μ D) -> P x
```

- We have a predicate P that goes from an element of a data type D to a `Set`, which contains the statement we want to prove.

- For every constructor we need a proof p that, given all constructor arguments and a proof that the predicate P holds for all inductive arguments (denoted by ?), shows that P holds for the constructor arguments applied to the constructor function.
- Provided we have all these proofs, we can prove that P holds for any element of that data type.

As we have a varying number of constructors, depending on how a data type in this universe is defined, we need a way to encapsulate that P holds for all inductive arguments. For this, we define an auxiliary data type `All D X P d` that states that $(P : X \to$ `Set`$)$ holds for every subobject in D. For this, we use the product type whenever the constructor description contains an inductive argument x to collect a proof $P\ x$:

```
All : (D : DataDesc cₙ)(X : Set)(P : X -> Set)(d : ⟦ D ⟧ X ) -> Set
All D X P (cᵢ , c) = AllC (π₂ (D cᵢ)) X P c

AllC : (D : ConDesc aₙ)(X : Set)(P : X -> Set)(d : ⟦ D ⟧c X) -> Set
AllC one'      X P tt      = ⊤
AllC (Σ' S D)  X P (s , d) = AllC (D s) X P d
AllC (ind×' D) X P (x , d) = (P x) × (AllC D X P d)
```

Filling `All D X P d` in at ?, we can get a proof that $P\ x$ holds for an arbitrary x of type $\mu\ D$ if we apply x to p. We also need a proof that P holds for each inductive argument, which is encapsulated in the `all` function. The `all` function applies the elimination principle recursively whenever an inductive element is found in x to ensure that P holds for each inductive argument.

```
all : (D E : DataDesc cₙ)(P : μ E -> Set)
  (p : (d : ⟦ E ⟧ (μ E)) -> All E (μ E) P d -> P ⟨ d ⟩)
  (d : ⟦ D ⟧ (μ E) ) -> All D (μ E) P d
all D E P p (s , d) = allC (π₂ (D s)) E P p d

allC : (D : ConDesc aₙ)(E : DataDesc cₙ)(P : μ E -> Set)
  (p : (d : ⟦ E ⟧ (μ E)) -> All E (μ E) P d -> P ⟨ d ⟩)
  (d : ⟦ D ⟧c (μ E) ) -> AllC D (μ E) P d
allC one'      E P p tt       = tt
allC (Σ' S D) E P p (s , d) = allC (D s) E P p d
allC (ind×' D) E P p (x , d) = elim-μ E P p x , allC D E P p d
```

We can use this generic elimination principle to define functions on any data type from the universe of data types. For example, we can define the addition function for natural numbers similar to how we have defined it in chapter 2 using only eliminators, except that we perform anonymous pattern matching on p to differentiate between the two constructors:

```
add₁ : μ NatD -> μ NatD -> μ NatD
add₁ = elim-μ NatD (λ n -> μ NatD -> μ NatD) (λ where
  (f0 , tt) tt m -> m
  (f1 , n , tt) (h , tt) m -> suc' (h m))
```

Using the generic eliminator we can also create a generic case-eliminator, which is a weaker version of the generic eliminator where the inductive hypothesis for the recursive arguments (`All D X P d`) is dropped:

```
case-μ : (D : DataDesc cₙ)(P : μ D -> Set)
  -> (p : (d : ⟦ D ⟧ (μ D)) -> P ⟨ d ⟩) -> (x : μ D) -> P x
case-μ D P p d = elim-μ D P (λ d h -> p d) d
```

3.4 Well-Founded Recursion

With a single call to the elimination principle, we can write functions that use structural recursion, where we only use the value of the evaluation of the immediate predecessor. Another form of recursion is well-founded recursion [37], where we may use the value of the evaluation of any of its predecessors. For example, the fibonacci function recursively calls the function for both `suc'` n and n:

```
fib : μ NatD -> μ NatD
fib zero'           = zero'
fib (suc' zero')    = suc' zero'
fib (suc' (suc' n)) = fib (suc' n) + fib n
```

To write this function using eliminators, we need a way to capture all possible recursive calls, which we can do by using a memoization technique [22] that inductively defines a data structure `Below` that collects all possible recursive calls of a predicate P on an element of data type D [29]. That is, for each inductive argument u in a constructor description, we collect a proof that P holds for u and that P holds for everything below u:

```
Below : {D : DataDesc cn} (P : μ D -> Set) -> (d : μ D) -> Set
Below {D} P ⟨ ci , c ⟩ = BelowC (π2 (D ci)) c where
  BelowC : (C : ConDesc an)(c : ⟦ C ⟧c (μ D)) -> Set
  BelowC one' tt = ⊤
  BelowC (Σ' S E) (s , c) = BelowC (E s) c
  BelowC (ind×' C) (u , c) = (P u × Below P u) × (BelowC C c)
```

We can create an element of this data type given that we have a proof p that states that $P\ d$ holds for an arbitrary d if we know that P holds for everything below d. For each inductive argument u of a constructor we can prove that P holds for everything below u by creating `Below P u` recursively and we can prove that P holds for u by calling p with u and the result of the recursive call:

```
below : {D : DataDesc cn}(P : μ D -> Set)
  -> (p : (d : μ D) -> Below P d -> P d)(d : μ D) -> Below P d
below {D} P p ⟨ ci , c ⟩ = belowC (π2 (D ci)) c where
  belowC : (C : ConDesc an)(c : ⟦ C ⟧c (μ D)) -> BelowC C c
  belowC one' tt = tt
  belowC (Σ' S E) (s , c) = belowC (E s) c
  belowC (ind×' C) (u , c) = ((p u (below P p u) , below P p u ) , belowC C c)
```

We can now define the Fibonacci function by calling the generic case-μ eliminators with the `Below` data type that contains recursive calls to `fib'`. Then, we can take h and h' from this data structure to get the recursive calls `fib' (suc n')` and `fib' n'`, respectively:

```
fib' : (n : μ NatD) -> Below (λ n -> μ NatD) n -> μ NatD
fib' = case-μ NatD (λ n -> Below (λ n -> μ NatD) n -> μ NatD) (λ where
  (f0 , tt    ) tt             -> zero'
  (f1 , n , tt) ((h , b) , tt) ->
    case-μ NatD (λ n -> Below (λ n -> μ NatD) n -> μ NatD) (λ where
      (f0 , tt     ) tt              -> suc' zero'
      (f1 , n' , tt) ((h' , b) , tt) -> h + h') n b)
```

The actual fibonacci function is then a call to this helper function. To collect all recursive calls we call the `below` function with the fibonacci helper to collect all recursive calls:

```
fib : (n : μ NatD) -> μ NatD
fib n =  fib' n (below (λ n -> μ NatD) fib' n)
```

Chapter 4

Representing Telescopes

A telescope is a list of typed variable bindings where each type can depend on the previous variables. For example, $(n : \mathbb{N})(p : n \equiv \texttt{zero})$ is a telescope containing a natural number n and a proof p that n is equivalent to `zero`. We can use telescopes to denote the types of the input variables of a function. For example, the telescope of input arguments for the addition function for natural numbers can be defined as a telescope containing two natural numbers $(n : \mathbb{N})(m : \mathbb{N})$. Additionally, we can use telescopes to denote the arguments of a constructor. For example, for the `zero` constructor for natural numbers, which has no constructor arguments, we can use an empty telescope (denoted by the unit type $\top$). In the case of the `suc` constructor for natural numbers we have one argument $(n : \mathbb{N})$. Therefore, we can use a telescope which contains one element of type $\mathbb{N}$ and from that telescope we can create another natural number by calling the `suc` constructor with that element in the telescope.

On telescopes we can define generic functions that allows us to easily replace, remove, and add variables. It allows us to replace a variable in the telescope of input arguments by the telescope of constructor arguments of the constructor that that variable maps to. For example, if we take the addition function for natural numbers we start with a telescope of input arguments containing two natural numbers n and m. Then, if we perform a case split on n, we can replace n with an empty telescope if n maps to `zero`. If n maps to `suc` n', we can replace n by a telescope containing n'.

In this chapter, we give the representation of a telescope in Agda (section 4.1) and define the telescope of constructor arguments for an arbitrary constructor and show that from such a telescope we can create an element of that constructor (section 4.2). We also define an `expand` operator that inserts a telescope at a given position in an input telescope (section 4.3).

4.1 Representing a Telescope

As each type in the telescope can depend on the previous variables, a telescope can be interpreted as an iterated dependent sum type, which means that if we have a set S the remainder of the telescope may be dependent on an element s of S. To eliminate the possibility that we try to perform a case split on an element in an empty telescope we need to be able to distinguish between an empty telescope and a non-empty telescope. For this, we add an index n in the type of the telescope that maintains the number of elements inside that telescope. For the empty telescope `nil` we have zero arguments and length `zero`. In the case of a non-empty telescope `cons` we have a non-zero length `suc` n for arbitrary $n : \mathbb{N}$:

```
data Telescope : ℕ -> Set₁ where
  nil  : Telescope zero
```

```
  cons : (S : Set)(E : (s : S) -> Telescope n) -> Telescope (suc n)
```

The telescope of input arguments for the addition function for natural numbers would thus contain two elements of type natural number:

```
Δ+ : Telescope 2
Δ+ = cons (μ NatD) (λ n -> cons (μ NatD) (λ m -> nil))
```

We often use an alternative syntax $s \in S, E$ to define a telescope, which would be equivalent to `cons` $S\ (\lambda\ s \rightarrow E)$. We can define the interpretation of a telescope by calling the unit type for the empty telescope and a dependent sum type for a non-empty telescope:

```
⟦_⟧telD : (Δ : Telescope n) -> Set
⟦ nil      ⟧telD = ⊤
⟦ cons S E ⟧telD = Σ[ s ∈ S ] ⟦ E s ⟧telD
```

The addition function for natural numbers can thus be defined by going from an interpretation of the telescope $\Delta+$ to a natural number. We have two elements in the telescope n and m and can perform pattern matching on n inside the telescope. In the case where n maps to `suc` n, we can call the function recursively by creating a new interpretation of the telescope $(n, m,$ `tt`$)$:

```
+ : ⟦ Δ+ ⟧telD -> μ NatD
+ (zero'  , m , tt) = m
+ (suc' n , m , tt) = suc' (+ (n , m , tt))
```

To reason about particular variables in a telescope, we often want to know its exact type and location. To do this, we define an auxiliary type `TelAt` that states that for a certain location k in a telescope of length n, we have an element of type $B\ a$ that is dependent on an element a of type A, which is derived from the first part of the telescope:

```
data TelAt (A : Set) (B : A -> Set) : Telescope n -> ℕ -> Set₁ where
  here  : {a : A}{E : (b : B a) -> Telescope n} -> TelAt A B (b ∈ B a , E y) 0
  there : {S : Set}{E : S -> Telescope (suc n)}
    -> ((s : S) -> TelAt A B (E s) k) -> TelAt A B (s ∈ S , E s) (suc k)
```

- Constructor `here` denotes that at this location in the telescope, we can create an element a of type A, such that the type at this position in the telescope is $B\ a$.
- Constructor `there` denotes that we have not yet reached the location of type $B\ a$. Therefore, we take and element s of the type at that location in the telescope (in this case S) and use that s to construct `TelAt` for the remaining telescope $(E\ s)$, which allows us to use this s to create the element a at position k. Then we can state that we can create an element of type $B\ a$, for some a, at position `suc` k.

We often use an alternative syntax $\Delta[k] : \Sigma[A]B$, which is equivalent to `TelAt` $A\ B\ \Delta\ k$. With this data type we can provide a proof that the element at location 1 in the telescope of function arguments for the addition function $(\Delta+)$ is of type μ `NatD`. As the type μ `NatD` does not depend on other elements in the telescope, we can use the unit type in the place of D. The element at location 0 (n) is also of type μ `NatD`, so we first use the `there` constructor and then `here` by filling in `tt` for the unit type:

```
ℕat1 : Δ+ [ 1 ]:Σ[ ⊤ ] (λ _ -> μ NatD)
ℕat1 = there (λ n -> here tt)
```

Similarly, we define an auxiliary type `TelAt'` $A\ B\ C\ \Delta\ k$, where $(C : (a : A) \rightarrow B\ a \rightarrow$ `Set`$)$, which shows that the types at position k and $k + 1$ in telescope Δ are B and C, respectively. The alternative syntax for this data type is $\Delta[k] : \Sigma[A]B : C$. For example, if we have a

telescope that contains a natural number n, an element m of type `Fin` n, and a proof e that m is equivalent to itself, then we can show that at position 1 in the telescope, we have an element of type `Fin` n and then and element of type $m \equiv m$:

```
FinAt1,≡At2 : (n ∈ ℕ , m ∈ Fin n , e ∈ m ≡ m , nil)
  [ 1 ]:Σ[ ℕ ] (λ n -> Fin n) : (λ n m -> m ≡ m)
FinAt1,≡At2 = there (λ n -> here n)
```

If we have a proof p that at index k in telescope Δ we have an element of type $B\ a$ for some a, and we have an interpretation of Δ, we can retrieve an element of type $B\ a$. In particular, we retrieve the element from the interpretation of the telescope at that position, which gives us an element of type $B\ a$, where a is the element in the `TelAt` data type, that is built from the previous elements in the interpretation of the telescope:

```
lookup : {Δ : Telescope n} {A : Set} {B : A -> Set} (p : Δ [ k ]:Σ[ A ] B)
  -> ⟦ Δ ⟧telD -> Σ A B
lookup (here  a) (b , _ ) = a , b
lookup (there p) (s , xs) = lookup (p s) xs
```

The alternative syntax for `lookup` $p\ xs$ is $xs\ \Sigma[p]$.

4.2 Telescope of Constructor Arguments

To replace a variable in the telescope by the arguments of a specific constructor, we first need to know what the telescope of constructor arguments looks like. For this, we create a function that creates a telescope of the length equal to the number of constructor arguments for a specific constructor C. If the constructor is empty we have an empty telescope. In the case where the types of the other constructor arguments may depend on the current argument of type S, we add S to the telescope and build the remaining telescope using an element $s : S$. In the case where we have an inductive argument, we add an element of type X, which is a set that is passed as an additional argument:

```
conTel : (X : Set)(C : ConDesc a_n) -> Telescope a_n
conTel X (one'   ) = nil
conTel X (Σ' S C ) = s ∈ S , conTel X (C s)
conTel X (ind×' C) = x ∈ X , conTel X C
```

In the previous chapter we have said that $[\![\_]\!]c$ is equivalent to the action defined on a data type description from the works of Chapman [8]. We thus have the following function:

```
⟦_⟧c : ConDesc a_n -> Set -> Set
⟦ one'    ⟧c X = ⊤
⟦ Σ' S D  ⟧c X = Σ[ s ∈ S ] (⟦ D s ⟧c X)
⟦ ind×' D ⟧c X = X × (⟦ D ⟧c X)
```

If we have a telescope of constructor arguments for a constructor description C that works on set X, we can create an element of that constructor description, which we show in the function `telToCon`. If the constructor description is `one'`, we have an empty telescope and also an empty constructor description. If the constructor description is $\Sigma'\ S\ C$ we have that the interpretation of the constructor arguments telescope contains an element $s : S$ and the remaining telescope d is dependent on this s. To create an element of this constructor, we need the element $s : S$ and build the remainder of the description using d. If the constructor description is `ind` $\times'\ C$, the telescope contains an element of type X, which is exactly what we need to build the constructor:

```
telToCon : {X : Set}{C : ConDesc a_n} -> ⟦ conTel X C ⟧telD -> ⟦ C ⟧c X
telToCon {C = one'   } _       = tt
```

```
telToCon {C = Σ' S C } (s , d) = s , telToCon d
telToCon {C = ind×' C} (x , d) = x , telToCon d
```

The reverse also holds. That is, if we have an element of the constructor description C that works on set X, we can create an element of the constructor arguments telescope. For the empty constructor description we have the empty telescope. When the constructor description is $\Sigma'\ S\ C$, we have an $s : S$ in the element of the constructor description, which is what the telescope requires. When the constructor description is `ind` $\times'\ C$, we have an element of type X, which is exactly what the telescope requires:

```
conToTel : {X : Set}{C : ConDesc a_n} -> ⟦ C ⟧c X -> ⟦ conTel X C ⟧telD
conToTel {C = one'   } _       = tt
conToTel {C = Σ' S C } (s , d) = s , conToTel d
conToTel {C = ind×' C} (x , d) = x , conToTel d
```

We can furthermore show that for any element of a constructor description C that works on set X, translating to the telescope of constructor arguments and translating that to the constructor is equivalent to the original element. In the case of the empty constructor we have to prove that `tt` $\equiv$ `tt` holds, which is trivially done with `refl`. When the constructor description is $\Sigma'\ S\ C$, we have to prove that $(s, \texttt{telToCon}\ (\texttt{conToTel}\ d)) \equiv (s, d)$, for which we can use `cong` that states that if we apply a function (in this case $\lambda d \rightarrow (s, d)$) to equivalent arguments, we get an equivalent result. We can prove that `telToCon` $(\texttt{conToTel}\ d) \equiv d$ by calling the function recursively. When the constructor description is `ind` $\times'\ C$ we can do the same trick except with $x : X$ instead of $s : S$:

```
telToCon∘conToTel : {X : Set}{C : ConDesc a_n} (args : ⟦ C ⟧c X)
  -> telToCon (conToTel args) ≡ args
telToCon∘conToTel {C = one'   } tt      = refl
telToCon∘conToTel {C = Σ' S C } (s , d) = cong (s ,_) (telToCon∘conToTel d)
telToCon∘conToTel {C = ind×' C} (x , d) = cong (x ,_) (telToCon∘conToTel d)
```

4.3 Expanding a Telescope

When we perform a case split on a variable of a certain data type in the telescope of input arguments, we want to replace that variable with the arguments of the constructor that the variable splits into. We can define a function `expandTel` that, given a function f that creates an element of the data type at position k from an interpretation of telescope Y telescope, creates a telescope that removes the element at position k and adds the m variables of the telescope of function f:

```
expandTel : {A : Set} {B : A -> Set} (X : Telescope n) (Y : A -> Telescope m)
    (p : X [ k ]:Σ[ A ] B)(f : {a : A} -> ⟦ Y a ⟧telD -> B a)
  -> Telescope (k + m + (n - suc k))
```

To create an interpretation of this telescope, we need an interpretation of the first part xs, which stays the same until we reach index k. Then, we want to merge an interpretation ys of telescope $Y\ x$, where x is the value that creates the type of variable $(y : B\ x)$ at location k in xs. To merge ys, we need to ensure that the value we get from ys applied to the function f is equivalent to the value y at location k in xs, such that the remainder of xs can be merged at the end, which is encapsulated in the proof eq. The type of the function `expand` thus becomes:

```
expand : {A : Set} {B : A -> Set} {X  : Telescope n} {Y : A -> Telescope m}
  (p  : X [ k ]:Σ[ A ] B)(f : {a : A} -> ⟦ Y a ⟧telD -> B a)
  (xs : ⟦ X ⟧telD)(let (a , b) = lookup p xs)(ys : ⟦ Y a ⟧telD)
  (eq : f ys ≡ b) -> ⟦ expandTel X Y p f ⟧telD
```

We can also create a function `shrink`, that returns only the interpretation of telescope X from an expanded telescope. Again, we keep the first part of the telescope the same, as it contains variables from the interpretation of X. If we reach position k, we call function f with the first part of the remaining telescope, to get an element of type $B\ a$, and keep the second part of the remaining telescope the same. The type of the function is as follows:

```
shrink : {A : Set} {B : A -> Set} {X : Telescope n} {Y : A -> Telescope m}
  (p : X [ k ]:Σ[ A ] B) {f : {a : A} -> ⟦ Y a ⟧telD -> B a}
  -> ⟦ expandTel X Y p f ⟧telD -> ⟦ X ⟧telD
```

Lastly, we can prove that `shrink` and `expand` is a section-retraction pair, meaning that shrinking an expanded telescope results in the original interpretation xs of telescope X:

```
shrink∘expand : {A : Set} {B : A -> Set} {X : Telescope n} {Y : A -> Telescope m}
  (p  : X [ k ]:Σ[ A ] B){f : {a : A} -> ⟦ Y a ⟧telD -> B a}
  (xs : ⟦ X ⟧telD)(let (a , b) = lookup p xs)(ys : ⟦ Y a ⟧telD)
  (eq : f ys ≡ b) -> shrink p (expand p f xs ys eq) ≡ xs
```

With the expand function, we can replace elements of data types with their constructor arguments. For example, if we have a telescope containing a natural number n and an equivalence relation $n \equiv$ `zero'`, we can define their expansion by pattern matching on the natural number n. For constructor `zero'`, we remove n and replace n in the equivalence relation by `zero'`. For constructor `suc'`, we replace n by a fresh variable and replace n in the equivalence relation by `suc'` n:

```
expandNatTel : ⟦ n ∈ μ NatD , e ∈ n ≡ zero' , nil ⟧telD -> Σ ℕ Telescope
expandNatTel (zero'  , _) = _ , e ∈ zero' ≡ zero' , nil
expandNatTel (suc' n , _) = _ , n ∈ μ NatD , e ∈ suc' n ≡ zero' , nil
```

We can show that these are their expansionsby using the `expand` function. If we have an interpretation of a telescope containing a natural number n and an equivalence relation $n \equiv$ `zero'`, we can use the function `expand` on the first element in the telescope xs (denoted by `here tt`), where Y is the telescope containing the arguments of the respective constructor (`f0` in the case of constructor `zero'` and `f1` in the case of constructor `suc'`). From an interpretation of telescope Y, we can create an element of data type `NatD` by calling the function `telToCon`:

```
expandNat : (xs : ⟦ n ∈ μ NatD , e ∈ n ≡ zero' , nil ⟧telD)
  -> ⟦ π2 (expandNatTel xs) ⟧telD
expandNat (zero'  , e , tt) = expand {Y = λ _ -> conTel (μ NatD) (π2 (NatD f0))}
  (here tt) (λ args -> ⟨ f0 , telToCon args ⟩) (zero' , e , tt) tt refl
expandNat (suc' n , e , tt) = expand {Y = λ _ -> conTel (μ NatD) (π2 (NatD f1))}
  (here tt) (λ args -> ⟨ f1 , telToCon args ⟩) (suc' n , e , tt) (n , tt) refl
```

Chapter 5

Case Trees for Simple Data Types

To create an evaluation function that evaluates a case tree without the use of pattern matching on simple data types, we need a generic representation of a case tree. A case tree for a function `f` is a tree where each node corresponds to a case split and each leaf corresponds to clause of `f`. In chapter 2 we showed an example of a translation from a function defined by pattern matching to a case tree for the addition function for natural numbers. This chapter provides the first contribution of this research: a generic representation of a case tree, where we only allow the user to perform a case split on simple data types from the universe of data type descriptions defined in chapter 3.

We first show the generic representation of the case tree in Agda (section 5.1). The remainder of the chapter is dedicated to example elaborations from functions defined by pattern matching to this generic representation. Section 5.2 shows the case tree for the `half`-function for natural numbers that divides a natural number by 2. Section 5.3 show the case tree for the `create`-function that creates a vector of length n for an input natural number n.

5.1 Generic Case Trees

A case tree for a function `f` is a tree where each node corresponds to a case split and each leaf corresponds to clause of `f` [4]. When performing a case split on a variable, the tree splits in the number of branches equivalent to the number of constructors that the variable can split into. That variable in the telescope of input arguments is then replaced in these branches by the constructor arguments that belong to the constructor of said branch.

To represent this generically, we first need a generic representation for a function f. For this, we use a telescope Δ of length n which represents the input arguments of the function. The return type of the function may depend on the value of the input arguments (e.g. in the `create`-function the return type is `Vec A n`, where `n` is one of the input arguments), so we have a function T that goes from an interpretation of the telescope of constructor arguments to a certain type. Together, we can thus create a case tree for a function that has as telescope of input arguments Δ and return type function T:

```
data CaseTree (Δ : Telescope n)(T : ⟦ Δ ⟧telD -> Set) : Set₁ where
```

In the leaf of a case tree, we do not split on any more variables, and we can fill in the clause of the function following the splits taken to get to that leaf. Given the telescope at the leaf Δ and a return type T based on an interpretation of this telescope, we can fill in a function t, which contains a clause, that goes from the interpretation $args$ of the telescope Δ to an element of the return type dependent on this:

```
  leaf : (t : (args : ⟦ Δ ⟧telD) -> T args) -> CaseTree Δ T
```

In the node of a case tree, we perform a case split on a variable in the telescope of input arguments. To perform this split, we need to know the different constructors of that data type and all their constructor arguments. To determine this we add a constraint that the type of the variable may only be from the universe of data type descriptions specified in chapter 3. We use the data type `TelAt` from chapter 4 to determine that the variable at a location k is of type $\mu\ D$ in telescope Δ, which we store in variable p, and as $\mu\ D$ does not depend on any variables in the telescope (as it is a simple data type which is not indexed) we can fill in $\top$ as presumption. Then, for each possible constructor from the data type (there are c_n possible constructors, which we denote by the variable c_i), we have to build a new case tree where the variable of type $\mu\ D$ is replaced by all constructor arguments in telescope Δ:

```
  node : {D : DataDesc cn} (p : Δ [ k ]:Σ[ ⊤ ] (λ tt -> μ D))
    -> (bs : (ci : Fin cn) -> CaseTree ? ?) -> CaseTree Δ T
```

To extend telescope Δ we can call `expandTel` from chapter 4 on the telescope where we replace $\mu\ D$ with the telescope of constructor arguments `conTel` following constructor c_i from data type D. From this telescope of constructor arguments, we can create an element of $\mu\ D$ by creating an element of the fixpoint using c_i and the function `telToCon` on the telescope of constructor arguments. As the telescope of input arguments Δ is updated after the case split, we can not call the function T directly, as it works on the original telescope Δ. Therefore, we first have to use the function `shrink` to return to the original Δ. The final description of the case tree is then as follows:

```
data CaseTree (Δ : Telescope n)(T : ⟦ Δ ⟧telD -> Set) : Set1 where
  leaf : (t : (args : ⟦ Δ ⟧telD) -> T args) -> CaseTree Δ T
  node : {D : DataDesc cn} (p : Δ [ k ]:Σ[ ⊤ ] (λ tt -> μ D))
    -> (bs : (ci : Fin cn) -> CaseTree (expandTel Δ
        (λ tt -> conTel (μ D) (π2 (D ci))) p (λ args -> ⟨ ci , telToCon args ⟩))
      (λ args -> T (shrink p args))) -> CaseTree Δ T
```

We will now look at some example functions.

5.2 `half`-Function for Natural Numbers

The `half`-function for natural numbers that takes a natural number n and returns that number divided by 2. In the case where n is 0 we return `zero`. In the case where n is 1 we cannot divide any more by 2, so we also return 0. For any other number (`suc (suc n)`) we return the successor of the `half` of `n`. Using the natural number data type description from chapter 3, we can thus define the function using pattern matching as follows:

```
half : μ NatD -> μ NatD
half zero' = zero'
half (suc' zero') = zero'
half (suc' (suc' n)) = suc' (half n)
```

If we write this out in a case tree, we split on n to get the cases `zero` and `suc n'`. For `zero` we can fill in the right-hand side of the equation and for `suc m` we again perform a case split on m to get the cases `suc zero` and `suc (suc k)`, for which we both fill in the right-hand side. We underline the variables that we perform a case split on:

$$[(n : \mathbb{N})]\ \underline{n} \begin{cases} [] \ \texttt{zero} \mapsto \texttt{zero} \\ [(m : \mathbb{N})]\ (\texttt{suc}\ \underline{m}) \mapsto \begin{cases} [] \ (\texttt{suc zero}) \mapsto \texttt{zero} \\ [(k : \mathbb{N})]\ (\texttt{suc}\ (\texttt{suc}\ k)) \mapsto \texttt{suc}\ (\texttt{half}\ k) \end{cases} \end{cases}$$

Now, if we want to translate this using the generic representation of the case tree, we first need to specify the telescope of input arguments and the return type. In the telescope of

input arguments halfΔ we need the element $n : \mu$ NatD. However, the case tree contains a recursive call in the third leaf, which we do not want in the final case tree, as then we would have no way to evaluate the case tree whilst ensuring that we do not use pattern matching. Hence, we add an element of the Below type from section 3.4 to the telescope that stores this recursive call:

```
halfΔ : Telescope 2
halfΔ = n ∈ μ NatD , b ∈ Below (λ n -> μ NatD) n , nil
```

As the return type does not depend on this telescope, we can state μ NatD as return type:

```
halfT : ⟦ halfΔ ⟧telD -> Set
halfT xs = μ NatD
```

Then, we can create the case tree using the generic representation as follows. We first perform a case split on the first variable in the telescope n (denoted by here tt). Then, we need a function from all constructors of n (in this case zero' and suc') to a new case tree where n is replaced by said constructor arguments. We have f0, which denotes the constructor zero', where n is replaced by the empty telescope as zero' does not have any constructor arguments and b is replaced by tt as there are no inductive arguments in zero'. This results in a leaf node where the function from an interpretation of the updated telescope to μ NatD is simply a lambda-function that maps each entry to zero'.

For the constructor suc' m, denoted by f1, we have that n is replaced by m and b is updated to contain Hm which contains all possible recursive calls for m and bm which contains all possible recursive calls below m, as suc' contains one inductive argument. We perform another case split on m, which is again the first variable in the telescope (denoted by here tt), resulting again in two new case trees (one for each constructor). We have that suc' zero' (denoted by f0) is a leaf node (where bm replaced by tt as zero' does not contain any inductive calls), where the function from an interpretation of the updated telescope to μ NatD is simply a lambda-function that maps each entry to zero'. In the case of suc' (suc' k) (denoted by f1) we have that m is replaced by k and bm is replaced to contain Hk and bk. We again have a leaf node and take the successor of the recursive call Hk from the interpretation of the updated telescope. This results in the following case tree representation:

```
CTHalfRoot : CaseTree halfΔ halfT
CTHalfRoot = node (here tt) (λ where
  f0 -> leaf (λ where (tt , tt) -> zero')
  f1 -> node (here tt) (λ where
    f0 -> leaf (λ where (((Hm , tt) , tt) , tt) -> zero')
    f1 -> leaf (λ where (k , (((Hm , ((Hk , bk) , tt)) , tt) , tt)) -> suc' Hk)))
```

5.3 create-Function for Vectors

We can describe the Vec data type by indexing on a natural number μ NatD:

```
data Vec (X : Set) : (n : μ NatD) -> Set where
  nilV  : Vec X zero'
  consV : (n : μ NatD) -> X -> Vec X n -> Vec X (suc' n)
```

The create-function for vectors takes a natural number $n : \mu$ NatD and an element $x : X$ and return a vector of length n containing only the element x. If n is zero' we can thus return the empty vector nilV and if n is suc' m we can return a non-empty vector consV containing element x and perform a recursive call to create a vector of length m. We can thus define the function using pattern matching as follows:

```
create : {X : Set} (n : μ NatD) -> (x : X) -> Vec X n
create zero'    x = nilV
create (suc' m) x = consV m x (create m x)
```

If we want to represent this function as a case tree using our generic representation, we first need to specify the telescope of input arguments and the return type. In the telescope of input arguments `create`Δ we have the natural number n and an element $x : X$. Note that we again have a recursive call in the pattern matching function, so we add an element of the below type that given a natural number n creates an element of type `Vec X n`:

```
createΔ : {X : Set} -> Telescope 3
createΔ {X} = n ∈ μ NatD , x ∈ X , b ∈ Below (λ n -> Vec X n) n , nil
```

The return type `Vec X n` is dependent on the input element n, so the `createT` function uses element n from the interpretation of the `create`Δ telescope:

```
createT : {X : Set} -> ⟦ createΔ {X} ⟧telD -> Set
createT {X} (n , _) = Vec X n
```

In the case tree, we split on the natural number n, which is the first element in the telescope (denoted by `here tt`). We then need a function from all constructors of n (`zero'` and `suc' m`) to the updated case trees. In the case of `zero'` (denoted by `f0`) we have that n is replaced by the empty telescope and in the return type n is also replaced by `zero'`, so we need a vector of length `zero'`. We reach a leaf node so we fill in the right-hand side `nilV`. In the case of `suc' m` (denoted by `f1`) we have a telescope where n is replaced by `suc' m` (and thus n in the return type as well) and b is updated to contain Hm, which contains all possible recursive calls for m and bm which contains all possible recursive calls below m as `suc'` contains an inductive call m. We again reach a leaf node, so we fill in the right-hand side, but instead of the recursive call we use Hm:

```
CTCreateRoot : {X : Set} -> CaseTree (createΔ {X}) (createT {X})
CTCreateRoot {X} = node (here tt) (λ where
  f0 -> leaf (λ _ -> nilV)
  f1 -> leaf (λ where (m , (a , (((Hm , bm) , _) , _))) -> consV m a Hm))
```

Chapter 6

From Case Trees to Eliminators

McBride [23] originally showed a translation on paper from a case tree to a function defined by pattern matching. In this chapter we create a generic evaluation function for the generic represention of the case tree presented in chapter 5 based on the proof by McBride. This evaluation function is not allowed to use pattern matching, except on meta-level. For example, we do allow pattern matching on the structure of the case tree and on the structure of a telescope, but not on the structure of an element of a data type D. Section 6.1 introduces this evaluation function and section 6.2 discusses the soundness of this evaluation.

6.1 Evaluation Function

Let `f` : $(args : [\![\Delta]\!]\texttt{telD}) \rightarrow T\ args$ be a function given by a case tree with telescope Δ that contains the input arguments of the function and a return type $T : [\![\Delta]\!]\texttt{telD} \rightarrow \texttt{Set}$. Then we can evaluate this function using its case tree and an interpretation of the telescope of input arguments $args$, where we return an element of type $T\ args$. To evaluate the case tree we perform pattern matching on the case tree. In the case of a leaf we have a function $f : (args : [\![\Delta]\!]\texttt{telD}) \rightarrow T\ args$, so we can simply apply $args$ to this function f to get an element of the return type $T\ args$:

