Constructing Simple and Mutual Inductive Types

Stefania Damato

School of Computer Science

University of Nottingham

September 2020

Submitted in partial fulfillment of the conditions for the award of the degree MSc Computer Science.

I hereby declare that this dissertation is all my own work, except as indicated in the text.

Date : 10/09/2020

Signature: S. Damato

EU funds for Malta 2014-2020

MINISTRY FOR EDUCATION AND EMPLOYMENT

Abstract

Martin-Löf's dependent type theory is a formal language developed on the principles of constructive mathematics. It acts as the basis for modern proof assistants like Agda, which are tools for doing computer-assisted mathematics. This dissertation investigates the central notion of an inductive type within Martin-Löf type theory. We construct a small theory of signatures as a framework in which we can express simple or mutual inductive types. For a given signature, we then construct algebras, algebra morphisms, the initial algebra, and a unique morphism from the initial algebra to any other algebra. We thus obtain a complete specification of simple and mutual inductive types. Next, we focus on the W-type, an inductive type which encapsulates the recursive aspect of any inductive type. For a given signature, we construct an algebra for the indexed W-type's representation of the signature. We then present our attempt at constructing the iterator for this algebra. This provides a starting point for completing a reduction from simple and mutual inductive types to W-types, in order to show that a type theory supporting W-types can support all simple and mutual inductive types.

Keywords: Martin-Löf type theory; Agda; theory of signatures; inductive types; W-types; indexed W-types.

The research work disclosed in this publication is funded by the ENDEAVOUR Scholarships Scheme (Malta). The scholarship is part-financed by the European Union – European Social Fund (ESF) under Operational Programme II – Cohesion Policy 2014-2020, "Investing in human capital to create more opportunities and promote the well-being of society".

Operational Programme II
European Structural and Investment Funds 2014 - 2020
"Investing in human capital to create more opportunities and promote the wellbeing of society"
Scholarships part-financed by the European Social Fund
Co-financing rate: 80% EU Funds; 20% National Funds

Acknowledgments

I would like to thank Malta's ENDEAVOUR scholarships scheme for allowing me to pursue this Master's degree. I am fortunate and grateful to have had Prof. Thorsten Altenkirch supervise this project. His guidance, support, and sense of humour were a great help during this journey. Many thanks also go to my parents for their backing, as well as friends who offered encouragement and assistance, particularly Duncan, Luke and Dr Jean-Paul Ebejer.

Contents

1

Introduction

Our project investigates inductive types within the context of Per Martin-Löf's Type theory. This chapter presents the main problem this project seeks to address, the motivation for our study, an overview of the rest of the dissertation, and the contributions of our work.

1.1 Aim and Motivation

The main aim of this project is to address a gap in the formalisation of inductive types left incomplete by more general work—namely, a complete specification of simple and mutual inductive types, and an attempt at showing that W-types are enough to represent all simple and mutual inductive types. We build a framework in which we can specify any simple or mutual inductive type, and give a general construction for an inductive type in this framework. We then specify indexed W-types and start to construct a reduction from the previously defined inductive types to indexed W-types.

This project contributes towards the development of the metatheory of Martin-Löf Type Theory, by investigating its notion of an inductive type and reducing all inductive types to a single type. Type theory is the basis for type systems of programming languages and full-scale software verification. The study of type theory allows us to give better behavioural guarantees for our software, by making our requirements more explicit through the use of more expressive types. The constructions involved in these software verification systems can however become very large and complicated. Reducing this code base to a small as possible trusted core has two main advantages, the first being that we have to 'trust' as little code as possible that is responsible for our verification, and the second that we can avoid bugs which can be taken advantage of by malicious users. Moreover, Martin-Löf Type Theory was also constructed as an alternative (constructive) foundation of mathematics to set theory. The study of its metatheory is therefore essential if we are to seriously regard Martin-Löf Type Theory as a fully-fledged foundation for constructive mathematics.

1.2 Overview of the Dissertation

Chapter 2 gives an overview of the background material required for the rest of this dissertation. It presents an introduction to type theory and contrasts it with set theory, explains the propositions as types paradigm, and discusses Martin-Löf Type Theory constructs and inductive types. It then gives a brief introduction to Agda and why we use it in our project. The section ends with some category theory background we use in Chapter 5.

Chapter 3 contains a review of the related literature, and further contextualises and motivates our project.

In Chapter 4 we introduce our suite of examples and define a general framework in which to express inductive types.

Chapter 5 details our constructions of algebras, morphisms, the initial algebra, the iterator, and a proof of the latter's uniqueness for a general inductive type.

In Chapter 6 we cover our construction of the carriers and constructors for WI-types. We also describe our attempt at constructing the WI-type iterator, explain the challenges involved in its construction, and provide some ideas for ways to solve these challenges.

Chapter 7 concludes the dissertation with a summary and critical evaluation of our work, as well as future directions for the work.

1.3 Contributions

The main contributions of this dissertation are listed below.

- The construction of a small 'theory of signatures', a framework in which we can express any simple or mutual inductive type as a signature which holds all the necessary details about the type. This framework's syntax, described using an inductive type itself, was designed to be extendable, and we aim to expand it in future work to accommodate for more general types.

- A constructive derivation of the initial algebra of any given signature from the above framework, thereby providing a complete specification of simple and mutual inductive types, achieved by using induction on the syntax of the theory of signatures. More specifically, for a given signature `S`, we define the following.

  i. Algebras for `S`, consisting of carriers and their constructors.

  ii. Morphisms over algebras for `S`, which map the carriers of one algebra to the carriers of the other, and consist of a proof that these mappings are structure-preserving.

  iii. The initial algebra for `S`.

  iv. A morphism from the initial algebra for `S` to any other `S`-algebra, called the iterator.

  v. A proof that the iterator is unique, i.e. that any morphism from the initial algebra for `S` to any other `S`-algebra is equivalent to the iterator.

- The construction of the carriers and constructors for WI-types, resulting in the WI-type algebra for a given signature `S`, as well as a significant starting point for completing a reduction from W-types to simple and mutual inductive types, in order to show that a type theory supporting W-types can support all inductive types.

  Our original objective was the complete reduction from simple and mutual inductive types to WI-types. Due to time constraints and some unforeseen challenges in our work, this target was not achieved, however the only missing elements are the completion of the iterator for the WI-type algebra and a proof of its uniqueness, which we believe should only be slightly more difficult than the one we completed for the theory of signatures representation. This should therefore not take too long, and we leave it for future work. All of the constructions mentioned here were formalised in Agda and type checked in Agda version 2.6.0.1. This code was submitted as supplementary material.

2

Background

This chapter introduces the reader to the general area in which this dissertation sits, and presents the background material required to understand the rest of the report. Most of this material is adapted from The Univalent Foundations Program (2013), Prof. Thorsten Altenkirch's notes on type theory and category theory, and Bartosz Milewski's 'Category Theory for Programmers' (Milewski, 2018).

2.1 What is Type Theory?

A *type theory* or *type system* is a formal system in which every term is of some definite type. It sets out rules about the introduction and computation of types and their terms.

Type theories were originally motivated by questions related to the foundations of mathematics. Bertrand Russell discovered that some formalisations of naïve set theory lead to a paradox when considering the set of all sets that are not members of themselves. He tried to amend this with a 'tentative' theory of types (Russell, 1903). An early elegant formulation of a type theory called the simply typed lambda calculus was developed by Alonzo Church (Church, 1940). Later, William Alvin Howard wrote about the Curry–Howard correspondence (Howard, 1980), that is the direct relation between simply-typed lambda calculus and natural deduction, introducing the idea of 'propositions as types'. This was the starting point for Per Martin-Löf's Type Theory, which is the type theory we will be considering in this dissertation, and which we refer to hereafter simply as Type Theory.

2.2 Type Theory vs. Set Theory

Most mathematicians today accept and use set theory as the foundation of mathematics. Martin-Löf developed Type Theory, among other things, as a foundation for intuitionistic mathematics.[1] It is therefore instructive to explain the basic concepts of Type Theory in relation to how they are treated in set theory.

Set theory organises mathematical objects into collections called sets, such as the set of natural numbers $\mathbb{N} = \{1, 2, 3, \dots\}$. All mathematical objects can be represented as sets. For instance, the number 0 is simply a shorthand representation for the empty set $\{\}$, 1 is the set containing 0 $\{\{\}\}$, 2 is the set containing 0 and 1 $\{\{\}, \{\{\}\}\}$, and so on. All other mathematical objects such as relations, functions, lines, curves, and algebraic structures, can be defined in terms of sets.

Type Theory organises mathematical objects into types instead of sets, such as the type $\mathbb{N}$ of the natural numbers. Types collect objects of the same nature or structure together, and there are specific ways of forming new types from existing ones. There are four basic judgements in Type Theory, the first two of which are 'A is a type' and 'A and B are equal types'.

The third basic judgement in Type Theory that expresses that 'a mathematical object (or term) a is of type A' is written as $a : A$. For instance, to express that 3 is of type $\mathbb{N}$, we write $3 : \mathbb{N}$. We note that $a : A$ differs from $a \in A$ in that the former is a judgment whereas the latter is a proposition which can be true or false. We can think of $a \in A$ as a relation about two pre-existing objects a and A which may or may not hold, and $a : A$ as an atomic statement such that we cannot talk about a term a without specifying its type. In Type Theory, we can only construct terms of a certain pre-existing type, so that the type comes first and its terms come later.

The fourth basic judgement in Type Theory is the judgement $a \equiv_A b$ for the terms $a, b : A$, expressing that 'a and b are definitionally equal'. Another form of equality in Type Theory, which is a proposition instead of a judgement

[1]Intuitionism is a philosophy of mathematics introduced by L.E.J. Brouwer, which states that a mathematical object exists only if it can be constructed. In particular, it rejects the law of the excluded middle, which states that any proposition is either true or false: for all propositions P, we have $P \vee \neg P$.

and is equivalent to the equality used in set theory, is propositional equality, written $a =_A b$. When defining a function $f\colon \mathbb{N} \to \mathbb{N}$ by $f(x) = x + 5$, that $f(2)$ and 7 are equal is definitional—it is simply a matter of expanding out a definition. It would not make sense to reason about this equality as a proposition. On the other hand, that $x + 5$ is equal to $5 + x$ is a proposition that can be proved, and is hence a propositional equality. This distinction between equalities allows Type Theory to be agnostic about representations or encodings of mathematical objects, unlike in set theory.

The last difference between the theories we will mention here is that contrary to set theory, Type Theory is its own deductive system. Set theoretic foundations consist of two levels: the deductive system of first-order logic, together with the axioms of a particular theory, such as the Zermelo–Fraenkel axioms with choice (ZFC). Therefore, the proofs exist within first-order logic, which is a separate universe to that of the mathematical objects they talk about. However, Type Theory encodes both proofs and mathematical objects in a single language. Notions like negation, conjunction, and disjunction can be encoded as types themselves within the Type Theory.

2.3 Propositions as Types

How do we encode concepts from predicate logic, such as negation and conjunction, within Type Theory? In Type Theory, we do not think of truth, but rather of evidence. Instead of saying that a proposition[2] P holds or is true, we say that we have evidence for P. We associate P to a type and then construct an element of this type as a witness. Hence, the typing judgment $a : A$ can be read both as 'a is an element (or term) of type A', and as 'a is a witness (or proof) of the proposition A'.

This notion is known as the *Curry–Howard correspondence* or *propositions–as–types*. The basic idea is that derivations in natural deduction, or proofs, and terms in lambda calculus, or computations, are essentially equivalent. Proofs correspond to programs, and what the proof is proving is the type of the program. Hence types play the role of propositions, and terms of a type A are proofs of the proposition A. We combine this with the Brouwer-Heyting-

[2] By proposition, we mean a statement that potentially has a proof. A theorem is then a proposition that has been proven.

Kolmogorov (BHK) interpretation of logical operators in intuitionistic logic. This gives us that the meaning of a proposition A is a proof, or evidence, for A, and this proof can be expressed in terms of proofs of the sub-parts of A, if there are any (Troelstra, 1991). The way to provide evidence for a given proposition is then shown below for types A and B and property P.

Logic	**Type Theory**
$A \implies B$	$A \to B$
$A \wedge B$	$A \times B$
$A \vee B$	$A + B$
$A \iff B$	$(A \to B) \times (B \to A)$
True	$\mathbf{1}$
False	$\mathbf{0}$
$\neg A$	$A \to \mathbf{0}$
$\forall x : A.P(x)$	$\Pi_{x:A} P(x)$
$\exists x : A.P(x)$	$\Sigma_{x:A} P(x)$

Table 2.1: Logic connectives and their type theoretic equivalents.

The type theoretic connectives shown in Table 2.1 will be covered in more detail in Section 2.5. Evidence for $A \Rightarrow B$ is a function which transforms evidence for A to evidence for B. To provide evidence for $A \wedge B$ we provide a pair, with the first element being evidence for A and the second element being evidence for B. Evidence for $A \vee B$ consists of either evidence for A or evidence for B, i.e. the sum, or disjoint union, of A and B. $A \Leftrightarrow B$ follows similarly to $A \Rightarrow B$ but in both directions. True is represented by the type with one element $\mathbf{1}$, and false is represented by the empty type $\mathbf{0}$. Evidence for $\neg A$ is a function which takes evidence for A and inhabits the empty type, i.e. leads to a contradiction. For the translation of the quantifiers $\forall$ (for all) and $\exists$ (exists), we require dependent types, which we will discuss in the next section. To provide evidence for $\forall x : A.P(x)$, we use the dependent function type which assigns to any element x of A evidence for $P(x)$. To provide evidence for $\exists x : A.P(x)$, we use the dependent pair type which specifies a particular element x of A and provides evidence for $P(x)$.

2.4 Dependent Types

To properly understand the type of the evidence necessary for the quantifiers ∀ and ∃, we need to look into dependent types. Martin-Löf Type Theory differs from other type theories which also follow the Curry-Howard correspondence, by extending this correspondence to predicate logic using dependent types.

Generally speaking, as the name suggests, a *dependent type* is a type depending on other types. The `List` data type is such a type, being parameterised by the type `A`.

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

However, types like `List`, which depend on `Set` itself as opposed to a type of type `Set`, are not usually called dependent types because they fall under polymorphism, which is implemented in languages such as Haskell which do not support general dependent types. A better example would be the `Vec` data type, which is parameterised by the type `A` and indexed by ℕ.

```
data Vec (A : Set) : ℕ → Set where
    [] : Vec A 0
    _::_ : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

`Vec A n` is the type of vectors of length n having elements of type `A`. This type shows the full power of dependent types, as the type `Vec A n` depends on the value of one of its arguments, not just its type as we saw with `List`. We also note that `Vec A` actually defines a collection of types, as it encapsulates the the definitions of the types `Vec A zero`, `Vec A (suc zero)`, `Vec A (suc (suc zero))`, and so on, as opposed to a single type.

Dependent types allow us to express requirements for our code, which can help us ensure that our code is correct. For instance, when appending two vectors of length i and j, the resulting vector should have length $i + j$. This constraint can be expressed in the type of the append function for vectors.

```
_++v_ : {A : Set}{i j : ℕ} → Vec A i → Vec A j → Vec A (i + j)
[] ++v y = y
```

```
(x :: xs) ++v y = x :: (xs ++v y)
```

This type disallows certain mistakes, like setting `[] ++v w` to be `[]` for any `w` of length j, because the empty list `[]` has length 0 but the compiler is expecting a list of length $0 + j = j$. Contrast this to the append function for lists, where such a mistake would not be caught by the compiler because the type is not as expressive.