```
eval : {Δ : Telescope n}{T : ⟦ Δ ⟧telD -> Set}
  -> (ct : CaseTree Δ T) -> (args : ⟦ Δ ⟧telD) -> T args
eval (leaf f) args = f args
```

In the case of a node, we perform a case split on an element of type $\mu\ D$ given a proof p, which states that there is an element of type $\mu\ D$ at position k in telescope Δ. We use the case-μ eliminator to split the element at position k in the interpretation $args$ (denoted by `ret`) of telescope Δ (retrieved using the `lookup` operator from chapter 4). Now, we cannot simply put $\lambda x \rightarrow T\ args$ as predicate in this eliminator, as we lose information about the element at location k in $args$. Hence, we use a technique called basic analysis [28] where we add the constraint that the element at position k in telescope $args$ is equivalent to x and fill in `refl` to satisfy the constraint:

```
eval {T = T} (node {D = D} p bs) args
  = case-μ D (λ x -> ret ≡ x -> T args) cs ret refl where

  ret : μ D
  ret = π2 (args Σ[ p ])
```

Then, we need a function cs that, given an element x of the data type and the knowledge that this x is equivalent to the element at position k in $args$, we can get an element of type $T\ args$:

```
cs : (x : ⟦ D ⟧ (μ D)) -> ret ≡ ⟨ x ⟩ -> T args
```

From x we can derive an element c_i of type `Fin` c_n that denotes the target constructor of x and an element c of type $[\![\pi_2\ (D\ c_i)]\!]c\ (\mu\ D)$ that denotes an interpretation of that constructor. We can derive two properties. The first property q states that an element of type $\mu\ D$ created with `telToCon` (`conToTel`c) is equivalent to the element at position k in the telescope `args`, which we prove using the function `telToCon` ∘ `conToTel` from chapter 4 and the equivalence relation e:

```
q : ⟨ ci , telToCon (conToTel c) ⟩ ≡ ret
q = trans (cong (λ c -> ⟨ ci , c ⟩) (telToCon∘conToTel c)) (sym e)
```

The second property r states that we can retrieve an element of type $T\ args'$, where $args'$ is the expanded and shrinked version of $args$ by p, by recursively evaluating the c_ith branch with the expanded telescope. For this, we use property q to prove that the element at position k in $args$ is equivalent to `conToTel` c applied to the function $(\lambda xs \rightarrow \langle c_i, \texttt{telToCon}\ xs\rangle)$:

```
r : T (shrink p (expand p (λ xs -> ⟨ ci , telToCon xs ⟩) args (conToTel c) q))
r = eval (bs ci) (expand p _ args (conToTel c) q)
```

But we need an element of type $T\ args$. In chapter 4 we showed that shrinking an expanded telescope is equivalent to its original telescope in the function `shrink` ∘ `expand`. If we thus substitute $args$ in T with this expanded telescope, we can use r directly to retrieve an element of type $T\ args$: Then, we can simply call this function with r to get an element of type $T\ t$:

```
cs : (x : ⟦ D ⟧ (μ D)) -> ret ≡ ⟨ x ⟩ -> T args
cs (ci , c) e = subst T (shrink∘expand p args _ q) r
```

Thereby completing the evaluation function.

6.2 Soundness of Evaluation

As the case tree and evaluation function are written in Agda, we have proven that for any function that is defined using the generic representation of the case tree from chapter 5 and any interpretation of the respective telescope of input arguments, we can evaluate to an element of the return type of the function, without the use of pattern matching.

Furthermore, on paper the computational behaviour of the evaluation function is proven to be sound [23]. In other words, if we have a function defined by pattern matching f and its respective case tree ct, where we have that f applied $args$ results in a variable $(x : T\ args)$ (i.e. $f\ args = x$), then we also have that `eval` $ct\ args = x$. We can check this for the example case trees from chapter 5 by proving that the function `eval` and the function defined by pattern matching, applied with the same input, have equivalent results.

An example for the `half`-function is as follows. We can create the proper telescope by calling the `below`-function from chapter 3 with a proof that everything below n holds by a call to the evaluation function with the same case tree:

```
half-tel : (n : μ NatD) -> ⟦ halfΔ ⟧telD
half-tel n = n , below (λ n -> μ NatD)
  (λ n b -> eval CTHalfRoot (n , b , tt)) n , tt
```

Then, we can show that for any natural number n, the evaluation of the case tree of the half function $CTHalfRoot$, with the previously defined telescope, is equivalent to the function `half` applied with n. We can do this by pattern matching on n and filling in `refl` for the leaf nodes and calling the function recursively for branches:

```
≡-half : (n : μ NatD) -> eval CTHalfRoot (half-tel n) ≡ half n
≡-half zero'           = refl
```

```
≡-half (suc' zero')     = refl
≡-half (suc' (suc' n)) = cong suc' (≡-half n)
```

Similarly, we can prove it for the `create`-function:

```
create-tel : {X : Set} -> (n : μ NatD) -> (x : X) -> ⟦ createΔ {X} ⟧telD
create-tel {X} n x = n , x , below (Vec X)
  (λ n b -> eval CTCreateRoot (n , x , b , tt)) n , tt

≡-create : {X : Set} (n : μ NatD) (x : X)
  -> eval CTCreateRoot (create-tel n x) ≡ create n x
≡-create zero'     x = refl
≡-create (suc' n) x = cong (consV n x) (≡-create n x)
```

To prove in Agda that the translation preserves the semantics of the original clause for every function, we would require a generic definition of pattern matching and a translation from that definition to the definition of case trees defined in chapter 5, which is not in the scope of this thesis.

Chapter 7

Unification

Something interesting is happening in Agda's type checker when we use pattern matching on indexed data types. As an example, let us look at the `head`-function defined by pattern matching in chapter 2 using the vector data type from chapter 5, which is indexed on the simple data type of natural numbers:

```
head' : {X : Set} (n : NI.μ NatD) (xs : Vec X (suc' n)) -> X
head' n (consV n x xs) = x
```

Here, we only need to define the case for the `consV` constructor as xs will always be of non-zero length and thus Agda can automatically infer that the case of `nilV` is not possible. Furthermore, we can derive that the input n is the same n in the `consV` constructor. This is true because Agda performs unification on the data type indices by constructing a unification problem and applying one of five unification rules (see chapter 2 for more details) to simplify the problem step by step. If a unification problem ends in $\bot$ we can eliminate that case and if a unification problem succeeds we update the variables.

This has consequences when defining the generic case tree for one-indexed data types, which are data types indexed on simple data types, as we need a way to show that some branches are not possible (e.g. the branch where xs maps to `nilV` should be eliminated). Furthermore, we need to update the variables in the branches that are possible (e.g. the telescope in the branch where xs maps to `consV` should contain only one natural number n instead of two natural numbers). Therefore, we define a unification algorithm that allows the user to specify a unification trace in each branch of the case tree.

In this chapter, we first show the definition of an equivalence in type theory (section 7.1) and then show the implementation of the solution rule (section 7.2), the deletion rule (section 7.3), the conflict rule (section 7.4), and the injectivity rule (section 7.5) for simple data types. Furthermore, we define the unification algorithm (section 7.6) and show how to define and solve the unification problem following this implementation for the `head`-function. We leave out the cycle rule as it is not a rule that is commonly used.

7.1 Equivalences

The identity type $x \equiv_A y$ expresses the property that x and y are equal elements of type A [26]. If x and y are definitionally equal (i.e. they both compute to the same term, denoted by $x = y$), we have the term `refl`: $x \equiv_A y$. However, if x and y are provably unequal, then $x \equiv_A y$ is an empty type. If we have a term of type $x \equiv_A y$, then x and y are propositionally equal. The elimination rule for identity types $x \equiv_A y$ is called the `J` rule:

$$\texttt{J} : (P : (y : A) \to x \equiv_A y \to \texttt{Set})(p : P\ x\ \texttt{refl})(y : A)(e : x \equiv_A y) \to P\ y\ e$$

That is, if we have a proof p of proposition P involving x, and we also know that $(e : x \equiv_A y)$, we can infer a proposition that replaces the occurrences of x in P by $(y : A)$ (i.e. $P\ y\ e$). The elimination rule for identity types $x \equiv_A x$ is called the `K` rule, which is equivalent to the uniqueness of identity proofs:

$$\texttt{K} : (P : x \equiv_A x \to \texttt{Set})(p : P\ \texttt{refl})(e : x \equiv_A x) \to P\ e$$

It states that any two proofs of $x \equiv_A y$ are equal. Cockx [12] showed that each unification rule (defined in section 2) can be represented as equivalences complete with a correctness proof, which is what we base the implementation on.

7.2 Solution Rule

The solution rule allows us to solve an equation $(e : t \equiv x)$ if t is a variable and t does not occur free in x. It thus maps x and e to an empty telescope:

```
solution : Σ[ x ∈ A ] (t ≡ x) -> ⊤
solution xe = tt
```

The function `solution'` maps an empty telescope back to a telescope containing t and `refl`:

```
solution' : ⊤ -> Σ[ x ∈ A ] (t ≡ x)
solution' {t = t} _ = t , refl
```

We can also prove that applying the result of `solution'` (`solution` xe) is equivalent to xe by directly applying the J-rule on the equivalence relation in xe:

```
solution'∘solution : Σ[ x ∈ A ] (t ≡ x) -> solution' (solution xe) ≡ xe
solution'∘solution xe
  = J (λ x e -> solution' (solution (x , e)) ≡ (x , e)) refl (π₂ xe)
```

We could prove that applying the result of `solution` (`solution'` xe) is equivalent to xe, thereby showing the `solution'` is the inverse of `solution`, but we do not need this for the evaluation.

The result of applying the solution rule to a telescope containing x and e is that we remove x and e from the telescope and replace every occurence of x by t and e by `refl`. To update a telescope Δ we define a function `doSolutionTel`, which takes a proof that at index k we can create an element a of type A, which the type B depends on, and an element of type $B\ a$, which is the element t we want to replace x by, such that the element at position k is an element $(x : B\ a)$, and the element at index $k + 1$ is an equivalence between this element x and the element t. At position k we then remove the two elements x and e and replace each occurrence of x by t and each occurence of e by `refl`:

```
doSolutionTel : {Δ : Telescope n} {B : A -> Set}
  -> (p : Δ [ k ]:Σ[ Σ[ a ∈ A ] (B a) ]
      (λ at -> B (π₁ at)) : (λ at x -> (π₂ at) ≡ x))
  -> Telescope (n + zero - 2)
doSolutionTel p = updateTel₂ p (λ _ -> nil)
  (λ at -> solution) (λ at -> solution') (λ at -> solution'∘solution)
```

We call the function $\texttt{updateTel}_2$, which takes a proof of type `TelAt'` and a telescope xs' that replaces the elements at index k and $k+1$ (in the case of solution this is the empty telescope). Provided we have the functions f, which takes the element at and the two elements at index k and $k+1$ of an interpretation of telescope xs and returns an interpretation of xs', a function f' which takes an interpretation of xs' and returns elements with the types at index k and $k + 1$ of telescope Δ, and a function $f' \circ f$, which proves that f and f' is a section-retraction pair. With this telescope, we can move between an interpretation of Δ and an interpretation

of $\texttt{updateTel}_2$ applied with Δ (using $\texttt{update}_2$ and $\texttt{update'}_2$), provided we supply the same functions for f, f', and $f' \circ f$.

We make sure that B can depend on other elements in the telescope, using type A, such that we can call the solution rule on any data type we want. For example, if we have a telescope containing two natural number n, m, and two equivalence relations $(e_1, e_2 : n \equiv m)$, we could eliminate an equivalence relation $(e : e_1 \equiv e_2)$, by calling the solution rule on e_2 and e. To use $\texttt{update}_2$, we would have to use the type $\mathbb{N} \times \mathbb{N}$ in the place of A, to denote the arguments n and m:

```
_ : ⟦ n ∈ ℕ , m ∈ ℕ , e₁ ∈ n ≡ m , e₂ ∈ n ≡ m , e ∈ e₁ ≡ e₂ , nil ⟧telD
  -> ⟦ n ∈ ℕ , m ∈ ℕ , e ∈ n ≡ m , nil ⟧telD
_ = update₂ (there λ n -> there λ m -> there λ e₁ -> here ((n , m) , e₁))
 (λ a -> nil) (λ xa -> solution) (λ xa -> solution') (λ xa -> solution'∘solution)
```

7.3 Deletion Rule

The deletion rule removes an equation $(e : t \equiv t)$ from the telescope. It thus maps e to the empty telescope:

```
deletion : (e : t ≡ t) -> ⊤
deletion e = tt
```

The function `deletion'` maps an empty telescope back to `refl`:

```
deletion' : ⊤ -> t ≡ t
deletion' _ = refl
```

Now we can prove that applying the result of `deletion'` (`deletion` e) is equivalent to e by directly applying the K-rule:

```
deletion'∘deletion : (e : t ≡ t) -> deletion' (deletion e) ≡ e
deletion'∘deletion e = K (λ e -> deletion' (deletion e) ≡ e) refl e
```

The result of applying the deletion rule to a telescope containing e is that we remove e from the telescope and replace every occurence of e by `refl`. If we have a telescope for which we can create an element $(fa : B\ a)$ at position k for some $(a : A)$, we require the element at position k to be $(e : fa \equiv fa)$, which is captured in proof p. At position k we then remove the element e and replace each occurrence of e by `refl` in the remaining telescope:

```
doDeletionTel : {Δ : Telescope n} {B : A -> Set} (f : (a : A) -> B a)
  (p : Δ [ k ]:Σ[ A ] (λ a -> f a ≡ f a))
  -> Telescope (n - 1)
```

7.4 Conflict Rule

The conflict rule captures that it is not possible to have an equivalence relation where both sides of the equation use different constructors (e.g. `suc` $n \equiv$ `zero`). So, if we have two elements x and y of data type D and a proof f that states that the constructor indices of x and y are not equal (the function $\texttt{con}_i$ retrieves the constructor index of a data type), but we have an element e that states that x and y are equivalent, we result in absurdity as we can eliminate f with e where both sides are applied to $\texttt{con}_i$:

```
conflict : {D : DataDesc cₙ} {x y : μ D} (f : ¬ (conᵢ x ≡ conᵢ y))
  -> (e : x ≡ y) -> Σ[ b ∈ ⊥ ] ⊤
conflict f e = ⊥-elim (f (cong conᵢ e)) , tt
```

The other way around is also possible. The function `conflict'` results in the equation $x \equiv y$ from absurdity b by eliminating b:

```
conflict' : {D : DataDesc cₙ} {x y : μ D} (f : ¬ (conᵢ x ≡ conᵢ y))
    -> Σ[ b ∈ ⊥ ] ⊤ -> x ≡ y
conflict' f b = ⊥-elim b
```

We can prove that applying the result of `conflict'` f (`conflict` f e) is equivalent to e by again eliminating f with e where both sides are applied to $\texttt{con}_i$:

```
conflict'∘conflict : {D : DataDesc cₙ} {x y : μ D} (f : ¬ (conᵢ x ≡ conᵢ y))
    -> (e : x ≡ y) -> conflict' f (conflict f e) ≡ e
conflict'∘conflict f e = ⊥-elim (f (cong conᵢ e))
```

The result of applying the conflict rule to a telescope containing e is that we remove e from the telescope and replace it with an element $(b : \bot)$ and replace every occurence of e with $\bot -$ `elim` b. The conflict rule is used to remove elements of type `zero` $\equiv$ `suc` n where n is defined earlier or in the telescope to be of type natural number, hence creating an element of type $\mu\ D$ may depend on some set A, so we take x and y to be of type $A \to \mu\ D$. Then, at position k in the telescope we should find an element of type $xa \equiv ya$ for some $(a : A)$. Then, if we have a proof that for any $(a : A)$ the constructor indices are not equal, then we can replace the equation $x\ a \equiv y\ a$ at position k with $\bot$:

```
doConflictTel :  {Δ : Telescope n}{D : DataDesc cₙ}{x y : A -> μ D}
  -> (p : Δ [ k ]:Σ[ A ] (λ a -> x a ≡ y a))
  -> (f : (a : A) -> ¬ (conᵢ (x a) ≡ conᵢ (y s))) -> Telescope n
```

7.5 Injectivity Rule

The injectivity rule simplifies an equation where both sides use the same constructor, by equating all constructor arguments instead. That is, we replace the equation with the telescope `injectivityTelC`, which equates all constructor arguments given two elements of that constructor. In the case of $\Sigma'\ S\ C$ we use this equation e to substitute y in ys by x:

```
injectivityTelC : {X : Set}{C : ConDesc aₙ} (x y : ⟦ C ⟧c X) -> Telescope aₙ
injectivityTelC {C = one'} _ _ = nil
injectivityTelC {X = X} {C = Σ' S C } (x , xs) (y , ys)
  = e ∈ x ≡ y , injectivityTelC (subst (λ x -> ⟦ C x ⟧c X) e xs) ys
injectivityTelC {C = ind× C} (x , xs) (y , ys)
  = e ∈ x ≡ y , injectivityTelC xs ys
```

To create this telescope from two elements x and y of data type D, we perform the case-μ operator on both elements and call the `injectivityTelC` function with the $(\pi_1\ x)$th constructor of data type D. Using the proof f which states that the constructors for both elements are the same, we can substitute $(\pi_1\ y)$ by $(\pi_1\ x)$:

```
injectivityTel : {D : DataDesc cₙ} {x y : μ D} (f : conᵢ x ≡ conᵢ y)
  -> Telescope (conₙ x)
injectivityTel {D = D} {x} {y}
  = case-μ D (λ x -> (y : μ D) -> conᵢ x ≡ conᵢ y -> Telescope (conₙ x))
    (λ x -> case-μ D (λ y -> π₁ x ≡ conᵢ y -> Telescope (π₁ (D (π₁ x))))
      (λ y e -> injectivityTelC (π₂ x)
        (subst (λ x -> ⟦ π₂ (D x) ⟧c (μ D)) (sym e) (π₂ y)))) x y
```

If we have a constructor x, we can create an interpretation of telescope `injectivityTelC` between these two constructors by filling in `refl` for each element:

```
injectivityC : {X : Set}{C : ConDesc a_n}(x : ⟦ C ⟧c X)
  -> ⟦ injectivityTelC x x ⟧telD
injectivityC {C = one'} x = tt
injectivityC {C = Σ' S D } (x , xs) = refl , injectivityC xs
injectivityC {C = ind×' C} (x , xs) = refl , injectivityC xs
```

Then, the injectivity rule states that if we have two elements x and y and we have a proof f that states that both constructor indices are the same, we can replace the equivalence $(e : x \equiv y)$ by the telescope of where all constructor arguments of x and y are equivalent. We perform case analysis on x and y and the J-rule on e. Then we need the K rule to replace f by `refl`, which allows us to create the interpretation of telescope from `injectivityC` for constructor x:

```
injectivity : {D : DataDesc a_n} {x y : μ D} (f : con_i x ≡ con_i y) (e : x ≡ y)
  -> ⟦ injectivityTel f ⟧telD
injectivity {D = D} {x} f e
  = J (λ y e -> (f : con_i x ≡ con_i y) -> ⟦ injectivityTel f ⟧telD)
    (λ f -> K (λ f -> ⟦ injectivityTel f ⟧telD)
      (case-μ D (λ x -> ⟦ injectivityTel refl ⟧telD) (λ x -> injectivityC (π2 x))
    x) f) e f
```

We can also derive the equivalence $x \equiv y$ from an interpretation of the `injectivityTelC` telescope, by combining the existing equivalence e in the interpretation of `injectivityTelC` with a recursive call on the remaining telescope. Here, `Σ-create` takes two equations $a_1 \equiv a_2$ and $b_1 \equiv b_2$, with $(a_1, a_2 : A)(b_1 : B\ a_1)(b_2 : B\ a_2)$ and builds a dependent sum equivalence equivalence $(a_1, b_1) \equiv (a_2, b_2)$. `×-create` takes two equations $a_1 \equiv a_2$ and $b_1 \equiv b_2$, with $(a_1, a_2 : A)(b_1, b_2 : B\ a_1)$ and builds a product equivalence equivalence $(a_1, b_1) \equiv (a_2, b_2)$:

```
injectivityC' : {X : Set}{C : ConDesc a_n} -> (x y : ⟦ C ⟧c X)
  -> ⟦ injectivityTelC x y ⟧telD -> x ≡ y
injectivityC' {C = one'} x y e = refl
injectivityC' {X = X} {C = Σ' S D'} (x , xs) (y , ys) (e , es)
  = Σ-create e (injectivityC' (subst (λ s -> ⟦ D' s ⟧c X) e xs) ys es)
injectivityC' {C = ind×' C} (x , xs) (y , ys) (e , es)
  = ×-create e (injectivityC' xs ys es)
```

By again performing case analysis on x and y and using the J-rule on the equivalence between the constructor indices f, we can use `injectivityC'` to retrieve that the constructors x and y are equivalent:

```
injectivity' : {D : DataDesc c_n} {x y : μ D}(f : con_i x ≡ con_i y)
  -> ⟦ injectivityTel f ⟧telD -> x ≡ y
injectivity' {D = D} {x} {y} = case-μ D (λ x -> (y : μ D) -> (f : con_i x ≡ con_i y)
  -> ⟦ injectivityTel f ⟧telD -> x ≡ y) (λ x -> case-μ D (λ y
    -> (f : π1 x ≡ con_i y) -> ⟦ injectivityTel f ⟧telD -> ⟨ x ⟩ ≡ y)
      (λ y f -> J (λ n2 e -> (y : ⟦ π2 (D n2) ⟧c (μ D))
        -> ⟦ injectivityTel e ⟧telD -> ⟨ x ⟩ ≡ ⟨ n2 , y ⟩) (λ y xs ->
          cong (λ xs -> ⟨ π1 x , xs ⟩) (injectivityC' (π2 x) y xs)) f (π2 y))) x y
```

We can then prove that applying the result of `injectivityC'` $x\ x$ (`injectivityC` x) returns `refl` for any element x of constructor C using `refl` for each element x and calling the function recursively for xs when x maps to (x, xs):

```
injectivityC'∘injectivityC : {X : Set}{C : ConDesc a_n} (x : ⟦ C ⟧c X)
  -> injectivityC' x x (injectivityC x) ≡ refl
injectivityC'∘injectivityC {C = one'} x = refl
```

```
injectivityC'∘injectivityC {X = X} {C = Σ' S E} (x , xs)
  = subst (λ e -> Σ-create refl e ≡ e) (sym (injectivityC'∘injectivityC xs)) refl
injectivityC'∘injectivityC {C = ind×' C'} (x , xs)
  = subst (λ e -> ×-create refl e ≡ e) (sym (injectivityC'∘injectivityC xs)) refl
```

We can prove that applying the result of `injectivity'` f (`injectivity` f e) is equivalent to e by again performing case analysis on x and y and performing the J-rule on e. Then, we have use the K-rule to replace f by `refl` which allows us to use `cong` and `injectivityC'∘injectivityC`:

```
injectivity'∘injectivity : {D : DataDesc c_n} {x y : μ D} (f : con_i x ≡ con_i y)
  -> (e : x ≡ y) -> injectivity' f (injectivity f e) ≡ e
injectivity'∘injectivity {D = D} {x} {y} f e
  = J (λ y e -> (f : con_i x ≡ con_i y) -> injectivity' f (injectivity f e) ≡ e)
    (λ f -> K (λ f -> injectivity' f (injectivity f refl) ≡ refl)
      (case-μ D (λ x -> injectivity' refl (injectivity refl refl) ≡ refl)
        (λ x -> cong (cong (λ xs -> ⟨ π_1 x , xs ⟩))
          (injectivityC'∘injectivityC (π_2 x))) x) f) e f
```

The result of applying the injectivity rule to a telescope containing e is that we remove e from the telescope and replace it with the telescope of equivalent constructor arguments `injectivityTel`. We then replace every occurence of e with `injectivity'` applied with this added telescope. The injectivity rule is used to remove elements of type `suc` $n \equiv$ `suc` m where the variable n and m are defined earlier or in the telescope, hence creating an element of type $\mu\, D$ may depend on some set A, so we take x and y to be of type $A \to \mu\, D$. Then, at position k in the telescope we should find an element of type $x\ a \equiv y\ a$ for some $a : A$. Then, if we have a proof f that for any $a : A'$ we have that the constructor indices are equal, we can replace the equation $x\ a \equiv y\ a$ at position k by a telescope of equivalent constructor arguments. But, Agda thinks that the length of this telescope depends on the actual value of $x\ a$ for some $a : A$. In reality, we know that the value of this a cannot alter the constructor index, hence we add an additional proof that states that for any a the number of constructor arguments of a variable $x\ a$ (denoted by the function $\mathtt{con}_{\mathbb{N}}$) is always equal to a natural number a'_n, which is the number of elements we add to the telescope:

```
doInjectivityTel : {Δ : Telescope n}{D : DataDesc c_n}{x y : A -> μ D}
  -> (f : (a : A) -> con_i (x a) ≡ con_i (y a))
  -> {a_n' : ℕ}(eℕ : (a : A) -> con_n (x a) ≡ a_n')
  -> (p : Δ [ k ]:Σ[ A ] (λ a -> x a ≡ y a))
  -> Telescope (n + a_n' - 1)
```

Note that we deviate from the literature here. In previous and related work concerning the efforts of translating functions defined by dependent pattern matching to functions defined by eliminators [14] [11] [35], the conflict and injectivity rule are both defined using the principle of no confusion. The no confusion property captures that constructors are both injective and disjoint. The reason that we do not need to define this property, is because of the specific datatype encoding we use. An element of this encoding is a pair containing a constructor tag and the collected constructor arguments. Hence, we can split up proof of the constructors being disjoint, which only needs to consider the first element of this pair, and injectivity, which only needs the second pair of this pair assuming the first parts are equal. In other presentations of datatypes the constructor and its arguments are seen as an indivisible whole, so injectivity and disjointness need to be handled together.

7.6 Unification Algorithm

We have defined each unification rule individually and showed its effect on an already existing telescope. To make these useable to work with, we define a data type `Unification` that

contains a constructor for each unification rule and perform that unification rule on an arbitrary telescope Δ of length n. We start with a constructor `UEnd` which denotes the end of the unification algorithm:

```
data Unification : (Δ : Telescope n) -> Set₁ where
  UEnd : (Δ : Telescope n) -> Unification Δ
```

Note that because the `UEnd` constructor does not put any requirements on the telescope, there is no guarantee that a unification trace solves all the equations. This is however not a problem, because sometimes you do not have to solve all equivalence relations to get the result you want. For example, in case of the function `head`, we would not need to solve the equivalence `suc` $n \equiv$ `suc` m to retrieve the first element x from the telescope, as it is not dependent on the indices of the vector.

For each unification rule, we add the requirements that is needed to update the telescope (the full specification can be found in `One_Indexed/unify.Agda`). For example, for the solution rule we needed a telescope Δ and and a proof p that at location k we have an element $(x : B\ (\pi_1\ t))$ and at position $k+1$ we have an element $(e : (\pi_2\ t) \equiv x)$. We then require the remaining unification algorithm call on the telescope updated with the `doSolutionTel` function with this proof p:

```
Usolution : {B : A -> Set}
  -> (p : Δ [ k ]:Σ[ Σ[ a ∈ A ] (B a) ]
    (λ t -> A (π₁ t)) : (λ t x -> (π₂ t) ≡ x))
  -> Unification (doSolutionTel p)
  -> Unification Δ
```

In a similar way, we add constructors `UDeletion`, `UConflict`, and `UInjectivity`. We furthermore add a variant of the solution rule that works on an equivalence relation where x and t are reversed. The unification rules that works for this are similar to that of the original solution rule:

```
Usolution₁ : {B : A -> Set}
  -> (p : Δ [ k ]:Σ[ Σ[ a ∈ A ] (B a) ]
    (λ t -> A (π₁ t)) : (λ t x -> x ≡ (π₂ t)))
  -> Unification (doSolutionTel₁ p)
  -> Unification Δ
```

As we require the x and e in the solution rule to be adjacent to each other, we also add a reorder rule that allows one to push back an element at position k in the telescope to a position $goal$ as long as the element at position k does not depend on any elements that come after the $goal$ place:

```
UReorder : (split : Fin i) (goal : Fin j)(p : (x : ⟦ π₁ (splitTel split Δ) ⟧telD)
    -> (Σ[ X ∈ Set ] ((π₂ (splitTel split Δ)) x) [ k ]:Σ[ ⊤ ] (λ _ -> X)))
  -> Unification (reorderTel split Δ goal p)
  -> Unification Δ
```

Following a series of unification rules `Unification` Δ for a telescope Δ, we can retrieve a telescope where these rules are applied in order, by calling the function recursively on each $(u$: `Unification` $\Delta')$. When we reach `UEnd`, we return Δ':

```
unifyTel : {Δ : Telescope n}(u : Unification Δ) -> Σ ℕ Telescope
```

Given a series of unification rules `Unification` Δ and an interpretation of telescope Δ, we can retrieve an interpretation of the unified telescope by following each telescope update rule for the respective unification rule. As we have shown that we can derive each seperate telescope from a unification equation and vice versa, and because we have proven that this results in

the same equation, it is possible to always retrieve the updated telescope from the original and vice versa. Hence, we do now show each individual update rule:

```
unify : {Δ : Telescope n} (u : Unification Δ)(xs : ⟦ Δ ⟧telD)
  -> ⟦ π₂ (unifyTel u) ⟧telD

unify' : {Δ : Telescope n} (u : Unification Δ)(xs : ⟦ π₂ (unifyTel u) ⟧telD)
  -> ⟦ Δ ⟧telD

unify'∘unify : {Δ : Telescope n}(u : Unification Δ)(xs : ⟦ Δ ⟧telD)
  -> unify' u (unify u xs) ≡ xs
```

We can define a unification problem for a telescope that contains a natural number n and an equivalence relation e between the constructors `zero'` and `suc' n` by calling the conflict rule on the second element e, where we know that the constructor index of `zero'` (`f0`) is not equal to the constructor index of `suc' n` (`f1`). The updated telescope then contains the natural number n and an element $b : \bot$:

```
UnifyZero : Unification (n ∈ NI.μ NatD , e ∈ zero' ≡ suc' n , nil)
UnifyZero = UConflict (there λ n -> here n) (λ d ())
  (UEnd (n ∈ NI.μ NatD , b ∈ ⊥ , nil))
```

We can define a unification problem for a telescope that contains the natural number n and m an element $(x : X)$, a vector $(xs : \texttt{Vec}\ X\ m)$, and an equivalence relation $(e : \texttt{suc'}\ m \equiv \texttt{suc'}\ n)$. First, we reorder the telescope such that e is located after element m (that is at position 2). Then, we can call injectivity on e where A is a set containing two natural numbers n and m. We know that the constructor indices of `suc'` are the same so we fill in `refl`, and as we add one equivalence relation, as `suc'` always contains one constructor element, so for $e\mathbb{N}$ we can also fill in `refl`. Then we perform the solution rule on m, e to replace every occurence of m by n and we derive a telescope containing a natural number n, an element $(x : X)$, and a vector $(xs : \texttt{Vec}\ X\ n)$:

```
UnifySuc : Unification (n ∈ NI.μ NatD , m ∈ NI.μ NatD , x ∈ X , xs ∈ Vec X m
  , e ∈ suc' m ≡ suc' n , nil)
UnifySuc = UReorder f2 f0 (λ x -> _ , there λ _ -> there λ _ -> here tt)
 (UInjectivity (there λ n -> there λ m -> here (m , n)) (λ_ -> refl) (λ_ -> refl)
  (Usolution₁ (there λ n -> here n)
   (UEnd (n ∈ NI.μ NatD , x ∈ X , xs ∈ Vec X n , nil))))
```

Chapter 8

Extending to Generic One-Indexed Data Types

We now extend everything from the previous chapters, except for unification, to work for one-indexed data types. One-indexed data types are data types that are indexed on simple data types. In chapter 3 we have seen what simple data types are. An example of a simple data type is that of natural numbers. An example of a one-indexed data type is that of vectors, which is indexed on the simple data type natural numbers:

```
data Vec (A : Set) : (n : ℕ) -> Set where
  nil : Vec A zero
  cons : (n : ℕ) -> A -> Vec A n -> Vec A (suc n)
```

Another example of a one-indexed data type is a natural number indexed on two other natural numbers, which is an inductive definition of equality on numbers:

```
data ℕ₁ : (n₀ n₁ : ℕ) -> Set where
  zero₁ : ℕ₁ zero zero
  suc₁ : {n₀ n₁ : ℕ} -> ℕ₁ n₀ n₁ -> ℕ₁ (suc n₀) (suc n₁)
```

Here, the natural numbers n_0 and n_1 are dependent on the choice of constructor. In the case of `zero`$_1$ we have that both are `zero` and in the case of `suc`$_1$ we have that both are the successor of some other natural numbers.

In this chapter we extend the data type descriptions, elimination principle, and well-founded recursion of chapter 3 to allow for these simple indices (section 8.1). We furthermore extend the telescope of constructor arguments from chapter 4 (section 8.2). Using the unification algorithm from chapter 7 we extend the case tree from chapter 5 to allow us to perform case splits on one-indexed data types (section 8.3) and extend the evaluation function from chapter 6 to work with this case tree (section 8.4).

8.1 One-indexed Data Type Representation

One-indexed data types are data types that are indexed on a variable number of simple data types. These simple data types do not depend on each other, as they are itself not indexed. Therefore, we use an indexed list, or vector, represent the indices of a one-indexed data type. Using a vector allows us to keep track of the number of indices that a one-indexed data type is dependent on. The vector contains elements of the dependent sum type that goes from a natural number, which denotes the number of constructors, to a simple data type description (`NI.DataDesc`), which is indexed on its number of constructors:

```
DVec : ℕ -> Set₁
```

```
DVec = Vec (Σ ℕ NI.DataDesc)
```

With this we can describe any vector of indices we want. For example, if we want to describe the `Vec` data type in our universe of data type descriptions, we can describe the vector of indices to contain only one natural number. We use the natural number definition from chapter 3:

```
VecTel : DVec 1
VecTel = (_ , NatD) :: []
```

An interpretation of a vector of indices contains fix-point elements of the simple data type (`NI.`μ). Because the indices are not dependent on each other, we use the product type:

```
⟦_⟧Vec : (is : DVec iₙ) -> Set
⟦_⟧Vec [] = ⊤
⟦_⟧Vec ((dₙ , D) :: is) = (NI.μ D) × (⟦ is ⟧Vec)
```

For example, the interpretation of the vector of indices for constructor `nil` is (`zero'`, `tt`). The interpretation of the vector of indices for constructor `suc` is (`suc'` n, `tt`) for a natural number n that is defined as a constructor argument:

```
nilVec : ⟦ VecTel ⟧Vec
nilVec = zero' , tt

nilVec : NI.μ NatD -> ⟦ VecTel ⟧Vec
nilVec n = suc' n , tt
```

Now, we have to extend the constructor description from chapter 3 to allow for a constructor to denote its specific indices. For example, for the vector data type we need to differentiate between n is `zero` for constructor `nil` and `suc` n for constructor `cons`. To allow this, we add an extra argument to the constructor `one'`, that contains an interpretation of the vector of indices is, which we add as an argument to the constructor description as the type of the indices will never change. For inductive arguments (denoted by $\times'$) we also need to specify the values of the indices with which the data type is called:

```
data ConDesc (is : DVec iₙ) : ℕ -> Set₁ where
  one' : ⟦ is ⟧Vec -> ConDesc is 0
  Σ'   : (S : Set)(D : S -> ConDesc is aₙ) -> ConDesc is (suc aₙ)
  ×'   : ⟦ is ⟧Vec -> ConDesc is aₙ -> ConDesc is (suc aₙ)
```

The data type description remains the same, except that we now add the vector of indices of that data type as an arguments:

```
DataDesc : DVec iₙ -> ℕ -> Set₁
DataDesc is cₙ = Fin cₙ -> Σ ℕ (ConDesc is)
```

With this, we can describe any one-indexed data type. For example, we can describe the vector data type, which contains two constructors, given a parameter X that denotes the type of the elements in the vector data type. For constructor `nil'` (denoted by `f0`), we have zero constructor arguments, so we call the empty constructor `one'`, which now requires an additional argument containing an interpretation of the `VecTel`, for which we use `nilVec`. For constructor `cons'` (denoted by `f1`) we have 3 arguments, of which one is an inductive argument. So we use Σ' for the natural number n and an element $(x : X)$. For the inductive argument we use $\times'$, which requires an additional argument stating the indices of the inductive call, for which we use $(n,$ `tt`$)$. Then we reach the end of the constructor description, so we use `one'`, which again requires an interpretation of `VecTel`, for which we use `consVec` applied with n:

```
VecD : (X : Set) -> DataDesc VecTel 2
VecD X f0 = _ , one' (zero' , tt)
```

```
VecD X f1 = _ , Σ' (NI.μ NatD) (λ n -> Σ' X (λ x ->
  ×' (n , tt) (one' (suc' n , tt))))
```

An action X on a data type description is dependent on the interpretation of its vector of data type indices, because when we reach an inductive call we need to add the interpretation of the vector of indices is on that action. Furthermore, if we reach the empty constructor we need to make sure that the expected indices d are equivalent to the indices t that we have tried to perform the action on:

```
⟦_⟧ : DataDesc is cn -> (⟦ is ⟧Vec -> Set) -> ⟦ is ⟧Vec -> Set
⟦_⟧ {cn} D X t = Σ[ ci ∈ Fin cn ] (⟦ π2 (D ci) ⟧c X t)

⟦_⟧c : ConDesc is an -> (⟦ is ⟧Vec -> Set) -> ⟦ is ⟧Vec -> Set
⟦ one' d ⟧c X t = d ≡ t
⟦ Σ' S D ⟧c X t = Σ[ s ∈ S ] ( ⟦ D s ⟧c X t)
⟦ ×' d D ⟧c X t = X d × ⟦ D ⟧c X t
```

To denote an element of the data type description we again use its fixpoint. However, now an element of a certain data type is dependent on the variables of its indices. Hence, we add an interpretation of the telescope of data type indices, which we pass to the action on the data type description:

```
data μ (D : DataDesc is cn) : ⟦ is ⟧Vec -> Set where
  ⟨_⟩ : {d : ⟦ is ⟧Vec} (x : ⟦ D ⟧ (μ D) d) -> μ D d
```

Using this, we can create elements of the `Vec` data type. For the `nil'` constructor we do not have any constructor arguments, but as a result we get a vector with index `zero'`. If we call the first constructor (using `f0`), we can show that the expected index `zero'` is equivalent to the index of constructor `nil'` (namely also `zero'`) by calling `refl`:

```
nil' : (X : Set) -> μ (VecD X) (zero' , tt)
nil' X = ⟨ f0 , refl ⟩
```

For the `cons'` constructor we need a natural number n, an element $(x : X)$ and another vector xs which is indexed on n. As a result we get a vector of length `suc'` n. If we call the second constructor (using `f1`), we can then show we that the expected index `suc'` n is equivalent to the index of constructor `cons'` (namely also `suc'` n) by calling `refl`:

```
cons' : (X : Set)(n : NI.μ NatD)(x : X)(xs : μ (VecD X) (n , tt))
  -> μ (VecD X) (suc' n , tt)
cons' X n x xs = ⟨ f1 , (n , x , xs , refl) ⟩
```

We can now define the `head`-function for vectors that, given a vector xs containing at least one element (i.e. with a non-zero length), returns the first element of that vector. The only constructor that first at the place of xs is `cons'` as we have specified that the length cannot be `zero'`, so we only need one case, and return the x in that constructor:

```
head' : (X : Set)(n : NI.μ NatD)(xs : μ (VecD X) (suc' n , tt)) -> X
head' X n (cons' n x xs) = x
```

The elimination principle now works on a certain element x of type $\mu\ D\ d$, where d is an interpretation of the vector of indices. Hence, we extend the predicate P and proof p to work for a variable interpretation of the vector of indices d:

```
elim-μ : (D : DataDesc is cn) (P : (d : ⟦ is ⟧Vec) -> μ D d -> Set)
  -> (p : (d : ⟦ is ⟧Vec) (x : ⟦ D ⟧ (μ D) d) -> All D (μ D) P d x -> P d ⟨ x ⟩)
  -> (d : ⟦ is ⟧Vec) -> (x : μ D d) -> P d x
```

This also means that we need to extend the `All` data type and `all` function to contain this variable d, which otherwise bears similarities with the data type defined in chapter 3. Similarly, we create the generic case-eliminator by dropping the recursive arguments `All D (μ D) P d x` from the elimination principle.

Extending the `Below` type is done similary, by adding a variable d that denotes the interpretation of the data type indices for an element of type $\mu\ D\ d$ to the predicate P. Which leads us to extending the function `below` to add this variable to the predicate P and proof p as well.

```
Below : {D : DataDesc is cn} (P : (d : ⟦ is ⟧Vec) -> μ D d -> Set)
  -> (d : ⟦ is ⟧Vec) -> μ D d -> Set
```

8.2 Extending Telescope of Constructor Arguments

To replace a variable in the telescope by the arguments of a specific constructor after a case split, we need to extend the telescope of constructor arguments from chapter 4 to contain the indices that the variable expects (e.g. if we replace a vector with index `suc n` we do not want to replace it with the `nil` constructor, which has index `zero`). We need this to be able to infer that `one'` holds when we translate an interpretation of this telescope to an interpretation of the constructor. Hence, we add an additional argument that contains the indices that the variable expects and compare those to the indices stored in the end of the constructor:

```
conTel : {is : DVec in}(X : ⟦ is ⟧Vec -> Set) -> ConDesc is an
  -> ⟦ is ⟧Vec -> Telescope (an + in)
conTel X (one' i') i = vecTel i' i
conTel X (Σ' S C ) i = s ∈ S , conTel X (C s) i
conTel X (×' i' C) i = x ∈ X i' , conTel X C i
```

In `vecTel` we state that each index from both vectors should be equivalent to each other:

```
vecTel : {is : DVec in} (d1 d2 : ⟦ is ⟧Vec) -> Telescope in
vecTel {is = []} tt tt = nil
vecTel {is = d :: ds} (d1 , ds1) (d2 , ds2) = e ∈ (d1 ≡ d2) , vecTel ds1 ds2
```

When translating a telescope of constructor arguments to a constructor, we then need to show that the index vectors d_1 and d_2 are equivalent to each other, given a telescope that the index of both vectors are equivalent to each other. To do this, we use the function $\times - \texttt{create}$ that takes two equations $a_1 \equiv a_2$ and $b_1 \equiv b_2$ and builds a product equivalence $(a_1, b_1) \equiv (a_2, b_2)$:

```
telToCon : {X : ⟦ is ⟧Vec -> Set} {C : ConDesc is an}{i : ⟦ is ⟧Vec}
  -> (t : ⟦ conTel X C i ⟧telD) -> ⟦ C ⟧c X i

telToVec : {is : DVec in}{d1 d2 : ⟦ is ⟧Vec}(t : ⟦ vecTel d1 d2 ⟧telD)
  -> d1 ≡ d2
telToVec {is = [] } tt = refl
telToVec {is = d :: ds} (t , ts) = ×-create t (telToVec ts)
```

When translating a constructor to a telescope, we have an equation $d' \equiv d$ at the end of the constructor description, from that we can create a telescope of indices by using the functions $\texttt{proj}\times_1$ and $\texttt{proj}\times_2$ that given an equation $(a_1, b_1) \equiv (a_2, b_2)$ return $a_1 \equiv a_2$ and $b_1 \equiv b_2$, respectively:

```
conToTel : {X : ⟦ is ⟧Vec -> Set} {C : ConDesc is an}{i : ⟦ is ⟧Vec}
  -> ⟦ C ⟧c X i -> ⟦ conTel X C i ⟧telD
```

```
vecToTel : {is : DVec iₙ}{d₁ d₂ : ⟦ is ⟧Vec}(e : d₁ ≡ d₂) -> ⟦ vecTel d₁ d₂ ⟧telD
vecToTel {is = [] } e = tt
vecToTel {is = d :: ds} e = proj×₁ e , (vecToTel (proj×₂ e))
```

Now, all that is left to prove is that `telToVec` $\circ$ `vecToTel` holds. If we reach the empty vector of indices, we have to prove that `refl` $\equiv e$ where e : `tt` $\equiv$ `tt`. To do this, we use a variant of the J-rule that, instead of proving that $P\ y\ e$ holds if $P\ x$ `refl` holds, proves that $P\ x$ `refl` holds if $P\ y\ e$ holds. For a non-empty vector of indices, we have to prove that the following statement holds:

$$\times - \texttt{create}\ (\texttt{proj}\ \times_1\ e)\ (\texttt{telToVec}\ (\texttt{vecToTel}\ (\texttt{proj}\ \times_2\ e))) \equiv e$$

We can substitute `telToVec` (`vecToTel` (`proj` $\times_2\ e$)) by `proj` $\times_2\ e$ by calling the function recursively with `proj` $\times_2\ e$ and then use a function `create` $\circ$ `proj`$\times$ that takes an equivalence $(e : (a_1, b_1) \equiv (a_2, b_2))$ and proves that creating a dependent sum from the first and second projection of e is equivalent to the original e:

```
telToCon∘conToTel : {X : ⟦ is ⟧Vec -> Set} {C : ConDesc is aₙ}{i : ⟦ is ⟧Vec}
  -> (t : ⟦ C ⟧c X d) -> telToCon (conToTel t) ≡ t

telToVec∘vecToTel : {is : DVec iₙ}{d₁ d₂ : ⟦ is ⟧Vec} (e : d₁ ≡ d₂)
  -> telToVec (vecToTel e) ≡ e
telToVec∘vecToTel {is = []} {d₁ = tt} {d₂ = tt} e = J' (λ _ e' -> e' ≡ e) e refl
telToVec∘vecToTel {is = d :: ds} {d₁ = (d₁ , ds₁)} {d₂ = (d₂ , ds₂)} e
  = subst (λ f -> ×-create (proj×₁ e) f ≡ e)
    (sym (telToVec∘vecToTel (proj×₂ e))) (create∘proj× e)
```

8.3 Extending the Case Tree

We extend the case tree from chapter 5 to allow case splits only on one-indexed data types (as we can write simple indexed data types also as one-indexed data types but with an empty vector of indices). This means that the type of the case tree and the leaf constructor stays the same, but we have to alter the proof p, such that we there is an element in telescope Δ at position k of type $\mu\ D\ d$ for some interpretation of the vector of indices (d : $[\![is]\!]$`Vec`). The function bs that splits the case tree in c_n branches now maps to a dependent sum type, which takes a unification algorithm u that is based on the extended telescope (which now contains an equivalence relation between each element in d and the expected indices of the target constructor). From this unification algorithm, we require a new case tree which takes the unified telescope. As the updated telescope is unified, we not only have to shrink $args$, but also reverse the unification process using the function `unify'`:

```
data CaseTree (Δ : Telescope n)(T : ⟦ Δ ⟧telD -> Set) : Set₁ where
  leaf : (t : (args : ⟦ Δ ⟧telD) -> T args) -> CaseTree Δ T
  node : {D : DataDesc is cₙ} (p : Δ [ k ]:Σ[ ⟦ is ⟧Vec ] (μ D))
    -> (bs : (cᵢ : Fin cₙ)
        -> Σ[ u ∈ Unification (expandTel Δ (conTel (μ D) (π₂ (D cᵢ))) p
            (λ args -> ⟨ cᵢ , telToCon args ⟩))]
          (CaseTree (π₂ (unifyTel u)) (λ args -> T (shrink p (unify' u args)))))
    -> CaseTree Δ T
```

We can now define the case tree for the `head`-function, where the telescope of input arguments consists of a natural number n and a vector xs of non-zero length:

```
headΔ : (X : Set) -> Telescope 2
headΔ X = n ∈ NI.μ NatD , xs ∈ μ (VecD X) (suc' n , tt) , nil
```

The return type is simply X as it does not depend on the telescope of input arguments:

```
headT : (X : Set) -> ⟦ headΔ X ⟧telD -> Set
headT X _ = X
```

To create the case tree for the head function, we perform a case split on the vector xs, which has as index vector (`suc' n`, `tt`). If we split to the case `nil'` (denoted by `f0`), we have a telescope containing (n : `NI.`μ `NatD`) and (e : `zero' ≡ suc' n`). We can perform the unification function `UnifyZero` from chapter 7 and then enter a leaf node, where as return function we can eliminate the branch by calling $\bot$ − `elim` on the element (b : $\bot$). If we split to the case `cons'` (denoted by `f1`), we can use the function `UnifySuc` to retrieve a telescope containing (n : `NI.`μ `NatD`), (x : X), and (xs : μ (`VecD X`) (`suc' n , tt`)) in the leaf node, where we can simply return x:

```
CTHeadRoot : (X : Set) -> CaseTree (headΔ X) (headT X)
CTHeadRoot X = node (there λ n -> here (suc' n , tt)) (λ where
  f0 -> UnifyZero , leaf (λ where (n , b , _)      -> ⊥-elim b)
  f1 -> UnifySuc  , leaf (λ where (n , x , xs , _) -> x))
```

8.4 Extending the Translation

To extend the evaluation function, we use the updated case-μ operator when evaluating an internal node. We furthermore add that the indices d' are equivalent to the indices of the element at position k in the telescope (denoted by d):

```
eval {T} (node {is} {D} p bs) args
  = case-μ D (λ d' x' -> (d' , x') ≡ (d , ret) -> T args) cs d ret refl
```

To create the function cs, we have to update the property q as the data type indices of c are not necessarily equivalent to those of ret. Hence, the type of q contains an extra substitution property. To solve this, we use the J-rule to substitute e by `refl` (we replace some parts of the code by _ for readability):

```
q : ⟨ cᵢ , telToCon (conToTel (subst (⟦ π₂ (D cᵢ) ⟧c (μ D)) (cong π₁ e) c)) ⟩ ≡ ret
q = J _ (cong (λ x -> ⟨ k , x ⟩) (telToCon∘conToTel x)) e
```

This problem extends to the property r as well. But for r we also have to account for the unification algorithm of the data type indices. Hence, we extend its proof by calling `unify` on u and the expanded telescope, where u is retrieved using π_1 ($bs\ c_i$):

```
r : T (shrink p (unify' u (unify u (expand p _ _
  (conToTel (subst (⟦ π₂ (D cᵢ) ⟧c (μ D)) (cong π₁ e) c)) q))))
r = eval (π₂ (bs cᵢ)) (unify u (expand p _ _ _ q))
```

To create cs, we can then call r, by substituting $args$ using `unify'` ∘ `unify` and `shrink` ∘ `expand`:

```
cs : (d' : ⟦ is ⟧Vec) (x' : ⟦ D ⟧ (μ D) d') -> (d' , ⟨ x' ⟩) ≡ (d , ret) -> T args
cs d' (cᵢ , c) e = subst T (subst _ (unify'∘unify _ _) (shrink∘expand _ _ _ q)) r
```

We can show that the computational behaviour of the evaluation function on the `head`-function is sound by showing that, for any natural number n and vector v of length `suc' n`, the evaluation function applied with the case tree from the previous section evaluates to the same result as the `head'`-function defined by pattern matching:

```
≡Head : {X : Set}(n : NI.μ NatD)(v : μ (VecD X) (suc' n , tt))
  -> eval (CTHeadRoot X) (n , v , tt) ≡ head' X n v
≡Head n (cons' n x xs) = refl
```

Chapter 9

Extending to Generic Indexed Data Types

We now extend everything from the previous chapters to work for indexed data types. Indexed data types are data types that can be indexed on any other data type (including other indexed data types). An example of this is the `Square` data type, which is indexed on the data type _ ≡ _, which itself is indexed on a variable of type A:

```
data _≡_ {A : Set} (a : A) : A -> Set where
  idp : a ≡ a
```

The `Square` data type is then indexed on four equivalence relations (its edges) between the four points a, b, c, d:

```
data Square {A : Set} {a : A} : {b c d : A}(p : a ≡ b)(q : c ≡ d)
    -> (r : a ≡ c) (s : b ≡ d) -> Set where
  ids : Square {a = a} idp idp idp idp
```

In this chapter we extend everything from the previous chapter to work for indexed data types (section 9.1). We furthermore extend the deletion and injectivity rules from chapter 7 to work for these indexed data types (section 9.2), as the solution and deletion rule already work for any type, and we show that we can extend the work to incorporate new concepts like higher-dimensional unification (section 9.3).

9.1 Generic Indexed Data Types Representation

Indexed data types are data types that are indexed on a variable number of other data types that may be indexed itself, hence the data types that the data type is indexed on may depend on each other. Therefore, we use a telescope Δ to represent the indices of an indexed data type. Using the telescope data type furthermore allows us to keep track of the number of indices that the indexed data type is dependent on:

```
DataDesc : Telescope iₙ -> ℕ -> Set₁
DataDesc is cₙ = Fin cₙ -> Σ ℕ (ConDesc is)
```

An interpretation of the telescope of indices is the same as the interpretation of any other telescope. To extend the constructor description of the previous chapter, we thus only have to replace the interpretation of the vector of indices by an interpretation of the telescope of indices:

```
data ConDesc (is : Telescope iₙ) : ℕ -> Set₁ where
  one' : ⟦ is ⟧telD -> ConDesc is 0
  Σ'   : (S : Set) (D : S -> ConDesc is aₙ) -> ConDesc is (suc aₙ)
  ×'   : ⟦ is ⟧telD -> ConDesc is aₙ -> ConDesc is (suc aₙ)
```

With this, we can describe any indexed data type. For example, we can describe the indexed data type ≡D from the beginning of this chapter, parameterized on an element $(a : A)$ and indexed on another element $(a' : A)$, with one constructor where a' is set to be a:

```
≡D : A -> DataDesc (a ∈ A , nil) 1
≡D a f0 = _ , one' (a , tt)
```

Note that we could define this data type in the previous sections only if A was replaced by a specific non-indexed data type. For example, we could define could define ≡NatD or ≡ListD, but not this generic version. The fixpoint of an indexed data type remains the same as the previous section, except that we change the interpretation of the vector of indices to an interpretation of the telescope of indices. The action on an element for this data type also remains the same. We can create an element of type $\mu\ (\equiv\texttt{D}\ a)\ (a, \texttt{tt})$ for any $(a : A)$ by calling f0 with refl:

```
idp : (a : A) -> μ (≡D a) (a , tt)
idp a = ⟨ f0 , refl ⟩
```

We can describe a data type SquareD that is indexed on this ≡D type, which is parameterized by an element $(a : A)$ and indexed on 3 elements $(b, c, d : A)$ and 4 equivalence relations between these elements and a. It consists of only one constructor where the interpretation of the telescope of indices consist of the idp type:

```
SquareD : (a : A) -> DataDesc (b ∈ A , c ∈ A , d ∈ A , p ∈ μ (≡D a) (b , tt) ,
 q ∈ μ (≡D c) (d , tt) , r ∈ μ (≡D a) (c , tt) , s ∈ μ (≡D b) (d , tt) , nil) 1
SquareD a f0 = _ , one' (a , a , a , idp a , idp a , idp a , idp a , tt)
```

We can create an element of type $\mu\ (\texttt{SquareD}\ a)\ (a, a, a, \texttt{idp}\ a, \texttt{idp}\ a, \texttt{idp}\ a, \texttt{idp}\ a, \texttt{tt})$ for any $(a : A)$ by calling f0 with refl:

```
ids : (a : A) -> μ (SquareD a) (a , a , a , idp a , idp a , idp a , idp a , tt)
ids a = ⟨ f0 , ⟩
```

We can create a function flip on the SquareD data type, that given an element $(w : A)$ flips the order of the points $(w, x, y, z : A)$ to w, y, x, z by pattern matching on the input square, which changes all elements x, y, z to w as t, b, l, r states that they should all be equal to w. We can then create an element of $\mu\ (\texttt{SquareD}\ w)\ (w, w, w, \texttt{idp w}, \texttt{idp w}, \texttt{idp w}, \texttt{idp w}, \texttt{tt})$ by calling constructor ids with w:

```
flip : (w x y z : A) (t : μ (≡D w) (x , tt)) (b : μ (≡D y) (z , tt))
  -> (l : μ (≡D w) (y , tt)) (r : μ (≡D x) (z , tt))
  -> μ (SquareD w) (x , y , z , t , b , l , r , tt)
  -> μ (SquareD w) (y , x , z , l , r , t , b , tt)
flip w w w w (idp w) (idp w) (idp w) (idp w) (ids w) = ids w
```

The elimination principle and Below type from the previous chapter also stays the same, except that we replace an interpretation of the vector of indices with an interpretation of the telescope of indices. The case-μ operator, for example, then gets the following type:

```
case-μ : (D : DataDesc is cₙ)(P : (d : ⟦ is ⟧telD) -> μ D d -> Set)
  -> (m : (d : ⟦ is ⟧telD) -> (x : ⟦ D ⟧ (μ D) d) -> P d ⟨ x ⟩)
  -> (d : ⟦ is ⟧telD) (x : μ D d) -> P d x
```

9.1.1 Extending Telescope of Constructor Arguments

To extend the telescope of constructor arguments from the previous chapter, we need to alter the telescope of equivalent indices. That is, the function vecTel from chapter 8 was based on the fact that each index was not dependent on each other. In a telescope of indices an index

may depend on the previously declared indices. Hence, we have use the previous equation e in `vecTel` to substitute the index in the remaining telescope of indices:

```
vecTel : {is : Telescope iₙ} (d₁ d₂ : ⟦ is ⟧telD) -> Telescope iₙ
vecTel {is = nil     } tt tt = nil
vecTel {is = cons S E} (d₁ , ds₁) (d₂ , ds₂) = e ∈ (d₁ ≡ d₂) ,
  vecTel (subst (λ s -> ⟦ E s ⟧telD) e ds₁) ds₂
```

This furthermore requires us to update the proofs for `telToVec`, `vecToTel`, and `telToVec ∘ vecToTel` by replacing $\times -$`create` with $\Sigma -$`create` (which builds an equivalence between two dependent sum types), replacing `proj`$\times_1$ and `proj`$\times_2$ by `proj`Σ_1 and `proj`Σ_2 (which projects out an equivalence from an equivalence between two dependent sum types), respectively, and replacing `create` $\circ$ `proj`$\times$ by `create` $\circ$ `proj`Σ (which shows that creating an equivalence between two dependent sum types from the projections out of an equivalence between two dependent sum types results again in the original equivalence).

9.1.2 Extending the Case Tree and Evaluation Function

We extend the case tree from the previous chapter to allow case splits only on indexed data types. This means that we have to replace the instance of the vector of indices of the data type we perform a case split on, by an interpretation of the telescope of indices. We still want to perform unification on these indices, so the remainder of the case tree remains the same:

```
data CaseTree (Δ : Telescope n)(T : ⟦ Δ ⟧telD -> Set) : Set₁ where
  leaf : (t : (args : ⟦ Δ ⟧telD) -> T args) -> CaseTree Δ T
  node : {D : DataDesc is cₙ} (p : Δ [ k ]:Σ[ ⟦ is ⟧telD ] (μ D))
    -> (bs : (cᵢ : Fin cₙ)
      -> Σ[ u ∈ Unification (expandTel Δ (conTel (μ D) (π₂ (D cᵢ))) p
            (λ args -> ⟨ cᵢ , telToCon args ⟩))]
          (CaseTree (π₂ (unifyTel u)) (λ args -> T (shrink p (unify' u args)))))
    -> CaseTree Δ T
```

We can define the case tree for the `split`-function, where we parameterize on $(w : A)$. The telescope of input arguments consists of $(x, y, z : A)$, the equivalence relations t, b, l, r between w and x, y, z, and a square s containing these arguments:

```
ΔFlip : (w : A) -> Telescope 8
ΔFlip {A = A} w = x ∈ A , y ∈ A , z ∈ A , t ∈ μ (≡D w) (x , tt) ,
  b ∈ μ (≡D y) (z , tt) , l ∈ μ (≡D w) (y , tt) , r ∈ μ (≡D x) (z , tt) ,
  s ∈ μ (SquareD w) (x , y , z , t , b , l , r , tt) , nil
```

The return type is based on these arguments, where we replace x in the square by y and y by x. This also means that we have to replace the order of equivalences to l, r, t, b instead of t, b, l, r:

```
TFlip : (w : A) -> ⟦ ΔFlip w ⟧telD -> Set
TFlip w (x , y , z , t , b , l , r , _)
  = μ (SquareD w) (y , x , z , l , r , t , b , tt)
```

To create the case tree for the `split`-function, we perform a case split on the square s, which has as index vector `(x , y , z , t , b , l , r , tt)`. There is only one case we can split to, namely `ids`, where we get a telescope containing the following elements:

$$x \in A, y \in A, z \in A, t \in \mu\ (\equiv D\ w)\ (x, tt), b \in \mu\ (\equiv D\ y)\ (z, tt), l \in \mu\ (\equiv D\ w)\ (y, tt),$$
$$r \in \mu\ (\equiv D\ x)\ (z, tt), ewx \in w \equiv x, ewy \in w \equiv y, ewz \in w \equiv z,$$
$$ewt \in \texttt{idp}\ w \equiv t, ewb \in \texttt{idp}\ w \equiv b, ewl \in \texttt{idp}\ w \equiv l, ewr \in \texttt{idp}\ w \equiv r, nil$$

We can use the solution rule on each element i together with the equivalence relation ewi, eventually reducing to an empty telescope, which we store in the function `unifyFlip`. In the return type, we thus replace every element of type A by w, and every element of the equivalence type by `ids` w. The return type thus requires an element of type

$$\mu\ (\texttt{SquareD}\ w)(w, w, w, \texttt{ids}\ w, \texttt{ids}\ w, \texttt{ids}\ w, \texttt{ids}\ w, \texttt{tt})$$

So, we can call a leaf node and fill in `ids w`:

```
CTFlip : (w : A) -> CaseTree (ΔFlip w) (TFlip w)
CTFlip w = node (there λ x -> there λ y -> there λ z -> there λ t ->
    there λ b -> there λ l -> there λ r -> here (x , y , z , t , b , l , r , tt))
  (λ where f0 -> unifyFlip , leaf (λ _ -> ids w))
```

The evaluation function stays the same as that of the previous chapter. We can show that the computational behaviour of the evaluation function on the `split`-function is sound by showing that for any interpretation of the telescope of input arguments, the evaluation function applied with the case tree evaluates to the same element as the function defined by pattern matching:

```
≡Flip : (w x y z : A) (t : μ (≡D w) (x , tt)) (b : μ (≡D y) (z , tt))
  -> (l : μ (≡D w) (y , tt)) (r : μ (≡D x) (z , tt))
  -> (s : μ (SquareD w) (x , y , z , t , b , l , r , tt))
  -> eval (CTFlip w) (x , y , z , t , b , l , r , s , tt) ≡ flip w x y z t b l r s
≡Flip w w w w (idp w) (idp w) (idp w) (idp w) (ids w) = refl
```

9.2 Extending Unification Rules

In the example from the previous section, we only need to use the solution rule to unify the data type indices. But, the solution and deletion rule already work for any type, so we did not have to change these to work for indexed data types. However, the conflict and injectivity rule work for specific data types. Hence, in this section we update these rules to work for indexed data types.

9.2.1 The Conflict Rule

We can index on other indexed data types. For example, we can have a data type that is indexed on a vector. The vector data type can be represented similar as in chapter 8, except that we replace the vector containing a natural number with a telescope containing a natural number. We can also represent natural numbers as an indexed data type, by filling in the empty telescope for is:

```
VecD : (X : Set) -> DataDesc (n ∈ μ NatD tt , nil) 2
VecD X f0 = _ , one' (zero' , tt)
VecD X f1 = _ , Σ' (μ NatD tt) (λ n -> Σ' X (λ x -> ×' (n , tt)
  (one' (suc' n , tt))))
```

If we perform a case split on an element of a data type that is indexed on a vector, we might reach a unification problem that wants to replace a vector of length at least one, by a vector of zero length (see element e_2). This problem would require another equation that states that the telescope of indices of both vectors are equivalent (see element e_1):

```
Δconflict : (X : Set) -> Telescope 5
Δconflict X = n ∈ μ NatD tt , x ∈ X , xs ∈ μ (VecD X) (n , tt) ,
  e₁ ∈ (suc' n , tt) ≡ (zero' , tt) ,
  e₂ ∈ (subst (μ (VecD X)) e₁ (cons' n x xs)) ≡ nil' , nil
```

Now, our current definition of `conflict` is not sufficient, as we work with elements of data types that are indexed on some vector d. Hence, if we want two elements to be equivalent, their indices should also be equivalent. Therefore, we have to define the `conflict` rule with equation e that combines these elements in a dependent sum type. With this, we can project out the second element of both dependent sums to get the equivalence that negates f:

```
conflict : {D : DataDesc is cₙ}{d₁ d₂ : ⟦ is ⟧telD}{x : μ D d₁}{y : μ D d₂}
  -> (f : ¬ (conᵢ x ≡ conᵢ y))(e : Σ[ e ∈ d₁ ≡ d₂ ] (subst (μ D) e x ≡ y))
  -> Σ[ b ∈ ⊥ ] ⊤
conflict f (ed , ex)
  = ⊥-elim (f (cong (λ dx -> conᵢ (proj₂ dx)) (Σ-create ed ex))) , tt
```

We thus require a proof p_1 that states that d_1 and d_2 are equivalent and, given p_1, a proof p_2 that states that x and y are equivalent, then if the constructor numbers are not equivalent we can use the conflict rule to replace p_1 and p_2 in the telescope by $\bot$:

```
UConflict : {D : DataDesc is cₙ} {d₁ : A -> ⟦ is ⟧telD} {d₂ : A -> ⟦ is ⟧telD}
  -> {x : (a : A) -> μ D (d₁ a)} {y : (a : A) -> μ D (d₂ a)}
  -> (p : Δ [ k ]:Σ[ A ] (λ a -> d₁ a ≡ d₂ a) : (λ a e ->
    subst (μ D) e (x a) ≡ y a)) (f : (a : A) -> ¬ (conᵢ (x a) ≡ conᵢ (y a)))
  -> Unification (doConflictTel p f) -> Unification Δ
```

So, if we want to solve the unification problem Δ`conflict` we gave earlier, we can simply use the conflict rule on the 3rd position in the telescope, giving us a telescope containing only n, x , xs, and an element of $\bot$:

```
unifyConflict : (X : Set) -> Unification (Δconflict X)
unifyConflict X = UConflict (there λ n -> there λ x -> there λ xs ->
  here (n , x , xs)) (λ x ()) (UEnd
    (n ∈ μ NatD tt , x ∈ X , xs ∈ μ (VecD X) (n , tt) , b ∈ ⊥ , nil))
```

9.2.2 The Injectivity Rule

Another unification problem we might get when performing a case split on an element of a data type that is indexed on a vector, is replacing two vectors of length at least one (see element e_2), which again requires another equation that states that the telescope of indices of both vectors are equivalent (see element e_1):

```
Δinjectivity : (X : Set) -> Telescope 8
Δinjectivity X = n ∈ μ NatD tt , m ∈ μ NatD tt , x ∈ X , y ∈ X ,
  xs ∈ μ (VecD X) (n , tt) , ys ∈ μ (VecD X) (m , tt) ,
  e₁ ∈ (suc' n , tt) ≡ (suc' m , tt) ,
  e₂ ∈ subst (μ (VecD X)) e₁ (cons' n x xs) ≡ cons' m y ys , nil
```

Now, our current definition of `injectivity` is not sufficient, as we again work with elements of data types that are indexed on some vector d. Hence, if we want two elements x and y to be equivalent, their indices should also be equivalent d_1 and d_2, respectively. Therefore, we have to define the `injectivity` rule with equation e that combines these elements in a dependent sum type, where `injectivityTel` is similar to the function defined in chapter 7, except that it works for indexed data types:

```
injectivity : {D : DataDesc is cₙ}{d₁ d₂ : ⟦ is ⟧telD}{x : μ D d₁}{y : μ D d₂}
  -> (f : conᵢ x ≡ conᵢ y)(e : Σ[ e ∈ d₁ ≡ d₂ ] (subst (μ D) e x ≡ y))
  -> ⟦ injectivityTel f ⟧telD
```

We then again require a proof p_1 that states that d_1 and d_2 are equivalent and, given p_1, a proof p_2 that states that x and y are equivalent, then we can use the same requirements for the injectivity rule from the chapter 7 to replace p_1 and p_2 by `injectivityTel`:

```
UInjectivity : {D : DataDesc is c_n} {d_1 : A -> ⟦ is ⟧telD} {d_2 : A -> ⟦ is ⟧telD}
  -> {x : (a : A) -> μ D (d_1 a)} {y : (a : A) -> μ D (d_2 a)}
  -> (p : Δ [ k ]:Σ[ A ] (λ a -> d_1 a ≡ d_2 a) : (λ a e ->
    subst (μ D) e (x a) ≡ y a)){a_n' : ℕ}(e ℕ : (a : A) -> con_n (x a) ≡ a_n')
  -> (f : (a : A) -> con_i (x a) ≡ con_i (y a))
  -> Unification (doinjectivityTel e ℕ p f) -> Unification Δ
```

In the functions of the injectivity rule, we combine the proofs in e (using Σ-`create`) and apply that to the J-rule. In the converse of injectivity rule, we can furthermore derive that d_1 is equivalent to d_2 from an interpretation of `injectivityTel`, because we have that each constructor variable from the two constructors are equivalent. From this we can thus derive that their indices, which are built from these constructor variables, are equivalent.

Now, when we want to solve the unification problem Δ`injectivity` we gave earlier, we can use the injectivity rule on the 8th position in the telescope replacing equation e_1 and e_2 by an equation e_1 that states that n is equivalent to m, an equation e_2 that states that x is equivalent to y, and an equation e_3 that states that xs is equivalent to ys given e_1. However, in reality we get the following telescope:

```
Δinjectivity_1 : (X : Set) -> Telescope 9
Δinjectivity_1 X = n ∈ μ NatD tt , m ∈ μ NatD tt , x ∈ X , y ∈ X ,
  xs ∈ μ (VecD X) (n , tt) , ys ∈ μ (VecD X) (m , tt) ,
  e_1 ∈ n ≡ m ,
  e_2 ∈ proj_1 (subst (λ n' -> ⟦ Σ' X (λ x_2 -> ×' (n' , tt) (one' (suc' n' , tt)))
    ⟧c (μ (VecD X)) (suc_1 n , tt)) e_1 (x , xs , refl {x = (suc' n) , tt})) ≡ y ,
  e_3 ∈ proj_1 (subst (λ x -> ⟦ ×' (proj_1 (subst (λ x_1 -> ⟦ snd (VecD X x_1) ⟧c
    (μ (VecD X)) (suc_1 m , tt )) (sym (refl {x = f0})) (m , y , ys , refl
    {x = (suc' m) , tt})) , tt) (one' (suc' (proj_1 (subst (λ x_1 -> ⟦ snd (VecD X
    x_1) ⟧c (μ (VecD X)) (suc_1 m , tt )) (sym (refl {x = f1})) (m , y , ys
    , refl {x = (suc' m) , tt})))) , tt)) ⟧c (μ (VecD X)) (suc_1 n , tt )) e_2 (snd
    (subst (λ x -> ⟦ Σ' X (λ x_1 -> ×' (x , tt) (one' (suc' x , tt))) ⟧c (μ (VecD
    X)) (suc_1 n , tt)) e_1 (x , xs , refl {x = (suc' n) , tt})))) ≡ proj_1 (snd (snd
    (subst (λ x -> ⟦ snd (VecD X x) ⟧c (μ (VecD X)) (suc_1 m , tt)) (sym (refl {x =
    f1})) (m , y , ys , refl {x = (suc' m) , tt})))) , nil
```

Which can be explained by the `injectivityTelC` rule from chapter 7. Because the index n of a vector is located in a Σ' type, we replace the n in the remaining constructor element using e_1. But, because we project out the x in e_2, which is not dependent on n, we can derive that e_2 should be equivalent to $x \equiv y$. Hence, we use an additional unification rule `UReplaceElem` that states that if two types B_1, B_2 are equivalent, and there is an element of type B_1 located in the telescope, we can replace B_1 by B_2:

```
UReplaceElem : {B_1 B_2 : A -> Set}(p : Δ [ k ]:Σ[ A ] B_1)
  -> (f : (a : A) -> B_1 a ≡ B_2 a)
  -> Unification (replaceInTel B_1 B_2 Δ p f) -> Unification Δ
```

Using this rule, we replace e_2 and e_3 to get the following telescope:

```
Δinjectivity_2 : (X : Set) -> Telescope 9
Δinjectivity_2 X = n ∈ μ NatD tt , m ∈ μ NatD tt , x ∈ X , y ∈ X ,
  xs ∈ μ (VecD X) (n , tt) , ys ∈ μ (VecD X) (m , tt) , e_1 ∈ n ≡ m ,
  e_2 ∈ x ≡ y , e_3 ∈ subst (λ n -> μ (VecD X) (n , tt)) e_1 xs ≡ ys , nil
```

Which we can then solve by applying the solution rule 3 times:

```
Δinjectivity_3 : (X : Set) -> Telescope 3
Δinjectivity_3 X = n ∈ μ NatD tt , x ∈ X , xs ∈ μ (VecD X) (n , tt) , nil
```

A limitation of this version of the injectivity rule, is that it can only be used if there are corresponding equations that state the equivalence between the indices (e.g. in the example the equation between the two vectors needs to be dependent on the proof `suc'` $n =$ `suc'` m). If we want to solve an equation that is not dependent on such a proof, for example if we want to solve an equivalence between two vectors that have equivalent indices, we have to use higher-dimensional unification.

9.3 Higher-Dimensional Unification

Cockx [10] introduced the notion of higher-dimensional unification, that allows us to use the injectivity rule for equations where the indices of a data type are not fully general. An example of this is solving the telescope Δ`HOUnification` where we have an equation e that is an equivalence between two vectors of type `Vec` A (`suc` n):

```
ΔHOUnification : (X : Set) -> Telescope 6
ΔHOUnification X = n ∈ μ NatD tt , x ∈ X , y ∈ X , xs ∈ μ (VecD X) (n , tt) ,
  ys ∈ μ (VecD X) (n , tt) , e ∈ cons' X n x xs ≡ cons' X n y ys , nil
```

In this case, it is not possible to apply the injectivity rule directly, as the index n is not fully general, but it is possible to construct the equivalences $(e_2 : x \equiv y)$ and $(e_3 : xs \equiv ys)$. We start by applying the solution rule in reverse to generalize the index n. We define an additional unification rule `U<-Solution` that adds the equations $(e' : a \equiv a)$ and $(p : e' \equiv$ `refl`$)$ to the telescope at index k and replaces the element $(e : x \equiv y)$, where $(x, y : B\ a)$, at index k by $(e :$ `subst` $B\ e'\ x \equiv y)$:

```
U<-Solution : {A' : Set}(B : A' -> Set)(f : A -> A')
  -> {x y : (a : A) -> B (f a)} (p : Δ [ k ]:Σ[ A ] (λ a -> x a ≡ y a))
  -> Unification (doSolution<-Tel B f p) -> Unification Δ
```

If we apply this new unification rule on Δ`HOUnification`, where we take f as a function that maps n to (`suc'` n, tt) we get a new telescope Δ`HOUnification`$_1$ that contains $(e' :$ (`suc'` $n, tt) \equiv$ (`suc'` n, tt)) and $(p : e' \equiv$ `refl`$)$:

```
ΔHOUnification₁ : (X : Set) -> Telescope 8
ΔHOUnification₁ X = n ∈ μ NatD tt , x ∈ X , y ∈ X , xs ∈ μ (VecD X) (n , tt) ,
  ys ∈ μ (VecD X) (n , tt) , e' ∈ (suc' n , tt) ≡ (suc' n , tt) ,
  e ∈ subst (μ (VecD X)) e' (cons' X n x xs) ≡ cons' X n y ys ,
  p ∈ e' ≡ refl , nil
```

As the indices are now fully general, we can apply the injectivity rule on e' and e to get the telescope Δ`HOUnification`$_2$, which now contains an element e_1 that contains an equivalence between the vector of indices we added when applying the solution rule in reverse, the predicate p which is now updated to contain `suc` n, and the equations $(e_2 : x \equiv y)$ and $(e_3 :$ `subst` $(\mu\ (\texttt{VecD}\ X))\ e_1\ xs \equiv ys)$:

```
ΔHOUnification₂ : (X : Set) -> Telescope 9
ΔHOUnification₂ X = n ∈ μ NatD tt , x ∈ X , y ∈ X , xs ∈ μ (VecD X) (n , tt) ,
  ys ∈ μ (VecD X) (n , tt) , e₁ ∈ (n , tt) ≡ (n , tt) ,
  p  ∈ cong (λ ntt -> suc' (proj₁ ntt) , tt) e₁ ≡ cong (λ _ -> suc' n , tt) e₁ ,
  e₂ ∈ x ≡ y , e₃ ∈ subst (μ (VecD X)) e₁ xs ≡ ys , nil
```

Now, p is an equation between equality proofs, called a higher-dimensional equation, which we need to solve to remove e_1. What we can do is consider a one-dimensional version of this problem, which takes a natural number n and contains a natural number w and a proof p that (`suc'` w, tt) and (`suc'` n, tt) are equivalent (i.e. the functions in `cong` on both sides of the equivalence in p):

```
ΔHOUnification₂' : (X : Set)(n : μ NatD tt) -> Telescope 2
ΔHOUnification₂' X n = w ∈ μ NatD tt , p ∈ (suc' w , tt) ≡ (suc' n , tt) , nil
```

We can solve this problem by splitting p and removing $tt \equiv tt$ (using the injectivity rule for $\top$) and applying the injectivity rule on `suc'` $w \equiv$ `suc'` n. Then, using the solution rule we can remove w and end up with an empty telescope. To use this for the higher-dimensional problem, we can lift the previous equivalence to get a new equivalence where we replace e and p by the the result of the unification of Δ`HOUnification`$_2$', which results in the removal of e_1 and p:

```
ΔHOUnification₃ : (X : Set) -> Telescope 7
ΔHOUnification₃ X = n ∈ μ NatD tt , x ∈ X , y ∈ X , xs ∈ μ (VecD X) (n , tt) ,
  ys ∈ μ (VecD X) (n , tt) , e₂ ∈ x ≡ y , e₃ ∈ xs ≡ ys , nil
```

Then, we can simply use the solution rule on e_2 and e_3 to get a telescope containing n, x, and xs, thereby solving the unification problem:

```
ΔHOUnification₄ : (X : Set) -> Telescope 3
ΔHOUnification₄ X = n ∈ μ NatD tt , x ∈ X , xs ∈ μ (VecD X) (n , tt) , nil
```

This lifting function is specified in `HOUnification`, which takes a unification algorithm f between two elements a and b that may be dependent on elements in telescope Δ. If we have the elements u and v which are interpretations of telescope Δ, and we know that u applied to a and b is equivalent and v applied to a and b is equivalent, we can replace the equivalence $e : u \equiv v$ and $(p : \text{cong } (\lambda\ xs \to a\ (f'\ xs))\ e \equiv \text{cong } (\lambda\ xs \to b\ (f'\ xs))\ e)$ by an equivalence between the unification algorithm f applied to (u, r) and (v, s), for which the proof is explained in detail in the work by Cockx [10].

```
HOUnification : {X Y : Set}(a b : X -> Y)(f' : ⟦ Δ ⟧telD -> X)
  -> (f : Unification (mergeTel Δ (λ x -> (p ∈ a x ≡ b x , nil)) f'))
  -> (u v : ⟦ Δ ⟧telD) (r : a (f' u) ≡ b (f' u)) (s : a (f' v) ≡ b (f' v))
  -> Σ[ e ∈ u ≡ v ] (subst (λ ab -> proj₁ ab ≡ proj₂ ab) (×-create r s)
      (cong (λ xs -> a (f' xs)) e) ≡ cong (λ xs -> b (f' xs)) e)
  -> unify f (merge u (r , tt)) ≡ unify f (merge v (s , tt))
```

We can also create `HOUnification'` and `HOUnification'`$\circ$`HOUnification` following the proofs in the same work. To use this in our work, we add unification rules that allow for the addition of custom rules. For this, we define `UAddRule`$_1$ and `UAddRule`$_2$ that work on the data types `TelAt` and `TelAt'` from chapter 4, respectively. For `HOUnification` we use `UAddRule`$_2$, which contains a proof p that states that at index k we have an element of type A and at index $k+1$ we have an elements of type B. Then, if we have some resulting telescope `fTel`, a function f that maps an element (a, b) to an interpretation of `fTel`, a function f' that does the reverse, and a proof $f' \circ f$ that proves that applying f' to f results in the original input, we can replace the element at positions k and $k+1$ with `fTel`:

```
UAddRule₂ : {X : Set}{A : X -> Set}{B : (x : X)(a : A x) -> Set}
  -> (p : Δ [ k ]:Σ[ X ] (λ x -> A x) : (λ x a -> B x a))
  -> (fTel : Telescope m)
  -> (f : (x : X) -> Σ[ a ∈ A x ] (B x a) -> ⟦ fTel ⟧telD)
  -> (f' : (x : X) -> ⟦ fTel ⟧telD -> Σ[ a ∈ A x ] (B x a))
  -> (f'∘f : (x : X) -> (e : Σ[ a ∈ A x ] (B x a)) -> f' x (f x e) ≡ e)
  -> Unification (updateTel₂ p fTel f f' f'∘f)
  -> Unification Δ
```

Then we can use `HOUnification` in the place of f, where we use the unification algorithm from Δ`HOUnification`$_2$' and `refl` for r and s.

Chapter 10

Discussion

In chapter 9 we have shown that we can represent functions that perform pattern matching on indexed data types from our universe of generic data type descriptions as an interpretation of the generic case tree, and that we can use our generic evaluation function to evaluate such a case tree without the use of pattern matching on these data type descriptions. In this chapter, we show the strengths and limitations of this implementation.

10.1 Case Tree Strengths and Limitations

We have provided a generic representation of a case tree that performs case splits on data types from our universe of generic data type descriptions. For any function that can be represented by this case tree, we can evaluate to an element of the proper return type using the `eval` function. However, we can only represent functions that perform pattern matching on an element of a data type that can be represented by the universe of generic data type descriptions. This universe allows for indexed data types, but not other data types (e.g. higher-order inductive types or coinductive data types). One can extend the case tree representation by adding a branch that allows for other kinds of case splits without disrupting the current functionality, provided that the evaluation function is also updated to deal with such branches.

Furthermore, each case tree has to be written by hand following the representation given in chapter 9. This is a time consuming and error-prone ordeal. Especially when we count in the fact that the unification algorithm requires the user to provide a unification trace. Removing this requirement would require meta-programming, which goes against the goal of having a correct-by-construction implementation. The unification algorithm allows us to easily update or eliminate variables in the telescope of the branches. However, adding this might not always be a trivial endeavor. We have provided four rules that Agda itself uses when verifying a function defined by pattern matching to lessen this burden.

When defining a case tree, one should be wary not to call a function that uses pattern matching in a leaf node, as we do not have any checks that could detect such calls. For defining functions that contain recursive calls as a case tree we have provided a `Below` type that captures recursive calls to the evaluation function, but this should be manually added to the telescope of input arguments which is time consuming. Furthermore, the `Below` type allows us to define functions that use course-of-value iteration, but there are many other recursive schemes that are not possible, for example mutual recursion, which are also not included in the work by McBride [23] and Cockx [11], which we have based this work on.

10.2 Unification Strengths and Limitations

We have provided four unification rules: solution, deletion, injectivity, and conflict (which are also used by Agda's compiler), which all work for elements of data types from our universe of generic data type descriptions. However, Agda's compiler also uses the cycle rule. This rule is not commonly used, and therefore not implemented. However, functions that do require this rule can currently not be translated into a case tree. For example, if we want to prove a function $(n : \mu\ \texttt{NatD}\ \texttt{tt})(e : \mu\ (\equiv_D\ \ n)\ (\texttt{suc'}\ n, \texttt{tt})) \to \bot$, we would split on e, resulting in the relation $(e : n \equiv \texttt{suc'}\ n)$, for which we would need to use the cycle rule to show that it results in $\bot$. We do allow for the addition of other unification rules, as long as they can provide and prove its converse.

The deletion and injectivity rule in this implementation make use of the K-axiom. This axiom is not admissible in some type theories (e.g. homotopy type theory) and thus this implementation is not be admissible in these type theories. It might be possible to define a variant of the injectivity rule that does not rely on the K-axiom, which we will discuss some more in chapter 12. The remainder of the code does not rely on this K-rule.

When working with the provided unification rules there are two problems that one will encounter fairly quickly, which could both be solved by using telescopic equality. The first problem is encountered when extending the injectivity rule in chapter 9. Using the injectivity rule on a constructor that has more than one variable results in verbose types. For this, we provide a quick solution where one can replace equivalent types, but proving this equivalence requires a lot of work from the user. Instead, we could have used the concept of telescopic equality, where we only have one equivalence relation between two interpretations of the same telescope of indices. For example, instead of $(e_1 : \texttt{suc}\ n \equiv \texttt{zero})(e_2 : \texttt{subst}\ (\texttt{Vec}\ A)\ e_1\ (\texttt{cons}\ n\ x\ xs) \equiv \texttt{nil})$, we would have $(e : (\texttt{suc}\ n, \texttt{cons}\ n\ x\ xs) \equiv (\texttt{zero}, \texttt{nil}))$. Telescopic equality is hard to implement and would require us to rethink the current implementation of the unification rules, but it would make it easier to provide a unification trace.

Another problem that could be solved by using telescopic equality would be that the `UDeletion` and `USolution` have a very strict notion of two elements of the same data types being equivalent. Namely that in the telescope we need one element e_1 which has as type that the two telescope of indices are equivalent and an adjacent element that states that the elements of the data type are equivalent given this e_1. Therefore, if we have a telescope that contains several elements that make up e_1, we would first have to define a function that combine these elements to form e_1 before we could use the deletion or solution rule.

10.3 Evaluation Strengths and Limitations

By implementing an evaluation function that works for any interpretation of the generic case tree in compiled Agda, we have shown that we can always evaluate to an element of the return type of the case tree. However, only for the examples we have shown have we proven that the computational behaviour of the function defined by pattern matching and the evaluation on the case tree is equivalent. As one has to define the case tree by hand, we cannot make any claims about whether the semantics of the clause of the original function by dependent pattern matching is preserved. What we could do, however, is give dynamic semantics for the case tree, by defining the case tree in terms of an inductive relation. We can then prove that the evaluation function follows these semantics. This would ensure that the semantics are preserved, assuming that the translation from dependent pattern matching to the case tree is correct.

Furthermore, we can only manually check that the evaluation function does not use pattern matching. We have allowed pattern matching on metaconstructs (e.g. case trees and telescopes), but not on elements of the data types that we perform a case split on. We cannot prove that this last part holds for the implementation.

We can call the evaluation function recursively on the remainder of the case tree, because we have section-retraction pairs for all functions defined on telescopes, which means that for any function f that we have defined on an interpretation xs of an arbitrary telescope, we have a function f' such that $f'\ (f\ xs) \equiv xs$ This allows us to substitute, for example, the results of expanding and shrinking the telescope with the original telescope. However, we often have to solve functions that have nested section-retraction pairs. For example, to reduce a function in the form $g'\ (f'\ (f\ (g\ xs)))$, where f and g are section-retraction pairs, we would have to call the `subst` function with a predicate P that contains g', an equivalence relation e that contains a call to the proof that f and f' is a section-retraction pair (which itself requires the argument $g\ xs$). To solve the remaining equation, we again have to call the `subst` function to replace the section-retraction pair of g and g'. The more nested section-retraction pairs we get, the more verbose the code will become. This is particularly a problem when formalizing higher-dimensional unification. It would be better to have a generic way to reduce equations of these form to increase the readability of the proofs. This could be solved using meta-programming, which we did not allow in this thesis.

Chapter 11

Related Work

This thesis implements an evaluation function for a generic case tree in Agda. We discuss related efforts for proving and implementing translations from dependent pattern matching to eliminators and other elaborations.

11.1 Elaborating Dependent Pattern Matching

In `Epigram`, a compilation scheme for definitions by dependent pattern matching is given [30]. In Coq, this works has been elaborated for the `Equations` package [35] which translates a given function defined by dependent pattern matching to eliminators on a case-by-case basis. This translated function is then type checked by Coq's type checker. They use constructors like `NoConfusion` and `Below`, similar as to what we have presented. In this work however, we provide a generic evaluation function in pure Agda code, which gives the guarantee that we can evaluate any interpretation of the generic case tree to the output term. This evaluation function is implemented in terms of generic eliminators only, even though this can only be manually enforced.

`Lean` compiles definitions by dependent pattern matching into eliminators [17]. These compilation methods are based on the same ideas that are used in this thesis and supports structural and well-founded recursion that work both with and without K. What makes the work different from this thesis is that this compilation takes place outside of the trusted kernel of Lean. This trusted kernel is based on dependent type theory and its main task is type checking. This means that their compilation methods for definitions by dependent pattern matching are not type checked, whilst in this work the entire evaluation is written in correct Agda code.

The `Generics` library [20] derives generic properties from data types defined in Agda, such as the elimination principle and the no confusion property. It does so by translating data types in Agda to a generic universe of data type descriptions and defines the properties for these descriptions. In this work, we have defined these properties for the universe of data type descriptions only, thus not allowing the user to define data types the "Agda" way and skipping the translation.

The original version of dependent pattern matching by Coquand [14] that is implemented in this thesis is incompatible with homotopy type theory, due to the reliance on axiom K-rule (or equivalently the uniqueness of identity proof) which is not admissible in homotopy type theory [34]. On paper it is proven that the translation can be done without the use of K [11], by altering the unification algorithm and changing the definition of structural recursion. Here, we tried to avoid the use of K as much as possible (e.g. by introducing

higher-dimensional unification), but still require its use for the deletion and injectivity rule.

11.2 Unification

The `Equations` package allows to work without K. But instead of using higher-dimensional unification, they rely on a homogeneous no-confusion principle that works on any two terms in the same instance of the inductive family [36]. This solution is more limited as it cannot handle higher-dimensional equations, but the authors claim that it would be too general to compare constructors of different inductive families. Using indexed data types is quite rare in Coq, compared to Agda, so not having higher-dimensional unification is not as much of a limitation. In this work, we do implement higher-dimensional unification for inductive families.

11.3 Verifying Agda

Originally, everything in Agda's compiler was implemented in unverified Haskell [32]. In this work, we have verified that it is possible to evaluate a case tree without the use of dependent pattern matching for indexed data types, which is part of the type checking pass. Another pass in the compiler of Agda is scope checking, where references to names, such as variable use, are checked to ensure that the corresponding definition is in scope. Recent work has shown that it is possible to implement the scope checking pass in Agda itself [21], thereby proving the correctness of its name resolution algorithm.

Chapter 12

Conclusion

This thesis presents a generic, correct-by-construction implementation of an evaluation function of a case tree that performs case splits on indexed data types, without the use of metaprogramming and unsafe transformation. We provide a generic representation of a case tree that allows case splits only on variables of indexed data types. For each constructor of that data type we create a new case tree that, given a unification algorithm between the indices of the variable and the constructor, updates the variables in the telescope of input arguments.

We use a fixed universe of data type descriptions to represent the data types that a case tree can perform a case split on. This allows us to derive all the constructors that a variable can split into and their arguments, which are added to the telescope of input arguments. Using the provided `Below` type, we allow for functions that require course-of-value recursion to be evaluated without the use of pattern matching.

Using unification, we can derive whether a constructor is possible for a given indexed data type, thus allowing us to update the variables if a variable can be replaced by a constructor or eliminate the branch of a case tree when the variable cannot be replaced by that constructor. We provide an implementation for the solution, deletion, injectivity, and conflict rule, whilst also allowing users to add their own unification rules, provided that they provide and prove its converse. Furthermore, we have shown that higher-dimensional unification is compatible with this work.

We can evaluate an interpretation of the generic case tree, given an interpretation of the telescope of input arguments, by performing case analysis on a variable x that a case split is performed on. We recursively evaluate the remaining case tree, or equivalently the branch which corresponds to the constructor of x. We update the telescope of input arguments by performing unification on the original telescope of input arguments with the arguments that make up variable x. When a leaf node is reached we evaluate to the right-hand side of the corresponding clause.

12.1 Future Work

While the evaluation function is type-safe for any case tree of the given representation, there are still interesting ways to extend the scope and usability of this work.

12.1.1 Extending the Case Tree

Adding universe polymorphism to the data type descriptions, allows us to expand the range of the data types that can be defined in the fixed universe of data type descriptions. Cur-

rently, it is not possible to put an element of type `Set` in a constructor description, as each element should be of type `Set`. For example, defining a vector where we index on a set A is currently not possible. Adding universe polymorphism also requires us to update the definition of telescopes to allow for elements of type `Set`.

This thesis fixed on one definition of indexed data types that does not include higher-order induction. More involved forms of data types include inductive-recursive data types [18] and inductive-inductive data types [31]. An example of an inductive-inductive data type is a sorted list, where we simultaneously define the set `SortedList` and a predicate $\leqslant$ `L` that works on `SortedList` that determines whether the elements in the list are sorted. Currently, we do not have functionality to define mutual data types, but we could add seperate branches to the case tree that handle case splits on these data types. However, these data types would also require their own conflict and injectivity rules, as they currently only work for the given universe of data type descriptions.

We could also extend the case tree by adding record types [6], which allows us to perform copattern matching [1]. Record types are types that group values together. They have additional laws for equality of records (η-laws), which can be used to define additional unification rules. This would also require us to extend the case tree to allow a case split on a record type and extend the evaluation function.

It is possible that the unification algorithm for a certain case split requires the cycle rule. This rule requires that the data types have a property called acyclicity, which means that a term can never be structurally smaller than itself. Acyclicity holds for inductive data types, but is quite hard to establish. It would be interesting to see this acyclicity property defined for the universe of data type descriptions.

Furthermore, it would be interesting to see whether the `Generics` library [20], discussed in chapter 11, can be integrated in this work, thus allowing us to create case trees where we perform case splits on Agda data types. As this work uses a more involved universe of data type descriptions, we would have to extend the current definition of the case tree and thus also the evaluation function. But as both works use similar constructs on the universe of data type descriptions (e.g. `NoConfusion` and eliminators) it should be feasible.

12.1.2 From Pattern Matching to Case Trees

We have stated in chapter 10 that we cannot make any claims about the computational behaviour of the evaluation function, because we have no notion of a function that uses dependent pattern matching. This would be an interesting addition to this work, which would require a generic definition of dependent pattern matching and an elaboration function to the generic representation of the case tree. This elaboration is shown on paper [9] and is implemented in Agda's type checker.

Furthermore, it would be interesting to see if the defined case tree is compatable with the case tree in Agda's type checker. If this were to be true it would be possible to reduce Agda's trusted base. But because Agda allows for more data types than indexed data types, this would require quite a significant amount of effort, as we would also have to prove the elaboration holds for other data types that Agda allows for (e.g. record types). Similarly, we could look into the elaboration process that Lean uses when compiling definitions by dependent pattern matching, which uses the same constructs that we have used in this thesis, to make the elaboration process admissible in their trusted kernel.

Interpretations of the case tree and the unification trace both have to be defined by hand. It would be interesting to see whether this can be generated by an elaborator from a function defined by pattern matching. For this, we would require meta-programming to determine where in the telescope of function arguments we have a data type that we perform a case split on, and which unification rules we can apply on the remaining telescope of function arguments.

12.1.3 Without K

In chapter 10 we have shown that our implementation is not compatable with homotopy type theory, due to the reliance on axiom K in the unification rules. However, on paper it is proven that this elaboration is possible without the use of K [11]. It would be interested to extend this work to not use K. The most redundant reliance being the one in the injectivity rule, where we have to use the proof that the constructor number of both elements are always equivalent regardless of the variables that are established earlier on in the telescope. Cockx [11] proposes that we use a strict version of the injectivity rule, which is the one implemented in this thesis. Consequently, we need to use higher-dimensional unification to not exclude applying the injectivity rule on constructors like `refl`, which we have already shown is possible to implement without the use of K. Our reliance on the K rule is thus only in the proof that the constructor number of both elements are always equivalent regardless of the variables that are established earlier on in the telescope. As we use the `Fin` type to denote the constructor number, which has a unique identity proof, it follows from Hedberg's theorem [25] that we could prove this without the use of K. Therefore, this rule should be able to work without the use of K.

Another reliance is on the deletion rule, which relies directly on the K rule. Without K we should prohibit the use of this rule. Following from Hedberg's theorem, this should not be a problem if we only allow types that satisfy the uniqueness of identity proof. However, homotopy type theory is an example of a type theory where not all types have unique identity proofs, hence some functions cannot be translated to eliminators. In particular, functions that split on constructor forms that are not linear (i.e. variables can occur more than once) are not guaranteed to translate to eliminators.

Bibliography

[1] Andreas Abel et al. "Copatterns: programming infinite structures by observations". In: *ACM SIGPLAN Notices* 48.1 (2013), pp. 27–38.

[2] Agda. *Agda Documentation*. `https://agda.readthedocs.io/en/v2.6.4.3/`. 2024.

[3] Guillaume Allais et al. "A type and scope safe universe of syntaxes with binding: their semantics and proofs". In: *Proceedings of the ACM on Programming Languages* 2.ICFP (2018), pp. 1–30.

[4] Lennart Augustsson. "Compiling pattern matching". In: *Conference on Functional Programming Languages and Computer Architecture*. Springer. 1985, pp. 368–381.

[5] Marcin Benke, Peter Dybjer, and Patrik Jansson. "Universes for generic programs and proofs in dependent type theory". In: *Nord. J. Comput.* 10.4 (2003), pp. 265–289.

[6] Gustavo Betarte and Alvaro Tasistro. "Extension of Martin-Löf's type theory with record types and subtyping". In: *Twenty-Five Years of Constructive Type Theory* 36 (1998), pp. 21–39.

[7] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of functional programming* 23.5 (2013), pp. 552–593.

[8] James Chapman et al. "The gentle art of levitation". In: *ACM Sigplan Notices* 45.9 (2010), pp. 3–14.

[9] Jesper Cockx and Andreas Abel. "Elaborating dependent (co) pattern matching: No pattern left behind". In: *Journal of Functional Programming* 30 (2020), e2.

[10] Jesper Cockx and Dominique Devriese. "Lifting proof-relevant unification to higher dimensions". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. 2017, pp. 173–181.

[11] Jesper Cockx, Dominique Devriese, and Frank Piessens. "Eliminating dependent pattern matching without K". In: *Journal of functional programming* 26 (2016), e16.

[12] Jesper Cockx, Dominique Devriese, and Frank Piessens. "Unifiers as equivalences: Proof-relevant unification of dependently typed data". In: *Acm Sigplan Notices* 51.9 (2016), pp. 270–283.

[13] Projet Coq. "The Coq Proof Assistant-Reference Manual". In: *INRIA Rocquencourt and ENS Lyon, version* 5 (1996).

[14] Thierry Coquand. "Pattern matching with dependent types". In: *Informal proceedings of Logical Frameworks*. Vol. 92. Citeseer. 1992, pp. 66–79.

[15] Thierry Coquand and Christine Paulin. "Inductively defined types". In: *International Conference on Computer Logic*. Springer. 1988, pp. 50–66.

[16] Haskell B Curry. "Functionality in combinatory logic". In: *Proceedings of the National Academy of Sciences* 20.11 (1934), pp. 584–590.

[17] Leonardo De Moura et al. "The Lean theorem prover (system description)". In: *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*. Springer. 2015, pp. 378–388.

[18] Peter Dybjer. "A general formulation of simultaneous inductive-recursive definitions in type theory". In: *The journal of symbolic logic* 65.2 (2000), pp. 525–549.

[19] Peter Dybjer. "Inductive families". In: *Formal aspects of computing* 6 (1994), pp. 440–465.

[20] Lucas Escot and Jesper Cockx. "Practical generic programming over a universe of native datatypes". In: *Proceedings of the ACM on Programming Languages* 6.ICFP (2022), pp. 625–649.

[21] Francesco Gazzetta. "An Agda scope checker implemented in Agda". In: (2023).

[22] Eduarde Giménez. "Codifying guarded definitions with recursive schemes". In: *International Workshop on Types for Proofs and Programs*. Springer. 1994, pp. 39–59.

[23] Healfdene Goguen, Conor McBride, and James McKinna. "Eliminating dependent pattern matching". In: *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*. Springer, 2006, pp. 521–540.

[24] William A Howard. "The formulae-as-types notion of construction". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[25] Nicolai Kraus et al. "Generalizations of Hedberg's theorem". In: *Typed Lambda Calculi and Applications: 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings 11*. Springer. 2013, pp. 173–188.

[26] Per Martin-Löf. "An intuitionistic theory of types". In: *Twenty-five years of constructive type theory* 36 (1998), pp. 127–172.

[27] Per Martin-Löf. "Constructive mathematics and computer programming". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 104. Elsevier, 1982, pp. 153–175.

[28] Conor McBride. "Elimination with a motive". In: *International Workshop on Types for Proofs and Programs*. Springer. 2000, pp. 197–216.

[29] Conor McBride, Healfdene Goguen, and James McKinna. "A few constructions on constructors". In: *Types for Proofs and Programs: International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*. Springer. 2006, pp. 186–200.

[30] Conor McBride and James McKinna. "The view from the left". In: *Journal of functional programming* 14.1 (2004), pp. 69–111.

[31] Fredrik Nordvall Forsberg and Anton Setzer. "Inductive-inductive definitions". In: *Computer Science Logic: 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings 24*. Springer. 2010, pp. 454–468.

[32] Ulf Norell. *Towards a practical programming language based on dependent type theory*. Vol. 32. Chalmers University of Technology, 2007.

[33] Robert Pollack. "BRICS, [1] Computer Science Department, Aarhus University". In: *Twenty Five Years of Constructive Type Theory* 36 (1998), p. 205.

[34] The Univalent Foundations Program. "Homotopy type theory: Univalent foundations of mathematics". In: *arXiv preprint arXiv:1308.0729* (2013).

[35] Matthieu Sozeau. "Equations: A dependent pattern-matching compiler". In: *International Conference on Interactive Theorem Proving*. Springer. 2010, pp. 419–434.

[36] Matthieu Sozeau and Cyprien Mangin. “Equations reloaded: High-level dependently-typed functional programming and proving in Coq”. In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), pp. 1–29.

[37] Tarmo Uustalu and Varmo Vene. “Primitive (co) recursion and course-of-value (co) iteration, categorically”. In: *Informatica* 10.1 (1999), pp. 5–26.