```
_++_ : {A : Set} → List A → List A → List A
[] ++ y = y
(x :: xs) ++ y = x :: (xs ++ y)
```

2.5 Martin-Löf Type Theory Constructs

Now that we have motivated the use of dependent types, we detail the relation between simple and dependent types, and revisit Section 2.3 to see how we can construct new types in Type Theory.

Universes

So far, we have been using the phrase 'A is a type', but have not been precise about what this means. A *universe* $\mathcal{U}$ is a type whose elements are types. Thus, for example, $\mathbb{N}$ is the type of natural numbers and its type is $\mathcal{U}$. A natural question arises: what is the type of $\mathcal{U}$? To avoid Russell's paradox, we say that there is a hierarchy of universes such that $\mathcal{U} = \mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \ldots$. Every universe $\mathcal{U}_i$ is an element of type $\mathcal{U}_{i+1}$, and $\mathcal{U}_i : \mathcal{U}_j$ for every $j > i$.

Thus, what we mean by 'A is a type' is 'A is an element of some universe $\mathcal{U}_i$'. We usually omit the subscript and unless stated otherwise, 'A is a type' will mean that 'A is an element of $\mathcal{U}$' with $\mathcal{U} = \mathcal{U}_0$.

Function Types (→)

The function type is a basic type former for simple types. Given types A and B, we can form the function type $A \to B$ which denotes the type of functions whose domain is A and whose codomain is B. Functions are constructed using λ-abstraction or pattern matching.

Product Types ($\times$)

The product type is another basic type former for simple types. Given types A and B, we can form the product type $A \times B$. Elements of this type are pairs (a, b) where $a : A$ and $b : B$, in fact this type is comparable to the Cartesian product of two sets in set theory.

Sum Types ($+$)

The sum type (or co-product type) is the last basic type former for simple types. Given types A and B, we can form the sum type $A + B$. Elements of this type are constructed either as $\text{inl}(a)$ where $a : A$, or as $\text{inl}(b)$ where $b : B$. The sum type is related to the disjoint union of two sets in set theory, although the sum type in Type Theory is agnostic to the representation of the types, whereas the disjoint union in set theory is not.

Π-types

A generalisation of the function type is the dependent function type, or Π-type, which is a basic type former for dependent types. Given a type A and a type $B \colon A \to \mathcal{U}$ indexed over A, we can form the type $\Pi_{x:A} B(x)$. This denotes a function whose codomain type depends on which element of the domain it is applied to. Elements of this type are of the form $(\lambda\, a \to b\,(a))$. Vectors are one such type that we have already seen, and indeed, given the type $\mathbb{N}$ and the type `Vec A` $: \mathbb{N} \to \mathcal{U}$, can be written in Π-type notation as $\Pi_{n:\mathbb{N}}((\text{Vec } A)\ n)$. The function type is a special case of the Π-type, namely when $B = \lambda\ \_ \to \mathcal{U}$, as in this case the type of the codomain of B does not depend on any elements of the domain.

Σ-Types

We also have a generalistion of the product type called the dependent pair type, or the Σ-type, acting as a basic type former for dependent types. Given a type A and a type $B \colon A \to \mathcal{U}$ indexed over A, we can form the type $\Sigma_{x:A} B(x)$. The Σ-type denotes a pair type where the second element's type depends on what the first element is. Elements of this type are of the form $(a\, , b\,(a))$. An example of such a type would be the type `FlexVec`. Given the type $\mathbb{N}$ and the type `Vec A` $: \mathbb{N} \to \mathcal{U}$, we form the dependent pair type $\Sigma_{n:\mathbb{N}}((\text{Vec } A)\ n)$ whose first element is a natural number n, and the second element is a vector of length n.

```
FlexVec : Set → Set
FlexVec A = Σ ℕ (Vec A)
```

The product type is a special case of the Σ-type, namely when $B = \lambda\_ \to \mathcal{U}$.

Finite Types

We now present some finite types that are widely used in Type Theory.

The empty type $\mathbf{0}$ is the uninhabited type. It has no constructors and therefore cannot have any elements. As we saw in Table 2.1, it is used to show a proposition's negation. We also always have a function from the empty type to any other type. This mirrors the logic rule of "false implies anything": $\forall q \in \mathbf{Bool}, \mathbf{false} \implies q$.

```
case⊥ : {A : Set} → ⊥ → A
case⊥ ()
```

The unit type $\mathbf{1}$ is the type with one element $\star : \mathbf{1}$. It can only be constructed in one way.

The last finite type we will mention here is the `Bool` type, or $\mathbf{2}$, with two constructors, `true` and `false`. The two element type can also be constructed using $\mathbf{1}$ and the sum type as $\mathbf{2} = \mathbf{1} + \mathbf{1}$. Its constructors would be $\text{inl}(\star)$ and $\text{inr}(\star)$. Other finite types like $\mathbf{3}$, $\mathbf{4}$, $\mathbf{5}$, …, can be constructed similarly as $\mathbf{3} = \mathbf{1} + \mathbf{2}$, and so on.

Lastly, we note that $+$ can be defined using `Bool` as $A + B = \Sigma_{b:\text{Bool}}(\lambda x \to \text{if } x \text{ then } A \text{ else } B)$, justifying the use of Σ, normally used to denote sums, for Σ-types. Similarly, $\times$ can also be defined using `Bool` as $A \times B = \Pi_{b:\text{Bool}}(\lambda x \to \text{if } x \text{ then } A \text{ else } B)$, justifying the use of Π, usually used for products, for Π-types.

2.6 Inductive Types

Another way we can define new types is by defining them inductively. Inductive types are the subject of this dissertation, and they are so central to Martin-Löf Type Theory that the latter has been described as "a theory of inductive definitions formulated in natural deduction" (Dybjer, 1994).

A *(simple) inductive type* X is one which can be defined by providing a list of constructors, each of which is a function (possibly having zero arguments) with codomain X, specifying how to form elements of this type. The simplest example is the set of natural numbers $\mathbb{N}$, whose constructors are `zero` and `suc`.

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

These are two of the so-called Peano axioms, which say that 0 is a natural number, and that if n is a natural number, then so is its successor `suc` n.

In Martin-Löf Type Theory (Martin-Löf and Sambin, 1984), the type $\mathbb{N}$, as well as every other inductive type, is specified by giving four rules:

- The *formation rule* tells us how to form a type from other types or families of types. (The formation rule for $\mathbb{N}$ simply states that $\mathbb{N}$ is a type. The formation rule for $A + B$ states that if A is a type and B is a type, then $A + B$ is also a type.)
- The *introduction rules* give meaning to the defined type by showing us how to form elements of the type. (The introduction rules correspond to a type's constructors.)
- The *elimination rule* tells us how to define functions on the type, and expresses a principle of proof by induction. (The elimination rule is described by a type's eliminator.)
- The *equality rules* relate the introduction and elimination rules, by telling us how functions defined on the type operate on the elements of the type i.e. it describes their computation.

A *mutual inductive type* is a collection of types defined simultaneously where each one refers to the other(s). An example is the mutual definition of the types `Even` and `Odd`.

```
data Even : Set
data Odd : Set
```

```
data Even where
    ezero : Even
    esuc : Odd → Even

data Odd where
    osuc : Even → Odd
```

We will only consider inductive type definitions to be valid if they are defined *strictly positively*. This roughly means that for an inductive type X, we allow X to occur in the input types of its constructors, but only to the right of arrows ($\rightarrow$) (The Univalent Foundations Program, 2013, p. 166). For example, we allow constructors like $c\colon (\mathbb{N} \rightarrow X) \rightarrow X$ for the type X, but not $d\colon (X \rightarrow \mathbb{N}) \rightarrow X$ or $e\colon ((X \rightarrow \mathbb{N}) \rightarrow \mathbb{N}) \rightarrow X$.

The *iterator* for an inductive type is a higher-order function to which we can reduce all recursion over the inductive type. It makes precise which functions can be defined on an inductive type. The ability to pattern match on a type in languages like Agda relies on the iterator, and any function defined by pattern matching can also be defined using the iterator directly. To express that an inductive type is defined uniquely in terms of its constructors and cannot be formed in any other way, we define a more general, dependently typed higher-order function, the *eliminator*. The eliminator can be derived from the iterator if we also have a proof that the iterator is unique. For $\mathbb{N}$, the eliminator (which corresponds to the principle of induction on $\mathbb{N}$) states that if a predicate holds for zero, and if whenever it holds for n it also holds for suc n, then it holds for all of $\mathbb{N}$. In other words, the eliminator expresses that we only need to consider the two constructors zero and suc to cover all possible ways to construct a natural number.

2.6.1 W-Types

W-types are also very central to our dissertation. W-types are of interest to us because any inductive type can be written as a W-type, so by assuming the existence of the W-type we obtain all other inductive types.

Drawing from the concept of well-orderings and the principle of transfinite induction introduced by Cantor, Martin-Löf (Martin-Löf, 1982) defined in his Type Theory a type former for well-orderings, now known as W-types. A

W-type $W_{x:\,A}B(x)$ is formed by providing a type $A\colon \mathcal{U}$ and a type $B\colon A \to \mathcal{U}$ indexed by A. To construct elements of $W_{x:\,A}B(x)$, we use the constructor

$$\mathrm{sup}\colon (a : A) \to (B(a) \to W_{x:\,A}B(x)) \to W_{x:\,A}B(x).$$

We can think of W-types as labelled trees, with A being the set of node labels and $B(a)$ for $a\colon A$ being the set of edge labels from the node labelled a. Hence a tree is described by a choice of an $a : A$ and a function b assigning to each edge $e : B(a)$ a child tree.

A variant of W-types to which we can reduce all inductive families are indexed W-types. Given $I\colon \mathcal{U}$, $S\colon I \to \mathcal{U}$, and $P\colon (i\colon I) \to S\ i \to I \to \mathcal{U}$, the *indexed W-type* or *WI-type* $WI\colon I \to \mathcal{U}$ has elements constructed by

$$\mathrm{sup}\colon (i\colon I)(s\colon S\ i)(f\colon (j\colon I) \to P\ i\ s\ j \to WI\ j) \to WI\ i.$$

We give examples of and construct WI-types in chapter 6.

2.7 Agda

All the formal development in this project was carried out in Agda. Agda is the ideal vehicle for our study, being a dependently-typed programming language and proof assistant, implementing a version of intensional Martin-Löf Type Theory. Agda disallows non-terminating programs by virtue of being total, and writing an Agda program involves refining a well-typed partial expression to obtain a well-typed total expression (Dybjer, 2018). Agda code is in fact rarely run but type checked instead. Being based on the propositions as types paradigm, if we have a proposition represented as a type, and an element of this type represented as a program, and this program type checks, this constitutes a proof that the proposition holds. Thanks to these guarantees, the way to evaluate our results is to ensure that our Agda code type checks at the relevant types.

This section briefly introduces the basic Agda constructs we will use throughout the dissertation. For a more comprehensive introduction, we refer the reader to Norell and Chapman (2009), Wadler et al. (2020), or The Agda Team (2020).

In Agda, we define data types as follows.

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

We have already seen this and other definitions of types, and now provide some more detail into the syntax. ℕ is the name of our data type, and its type is `Set`. `Set` is the first universe in Agda's hierarchy of universes `Set` = $\texttt{Set}_0$: $\texttt{Set}_1$: $\texttt{Set}_2$: ..., which is an implementation of what we discussed in Section 2.5. ℕ has the two constructors `zero` and `suc`. We represent the empty type **0** as

```
data ⊥ : Set where
```

and the type **1** with one constructor as

```
data ⊤ : Set where
  tt : ⊤.
```

Another important data type we will come across is `Fin`, the type of finite sets, where `Fin n = {0, 1, . . . , n-1}`.

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc : {n : ℕ} → Fin n → Fin (suc n)
```

Implicit arguments that are required for the definition of a type but can be automatically inferred by Agda are placed inside curly braces, like the `{n : ℕ}` above.

Functions can be defined using λ-abstraction, or else by pattern matching, like so. (The underscores in the function name are placeholders for the arguments. This type of notation is called mixfix notation.)

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)
```

Another way we can define data types in Agda is using records, which generalise Σ-types and allow us to store named fields in the type. One example of a record type is the implementation of the product type (discussed in Section 2.5).

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B
```

The implementation of the sum type (discussed in Section 2.5) is shown below.

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

We use Agda's equality type ≡ having a unique constructor `refl`. This relation has various properties, such as `symmetry`, `transitivity`, `congruence`, and `substitutivity`, which we will utilise in our equality proofs. This equality type is in contrast with the one used in Agda's Cubical mode, which we do not use here but discuss in later sections.

```
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  instance refl : x ≡ x
```

When using Emacs mode, we can type check a document which is not yet complete. We do this by writing `?` in the place of an expression. Agda will then replace `?` by a hole, an example of which is shown below, which we can fill later with the correctly typed code.

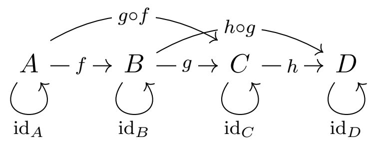
```
_+_ : ℕ → ℕ → ℕ
m + n = {!!}
```

2.8 Category Theoretic Semantics of Inductive Types

This final section gives a brief overview of the mathematical background that informs our constructions in Chapter 5.

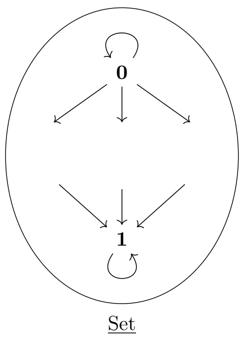
A category is a straightforward concept. A *category* $\underline{\mathbf{C}}$ consists of *objects* and *arrows* between the objects. Crucially, we need arrows that *compose*, i.e. given objects A, B, and C, and arrows $f\colon A \to B$ and $g\colon B \to C$, then there must exist the arrow $g \circ f\colon A \to C$. The other requirements are that composition of arrows is associative, i.e. given $f\colon A \to B$, $g\colon B \to C$, and

$h\colon C \to D$, we have $h \circ (g \circ f) = (h \circ g) \circ f$, and that for every object A, we always have an identity arrow $\mathrm{id}_A\colon A \to A$ such that for any $f\colon A \to B$, $f \circ \mathrm{id}_A = \mathrm{id}_B \circ f = f$. Arrows are also called morphisms or maps.

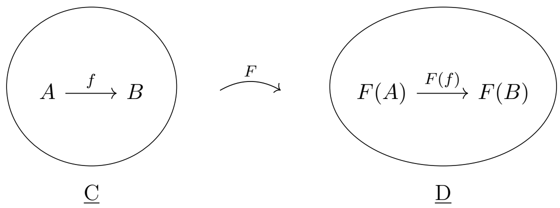


The main category we will be considering is the category of sets **Set**. The type of objects in this category is **Set**, and the set of morphisms is the set of functions, **Set**$(A, B) = A \to B$ for objects A and B. Composition of morphisms is defined as function composition, and the identity morphism for an object A is simply the identity function $\mathrm{id}_A\colon A \to A$, defined by $\mathrm{id}_A\, a = a$.

An *initial object* $\mathbf{0}$ is an object such that there exists exactly one morphism from this object to any other object. In **Set**, this would be the empty set, for which this morphism exists trivially since there are no elements in the empty set (the morphism is precisely `case⊥`). Similarly, a *terminal object* $\mathbf{1}$ is an object such that there is exactly one morphism from any other object to it. In **Set**, this would be the unit set. The morphism from an object type A would map all the elements of A to the unique element in the unit set.



Given categories **C** and **D**, a *functor* $F\colon \mathbf{C} \to \mathbf{D}$ associates to each object A in **C** an object $F(A)$ in **D**, and associates to each morphism $f\colon A \to B$ in **C** a morphism $F(f)\colon F(A) \to F(B)$ in **D**. The latter must preserve the identity morphism and composition of morphisms. Note how F is overloaded to mean both a map on objects and a map on morphisms. Functors are simply mappings between categories. An *endofunctor* is a functor that maps a category to that same category.

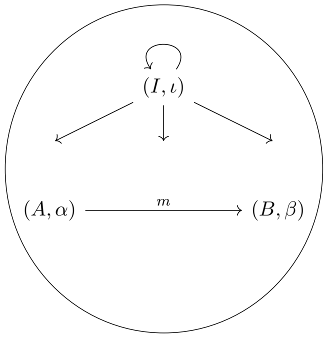


Given a category $\underline{\mathbf{C}}$ and an endofunctor $F\colon \underline{\mathbf{C}} \to \underline{\mathbf{C}}$, an F-*algebra* is a tuple (A, α) where A is an object in $\underline{\mathbf{C}}$ and is called the carrier of the algebra, and $\alpha\colon F(A) \to A$ is a morphism in $\underline{\mathbf{C}}$.

Algebras over a given endofunctor $F\colon \underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$ form a category in their own right. The objects are the tuples (A, α) as described above, and a morphism m between two objects (A, α) and (B, β) maps A to B while preserving the structure of α and β. We define such a morphism by noting that since F is an endofunctor and $m\colon A \to B$ is a morphism in $\underline{\mathbf{Set}}$, we can apply F to m and get $F\ m\colon F\ A \to F\ B$. Now since $\alpha\colon F(A) \to A$ and $\beta\colon F(B) \to B$, we can follow $F\ m$ with β to get $\beta \circ (F\ m)$, or equivalently, use α followed by m to get $m \circ \alpha$. Hence a morphism between objects (A, α) and (B, β) is defined as a map $m\colon A \to B$ such that $\beta \circ (F\ m) = m \circ \alpha$.

$$\begin{array}{ccc} F(A) & \xrightarrow{F(m)} & F(B) \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ A & \xrightarrow[m]{} & B \end{array}$$

An initial object in this category of F-algebras, if it exists, is called the *initial algebra*. Let us assume that the initial algebra exists, and let us call it (I, ι), with $I : \mathbf{Set}$ and $\iota\colon F(I) \to I$. Then Lambek's lemma states that ι is a special type of morphism, an isomorphism. This essentially means that $F(I)$ and I are structurally equal, and it makes I a fixed point of the endofunctor F. Being initial, I is also the *smallest* fixed point of F, in that there is a map from I to any other fixed point (because there is a map from I to any other algebra).



Category of F-algebras

Now consider the simple inductive type ℕ.

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

This type is entirely described by its two constructors `zero` and `suc`. The first picks the zero element while the second maps a number to its successor. We can write both constructors as functions by rewriting `zero` as a function from the terminal object **1** to ℕ, obtaining two functions which completely specify the type ℕ.

$$\mathsf{z} : \mathbf{1} \to \mathbb{N}$$
$$\mathsf{s} : \mathbb{N} \to \mathbb{N}$$

These functions can be rewritten using the exponential notation

$$\mathsf{z} : \mathbb{N}^{\mathbf{1}}$$
$$\mathsf{s} : \mathbb{N}^{\mathbb{N}}$$

so that we can now describe this pair of functions by a single function using the product type (and some simple algebra that holds in any Cartesian closed category).

$$\mathsf{z} \times \mathsf{s} : \mathbb{N}^{\mathbf{1}} \times \mathbb{N}^{\mathbb{N}}$$
$$\mathsf{z} \times \mathsf{s} : \mathbb{N}^{\mathbf{1}+\mathbb{N}}$$
$$\mathsf{z} \times \mathsf{s} : \mathbf{1} + \mathbb{N} \to \mathbb{N}$$

The sum of powers of $\mathbb{N}$ on the left hand side of the resultant function defines an endofunctor on $\underline{\mathbf{Set}}$. This endofunctor, which we will call F, takes an object X in $\underline{\mathbf{Set}}$ and maps it to the sum type $\mathbf{1} + X$.

$$F\colon \underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$$
$$F(X) = \mathbf{1} + X$$

Given the category $\underline{\mathbf{Set}}$ and the endofunctor F, we obtain F-algebras of the form (A, α) with A : **Set** and $\alpha\colon \mathbf{1} + A \to A$, and hence we can form the category of F-algebras. One such F-algebra could have carrier **Bool** and constructors `true` : **Bool** and `not` : **Bool** $\to$ **Bool** (with `not` defined as the usual boolean negation). The initial algebra of this category exists and is precisely the inductive type $\mathbb{N}$, where the carrier is $\mathbb{N}$: **Set**, and the morphism $F(\mathbb{N}) \to \mathbb{N} = \mathbf{1} + \mathbb{N} \to \mathbb{N}$ is comprised of the two constructors `zero` : $\mathbb{N}$ and `suc` : $\mathbb{N} \to \mathbb{N}$. $\mathbb{N}$ is the fixed point of the endofunctor $F(X) = \mathbf{1} + X$, so that $F(\mathbb{N}) \simeq \mathbb{N}$. In Chapter 5, we construct the initial algebra for any simple or mutual inductive type, constructively proving that every such inductive type has an initial algebra.

3

Literature Review

This chapter aims to further introduce the context of our project. We analyse and synthesise work done in relation to our research, as well as locate our project within the literature.

A type theory is a formal system that serves as the basis for automated type systems of modern programming languages, as well as the foundation of proof assistants like Coq and Agda, which are tools used for computer-assisted mathematical proofs and software verification. Per Martin-Löf's Type Theory, the type theory in which our project takes place, is a foundational language for programming and constructive mathematics, based on the Brouwer-Heyting-Kolmogorov interpretation of logic.

After learning of the analogy between formulae and types from his colleague W. A. Howard, and subsequently of the equivalence of natural deduction and lambda calculus, Martin-Löf began working on normalisation results for systems of natural deduction (Dybjer et al., 2012). His initial attempt at an intuitionistic theory of types (Martin-Löf, 1971) was found to contain a paradox by J. Y. Girard. He fixed this paradox and laid out the foundations of what we now refer to as Martin-Löf Type Theory (or intuitionistic type theory) in Martin-Löf (1972), and introduced different versions of it in following years (Martin-Löf, 1982; Martin-Löf and Sambin, 1984).

Martin-Löf's formulations of Type Theory include inductive definitions of, among others, the set of natural numbers $\mathbb{N}$, finite sets $\texttt{Fin}(n)$, and the disjoint union of two types $A + B$. Inductive definitions are fundamental to modern type theory and functional programming languages, and they have been studied and generalised to obtain a more expressive notion of types. A survey of inductive types within intensional and extensional type theories, as well as the more recent development of homotopy type theory, is available in Awodey et al. (2012). Our research helps to gain a better, more complete understanding of inductive types.

One of the established results in this area is due to Dybjer (Dybjer, 1997). He notes that in Martin-Löf and Sambin (1984), Martin-Löf demonstrates how to encode the natural numbers and the ordinal numbers using W-types. Dybjer generalises this idea and proposes a general criterion for using W-types to define any inductive type, by referring to their category theoretic semantics. He shows that for any strictly positive functor Φ on the category **Set** (built using only constants, variables, $+$, $\times$, and $\rightarrow$), there exists a set A and a family of sets B indexed by A such that for all sets X, $\Phi(X) \simeq \Sigma_{x:\,A}(B(x) \rightarrow X)$. The right hand side being isomorphic to $W_{x:\,A}B(x)$ justifies the fact that any inductive type represented by a strictly positive endofunctor can be represented as a W-type. As a corollary to this result, he obtains that every strictly positive endofunctor has an initial algebra. Despite showing the relation between inductive types and W-types, this paper does not contain a clear specification of inductive types, nor does it cover nested inductive types, such as the type for rose-trees, in its treatment of simple inductive types.

Our research is largely inspired by Kaposi et al. (2019). In this paper, the authors use a quotient inductive-inductive type (QIIT) to define a small type theory, which they call the theory of signatures, and show that if a given type theory supports this QIIT, then it supports all finitary QIITs. They achieve this by using induction on the theory of signatures to define algebras, morphisms, the initial algebra, the recursor, and some other constructions. QIITs generalise simple inductive types, but in particular, they admit not only point constructors (like the ones we have seen so far in the report), but also equality constructors (which specify how to equate two elements). These constructors and their homotopical interpretations are studied in homotopy type theory, but we will not be looking into them in this project as they are more general than the types we tackle. The focus of our project is an analogous result to the one in this paper, but for less general types: if a type theory supports W-types, then it can express all inductive types; and if a type theory supports indexed W-types, it can express all indexed inductive types. These analogous results have in fact been mentioned in the paper, but have not been explicitly formalised and shown. Our research will fill in this gap in the formalisation of inductive types.

Our approach will involve following a similar sequence of steps taken in Kaposi et al. (2019), by first defining a small theory of signatures, and then using induction on its structure for our constructions. Our project builds on established results regarding inductive types, like the result in Dybjer (1997), while combining them with the new development of a theory of signatures in the style of the above paper. Dybjer (1997) motivates our project and provides the mathematical theory behind what we aim to formalise algorithmically. More specifically, given a strictly positive endofunctor, we construct its initial algebra by defining algebras for the endofunctor, constructing the initial algebra's carriers and constructors, constructing a morphism from this algebra to any other algebra, and showing that this morphism is unique.

A complete description of strictly positive inductive types is given in Abbott et al. (2003) and extended in Abbott et al. (2005). The authors define a *container* by a type of 'shapes' S and a family of 'position types' indexed by S, $P\,s$, and write the container as $(s\colon S \triangleright P\,s)$ or simply $(S \triangleright P)$. For example, the type $\mathsf{List}(X)$ of lists with elements of type X can be represented by the length of the list $n\colon \mathbb{N}$ along with some function $\sigma\colon \mathsf{Fin}(n) \to X$, where $\mathsf{Fin}(n) = \{0, 1, \ldots, n-1\}$, which gives the element present in the list at any given position. Hence we can express $\mathsf{List}(X)$ as the container $(n\colon \mathbb{N} \triangleright \mathsf{Fin}(n))$. Their main result states that any strictly positive inductive type can be interpreted as a container, so that containers can be regarded as a normal form for these types. A container gives rise to a container functor $T_{S \triangleright P}(X) = \Sigma_{s\colon S}(P\,s \to X)$ denoted by $[\![S \triangleright P]\!]$. As shown by Abbott (2003), the initial algebra of a container functor $[\![S \triangleright P]\!]$ is given by the W-type $W_{x\colon S}P(x)$. Abbott et al. (2005) generalise on the result of Dybjer (1997) by considering coinductive types and nested occurrences of inductive and coinductive types, neither of which were studied in Dybjer (1997), as well as by analysing the categorical infrastructure of Martin-Löf categories required for their proofs. These papers offer a more detailed and comprehensive mathematical definition of inductive types and W-types, and these insights will be employed in our definitions of signatures and W-types.

The more general indexed inductive types are explored in Altenkirch and Morris (2009). The semantics of inductive types as initial algebras for polynomial endofunctors can be extended to indexed inductive types (or inductive families). Instead of considering functors on the category of sets, the authors consider functors on the category of indexed families, i.e. families indexed by a given type. Similar results from Abbott et al. (2005) carry over from inductive types to inductive families. The main result is that any indexed inductive type can be represented as an indexed W-type, which is an indexed version of the W-types studied in Abbott et al. (2005). While we do not consider indexed inductive types in our project, we do model indexed W-types, which are studied extensively in this paper and which inform our definition of WI-types. Perhaps surprisingly, W-types are still enough to represent the more general indexed inductive types. This is due to a reduction from indexed W-types to W-types, which was presented in Altenkirch and Morris (2009) and formalised further in Altenkirch et al. (2015).

4

The Theory of Signatures

Our first task was to construct a framework in which we could express any simple or mutual inductive type. We begin this section by looking at a few different inductive types to familiarise ourselves with what we are trying to formalise. The rest of the section details our definition of a small theory of signatures in which we can express these types.

4.1 Suite of Examples

The following inductive types will be used as running examples throughout the rest of the dissertation.

ℕ – The first type is the familiar type of natural numbers à la Peano. This is one of the simplest and most well-known inductive types.

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

`Lam` – Next is the 'naive' type of λ-terms. We have constructors for variables, abstraction, and application. It is 'naive' in the sense that there are more elaborate ways to define λ-terms that would be a more correct representation, however this will suit our purposes.

```
data Lam : Set where
    var : String → Lam
    abs : String → Lam → Lam
    app : Lam → Lam → Lam
```

`InfTree` – Our third inductive type is the type of infinitely branching trees. Such a tree is either empty, or else each of its nodes branches with a countably infinite factor. This type is different to the other types presented here due to its second constructor having a function as an argument.

```
data InfTree : Set where
    ε∞ : InfTree
    sp∞ : (ℕ → InfTree) → InfTree
```

`NF-NE` – Next we have our first mutual inductive type. It is the type of β-normal forms and neutral λ-terms. To define mutual inductive types in Agda, we first define all the types and then define their constructors. Note how the constructors for `NF` use `NE` and vice versa.

```
data NF : Set
data NE : Set

data NF where
   ne : NE → NF
   lam : String → NF → NF

data NE where
   var : String → NE
   app : NE → NF → NE
```

`Tree-Forest` – The last inductive type in our suite of examples is the tree-forest type emulating the rose-tree nested inductive type. This is another mutual inductive type, however this time it is also parameterised by a type A (A is of type `U` instead of `Set` for reasons detailed later).

```
data Tree (A : U) : Set
data Forest (A : U) : Set

data Tree A where
    sp : Forest A → Tree A

data Forest A where
    εF : Forest A
    consF : Tree A → Forest A → Forest A
```

The rose-tree nested inductive type is shown below. We emulate having a

List argument using the Forest type, and the sp constructor in Tree is the same as node in RoseTree.

```
data RoseTree : Set where
    node : List (RoseTree) → RoseTree
```

We do not consider nested inductive types in our dissertation, but we aim to do so in future work. Nested inductive types can also be 'indirectly' represented using mutual inductive types, like in the case of rose-trees.

4.2 Signatures

Now that we have looked at a few examples, we can start discussing how to give a general framework for describing inductive types. In category theory terms, what we specify here is the endofunctor corresponding to the constructors of a particular inductive type over the category **Set**.

Firstly, each inductive definition has a number of mutual types which we call sorts. ℕ has one sort while NF-NE has two. Each of these sorts then has a number of constructors related to it. The signature Sig corresponding to an inductive definition is thus defined as the number of sorts, and a function associating each sort to a list of constructors.

```
record Sig : Set where
    field
        sorts : ℕ
        cns : Fin sorts → List (Con sorts)
```

A constructor for a fixed sort is given by a list of arguments. We omit the resultant type of the constructor from this list, as this type will always be the sort we are defining.

```
data Con (n : ℕ) : Set where
    cn : List (Arg n) → Con n
```

The arguments of a constructor can refer to the sort/s we are defining. If they do not, like String in the constructor abs in Lam, they are non-recursive arguments. If they do, like ℕ in the constructor suc of ℕ, or (ℕ → InfTree) in sp∞ of InfTree, or NF in app of NE, they are recursive arguments. Non-recursive arguments are described simply by their type. For recursive arguments, their specification depends on whether they are function arguments or not. We associate a list of types to recursive arguments, where for function arguments like (ℕ → `InfTree`) it contains the types of the arguments of this function, in this case (ℕ :: []), whereas the list is empty for recursive arguments that are not functions. Lastly, every recursive argument must also be associated with the sort it is referring to.

```
data Arg (n : ℕ) : Set where
    nrec : U → Arg n
    rec : List U → Fin n → Arg n
```

We make use of `U` instead of `Set` to describe the type of arguments of constructors. Had we used `Set` in the place of `U` in `Arg` above, `Arg` would have type $\texttt{Set}_1$ to accomodate for its constructors having arguments of type `Set`. $\texttt{Set}_1$ is bigger than `Set` and would not fit into some types we define later. To avoid this, we use `U` instead, which mimics a universe. Elements of `U` represent (while not actually being) other types, and the function `El` associates this representation to the actual type. The only types we include in `U` are the ones we use in our examples, i.e. ℕ and `String`, but `U` can easily be extended to accomodate for more types.

```
data U : Set where
    nat : U
    string : U

El : U → Set
El nat = ℕ
El string = String
```

4.3 Example Signatures

The signature for ℕ is given below.

```
ℕSig : Sig
sorts ℕSig = 1
cns ℕSig = λ {zero → cn [] -- zero
                   :: cn (rec [] zero :: []) -- suc
                   :: []}
```

ℕ has one sort which can be constructed using either `zero` or `suc`. `zero` has no arguments, hence its list of arguments is empty, and it is represented as `cn []`. `suc` has one recursive argument and is represented as `cn (rec [] zero :: [])`. The `[]` in `rec [] zero` reflects that the recursive argument is not a function, and the `zero` refers to the sort the recursive argument corresponds to (in this case, the only sort).

The signature for `InfTree` is given below.

```
InfTreeSig : Sig
sorts InfTreeSig = 1
cns InfTreeSig = λ {zero → cn [] -- ε∞
                        :: cn (rec (nat :: []) zero :: []) -- sp∞
                        :: []}
```

This signature is similar to ℕ's signature, except this time, the constructor `sp∞` has a recursive function argument represented as `rec (nat :: []) zero`.

The signature for `NF-NE` is given below.

```
consNF : Fin 2 → List (Con 2)
-- NF constructors
consNF zero = cn (rec [] (suc zero) :: []) -- ne
            :: cn (nrec string :: rec [] zero :: []) -- lam
            :: []
-- NE constructors
consNF (suc zero) = cn (nrec string :: []) -- var
                  :: cn (rec [] (suc zero) :: rec [] zero :: []) -- app
                  :: []
NFNESig : Sig
sorts NFNESig = 2
cns NFNESig = consNF
```

This inductive type contains two sorts, with `NF` being the 0^{th} sort and `NE` being the 1^{st}. The function `consNF` assigns constructors to each sort. The constructors have both recursive and non-recursive arguments. Note how we represent constructors with recursive arguments, like `ne : NE → NF` represented as `cn (rec [] (suc zero) :: [])`. The `(suc zero)` refers to this recursive argument being of type `NE`, the 1^{st} sort.

5

Constructions on Signatures

`Sig` provides a way to encode any simple or mutual inductive type. Throughout this chapter, we will use `Sig`'s structure to define various constructions. In particular, we define a structure for *algebras* for a given signature, as well as *morphisms* between these algebras. We then construct the *initial algebra*, and a morphism from the initial algebra to any other algebra of the signature, called the *iterator*. Finally, we prove that the iterator is unique.

5.1 Algebras

As we saw in Section 2.8, an algebra over a given endofunctor F consists of a carrier A and a map $\alpha\colon F(A) \to A$. Translating this into Agda code, endofunctors are represented by signatures as defined in the previous chapter, and α is represented by constructors that can be written as maps and combined together using the product type. Hence, an algebra over a given signature consists of a carrier type and constructors forming this type.

As an example, an algebra for ℕ can be described as a record type consisting of a type `N` and functions of type `N` and `N → N`.

```
record ℕAlg : Set₁ where
  field
    N : Set
    z : N
    s : N → N
```

A slightly more complex example is an algebra for `NF-NE` which has two sorts.

```
record NormalFormAlg : Set₁ where
  field
    F : Set
```

```
E : Set
n : E → F
l : String → F → F
v : String → E
a : E → F → E
```

To encapsulate this structure in a general way, we define an algebra by a record type having two components. Given a sort number from the signature, the first component assigns to it a type, and the second component assigns to every constructor of the sort, a function taking the relevant arguments and generating an element of this type, in other words an actual constructor for the corresponding sort.

```
record Alg (S : Sig) : Set₁ where
  field
    carriers : Fin (sorts S) → Set
    cons : (srt : Fin (sorts S)) (c : Con (sorts S)) →
           c ∈ (cns S) srt → conType srt carriers c
```

The field `carriers` is straightforward, it associates a type to each sort. The field `cons` is more subtle because it needs to associate a function to each of the sort's constructors. The subtlety arises when having to express that we cannot just accept any constructor as an argument for `cons`, but only constructors associated to a particular sort. The constructors for a given sort `srt` in a signature `S` are represented as the list `(cns S) srt` in `Sig`. One possible approach is to refer to the constructor's index in this list instead of its actual constructor representation, and look up its index in the list to obtain the constructor representation. This would, however, result in complex code when using `Alg` later. This is mainly due to lists in Agda being constructed using the [] and :: constructors, so that using them in Agda is facilitated, but using functions like `lookup` is less primitive. We therefore exploit the constructors of `List` and define the data type `_∈_`.

```
data _∈_ {l}{A : Set l}(a : A) : List A → Set where
  hd : {l : List A} → a ∈ (a :: l)
  tl : {l : List A}{b : A} → a ∈ l → a ∈ (b :: l)
```

This type provides the 'proof' that a constructor is in a given list of constructors, by providing its position, which can either be `hd` if it is at the head of the list, or `tl _` if it is in the rest of the list, where `_` is the proof that the constructor is in the tail of the list.

Lastly, the type of the function we associate to each of the sort's constructors is `conType srt carriers c`. We are using the just defined `carriers` to provide `conType` our sorts' types. This function takes a constructor's representation and produces the type of the constructor as a function. It goes through the constructor one argument at a time, using `argType` to find out the type of each argument. If the argument is non-recursive, we use `El` to return its type, whereas if it is recursive, we use `conTypeAux` to go through its list of possible arguments in case it is a recursive function argument. Upon reaching the end of the list of arguments both in `conType` and `conTypeAux`, we return the type of the sort we are constructing.

```
conTypeAux : {n : ℕ} → (Fin n → Set) → List U → Fin n → Set
conTypeAux f [] s = f s
conTypeAux f (set ∷ sets) s = El set → conTypeAux f sets s

argType : {n : ℕ} → (Fin n → Set) → Arg n → Set
argType f (nrec set) = El set
argType f (rec lst fin) = conTypeAux f lst fin

conType : {n : ℕ} → Fin n → (Fin n → Set) → Con n → Set
conType s f (cn []) = f s
conType s f (cn (arg ∷ xs)) = argType f arg →
                              conType s f (cn xs)
```

As an example of their functionality, running

```
conTypeAux {suc zero} (λ {zero → InfTree}) (nat ∷ []) zero
```

gives us ℕ → `InfTree`, the type of `InfTree`'s constructor `sp∞`'s recursive function argument. Also, running

```
conType {suc (suc zero)} zero (λ {zero → NF ; (suc zero) → NE})
(cn (nrec string ∷ rec [] zero ∷ []))
```

gives `String → NF → NF`, the type of the constructor `lam` of `NF-NE`.

It is important to note that the record type `Alg` groups *types* of values together, not actual values. This means that `Alg` is a structure that, given a `Sig`, will provide the types of elements necessary to construct an algebra for the signature. It is then up to us to provide these elements of the given types. We now show some examples to illustrate how to use the `Alg` record type. To construct an ℕ-algebra, we have to provide a type `T : Set` as well as functions of type `T` and `T → T`. First, we construct the familiar ℕ type, which is actually just the initial ℕ-algebra, where `T` = ℕ, `zero : T`, and `suc : T → T`.

```
ℕInit' : Alg ℕSig
ℕInit' = record { carriers = λ {zero → ℕ} ;
                  cons = λ {zero → λ c → λ {hd → zero ;
                                             (tl hd) → suc}} }
```

We can also construct another ℕ-algebra with `T` = `Bool`, `true : Bool`, and `not : Bool → Bool`, where `not true = false` and `not false = true`. Just like the initial ℕ-algebra above represents the natural number Peano representation of zero, suc(zero), suc(suc(zero)), . . . , this ℕ-algebra represents the natural numbers as boolean values depending on whether they are even: true, not(true), not(not(true)), ... Each ℕ-algebra is a different way of representing the natural numbers.

```
BoolℕAlg' : ℕAlg'
BoolℕAlg' = record { carriers = λ {zero → Bool} ;
                     cons = λ {zero → λ c → λ {hd → true ;
                                                (tl hd) → not}} }
```

Another example of the use of `Alg` is the initial `NF-NE`-algebra shown below. This time, we have to provide two types `S` and `T`, and functions of types `T → S`, `String → S → S`, `String → T`, and `T → S → T`.

```
NormalFormInit' : Alg NFNESig
NormalFormInit' = record { carriers = λ {zero → NF ;
                                         (suc zero) → NE} ;
                           cons = λ {zero → λ c → λ {hd → ne ;
                                                      (tl hd) → lam} ;
                                     (suc zero) → λ c → λ {hd → var ;
                                                           (tl hd) → app}} }
```

5.2 Morphisms

We recall from Section 2.8 that algebras over a given endofunctor F form a category, where the objects are the algebras themselves, and a morphism between two algebras (A, α) and (B, β) is a mapping between the carriers $m \colon A \to B$, such that $\beta \circ (F\,m) = m \circ \alpha$. Having defined algebras over a given endofunctor, we now construct morphisms between the algebras.

A morphism from ℕ-algebra n_1 to ℕ-algebra n_2 is described by two main components. Firstly, a map f between the carriers of the algebras N n_1 and N n_2. Secondly, a proof of equality for each constructor of the carriers, ensuring that, given an argument x of type N n_1, applying f to the constructors of n_1 applied to argument x equates to applying the constructors of n_2 to f applied to x.

```
record ℕMor (n₁ n₂ : ℕAlg) : Set where
  field
    f : N n₁ → N n₂
    f_z : f (z n₁) ≡ z n₂
    f_s : (x : N n₁) → f ((s n₁) x) ≡ (s n₂) (f x)
```

A morphism from NF-NE-algebra nf_1 to NF-NE-algebra nf_2 is described as shown below. In this case we have two maps between carriers, one for each sort, and four equality proofs, one for each constructor.

```
record NormalFormMor (nf₁ nf₂ : NormalFormAlg) : Set where
  field
    f_f : F nf₁ → F nf₂
    f_e : E nf₁ → E nf₂
    f_n : (e : E nf₁) → f_f ((n nf₁) e) ≡ (n nf₂) (f_e e)
    f_l : (s : String) (f : F nf₁) → f_f ((l nf₁) s f) ≡
                                    (l nf₂) s (f_f f)
    f_v : (s : String) → f_e ((v nf₁) s) ≡ (v nf₂) s
    f_a : (e : E nf₁) (f : F nf₁) →
           f_e ((a nf₁) e f) ≡ (a nf₂) (f_e e) (f_f f)
```

To generalise these specific morphism examples, we present our definition of a general morphism Mor from the S-algebra A_1 to the S-algebra A_2 as a record type containing two fields.

```
record Mor (S : Sig) (A₁ A₂ : Alg S) : Set where
  field
    f : (srt : Fin (sorts S)) →
        (carriers A₁) srt → (carriers A₂) srt
    eq : (srt : Fin (sorts S)) (c : Con (sorts S))
         (p : c ∈ (cns S) srt) (xs : args srt (carriers A₁) c) →
         (f srt) (apply S A₁ srt c ((cons A₁) srt c p) xs) ≡
         apply S A₂ srt c ((cons A₂) srt c p)
         (map S A₁ A₂ srt c f xs)
```

The field f maps each carrier of A_1 to the corresponding carrier in A_2. The field eq provides an equality proof. Given a sort srt and one of its constructors c, let us call the corresponding constructor in A_1 c_1 and that in A_2 c_2. Also, call the arguments in A_1 to be passed to c_1 args_1, and denote application by @. Allowing for some abuse of notation, eq provides a proof of

$$\mathsf{f} \ @ \ (c_1 \ @ \ \text{args}_1) \equiv c_2 \ @ \ (\mathsf{f} \ @ \ \text{args}_1).$$

In order to define eq, we had to define a number of intermediate functions. We first look at the function args, which takes a constructor and returns a product type of the arguments to be passed to that constructor. It does this by going through the constructor's arguments one by one and using argType (defined previously) on each argument.

```
args : {n : ℕ} → Fin n → (Fin n → Set) → Con n → Set
args s f (cn []) = ⊤
args s f (cn (x ∷ xs)) = argType f x × args s f (cn xs)
```

As an example, the type of arguments of the constructor lam : String → NF → NE can be obtained by running

```
args {suc (suc zero)} zero (λ {zero → NF ; (suc zero) → NE})
(cn (nrec string ∷ rec [] zero ∷ []))
```

to get String × NF × ⊤. (The ⊤ at the end is due to our definition of args, but an argument of type ⊤ can easily be provided: tt.) We note that args and conType are defined very similarly, in fact, args only differs from conType in that it eliminates the return type of the constructor so we only have its arguments, and uncurries the arguments.

The next function we look at is apply. This applies the constructor c having type conType srt (carriers A) c to the supplied arguments. When the constructor has no arguments, we do not perform any applications and can simply return the constructor, which has type (carriers A) srt (replace c in conType srt (carriers A) c with cn [] and look at conType's definition). When the constructor has one or more arguments, the provided arguments args are in the form of a tuple, so that we can apply each argument at a time to the constructor until the end of the list of arguments.

```
apply : (S : Sig) (A : Alg S) (srt : Fin (sorts S))
        (c : Con (sorts S)) → conType srt (carriers A) c →
        args srt (carriers A) c → (carriers A) srt
apply S A srt (cn []) type argsEq = type
apply S A srt (cn (x :: xs)) type (arsX , arsXs) =
    apply S A srt (cn xs) (type arsX) arsXs
```

Running

```
apply ℕSig ℕInit' zero (cn []) zero tt
```

returns 0 (the first natural number, not the constructor), while running

```
apply NFNESig NormalFormInit' zero (cn (nrec string ::
rec [] zero :: [])) lam ("x" , ne (var "y") , tt)
```

applies the arguments ("x" , ne (var "y") , tt) to the constructor lam : String → NF → NE and returns lam "x" (ne (var "y")).

The last function we need is map. For two S-algebras A_1 and A_2 and sort srt, this function maps arguments of type (carriers A_1) srt to arguments of type (carriers A_2) srt. Looking back at our ℕMor example, map is emulating the (f x) in f_s : (x : N n_1) → f ((s n_1) x) ≡ (s n_2) (f x). map pattern matches on the constructor and goes through its arguments one by one. Similarly to what we saw for the definition of conType, it makes use of the auxiliary functions mapArgType, which deals with individual arguments, and mapConTypeAux, which takes care of recursive function arguments.

```
module _(S : Sig) (A₁ A₂ : Alg S) (srt : Fin (sorts S)) where

    mapConTypeAux : (fin : Fin (sorts S)) (lst : List U)
                    (f : (srt : Fin (sorts S)) →
```

```
                   (carriers A₁) srt → (carriers A₂) srt) →
                   conTypeAux (carriers A₁) lst fin →
                   conTypeAux (carriers A₂) lst fin
    mapConTypeAux fin [] f cta = f fin cta
    mapConTypeAux fin (x ∷ xs) f cta =
        λ s → mapConTypeAux fin xs f (cta s)

    mapArgType : (a : Arg (sorts S))
                 (f : (srt : Fin (sorts S)) →
                 (carriers A₁) srt → (carriers A₂) srt) →
                 argType (carriers A₁) a →
                 argType (carriers A₂) a
    mapArgType (nrec x) f ars = ars
    mapArgType (rec lst fin) f ars = mapConTypeAux fin lst f ars

    map : (c : Con (sorts S))
          (f : (srt : Fin (sorts S)) →
          (carriers A₁) srt → (carriers A₂) srt) →
          args srt (carriers A₁) c → args srt (carriers A₂) c
    map (cn []) f ars = ars
    map (cn (x ∷ xs)) f (arType , ars) =
        mapArgType x f arType , map (cn xs) f ars
```

As with the definition of `Alg`, `Mor` provides the structure for a morphism between two given algebras, and it is up to us to populate the structure with elements. Consider the following morphism between two algebras that we have already seen, `ℕInit'` and `BoolℕAlg'`. The function `even` between the carriers ℕ and `Bool` encapsulates the relationship between the two algebras: even numbers are mapped to `true`, and odd numbers are mapped to `false`. The equality proofs, the types of which are worked out by `eq`, hold trivially using `refl`. This stands witness to how much we can express using types—using a general way of expressing `eq` pays off as Agda can, in cases which are not too complex, work out the equalities automatically.

```
even : ℕ → Bool
even zero = true
even (suc n) = not (even n)
```

```
MorℕEven'' : Mor ℕSig ℕInit' BoolℕAlg'
MorℕEven'' = record { f = λ {zero → even} ;
                      eq = λ {zero → λ c → λ {hd ar → refl ;
                                               (tl hd) ar → refl}} }
```

5.3 The Initial Algebra

`Alg` and `Mor` correspond to the objects and maps in the category of F-algebras for a given endofunctor F. Now that we have a complete picture of the F-algebra, we can define its initial object, the initial algebra. The initial algebra is characterised by the property that there exists exactly one morphism from this algebra to any other algebra. This morphism will be dealt with later when discussing the iterator, for now we focus on the initial algebra's carriers and constructors, i.e. we construct a pair (I, ι).

Being an algebra, the initial algebra is of type `Alg` and hence consists of carriers and constructors. We note that this time, contrary to what we did for `Alg` and `Mor`, we are not defining the framework for a particular structure, but actually populating a framework we have previously defined, `Alg`. Indeed, we are not defining a record type of type `Set` but an element of type `Alg`.

The idea here is that given a signature of type `Sig`, we emulate the inductive type this signature is representing, i.e. its sorts and constructors. For example, consider ℕ's signature `ℕSig` below.

```
ℕSig : Sig
sorts ℕSig = 1
cns ℕSig = λ {zero → cn [] -- zero
                  ∷ cn (rec [] zero ∷ []) -- suc
                  ∷ []}
```

We want to construct some data type `⟦ℕSig⟧` acting as the carrier of the algebra, and constructors `z` and `s` as follows

```
⟦ℕSig⟧ : Set
z : ⟦ℕSig⟧
s : ⟦ℕSig⟧ → ⟦ℕSig⟧
```

that emulate the inductive type

```
ℕ : Set
zero : ℕ
suc : ℕ → ℕ.
```

To construct the data type ⟦ℕSig⟧, we have to keep in mind what happens in Agda when we pattern match on an element `n` of ℕ. The value `n` is replaced by its two possible values—either `zero`, or `suc n'` for some other natural number `n'`. In the latter case, `n'` is the argument that is passed to `suc` to produce an element of type ℕ. This simple example illustrates how an element of type ℕ contains the arguments that need to be passed to its constructor. Hence in general, when constructing an element of type ⟦S⟧ for some signature `S`, the element must also contain this same data. This is why we define ⟦_⟧_ and `I-Args` mutually as shown below.

```
data ⟦_⟧_ (S : Sig) : Fin (sorts S) → Set
data I-Args (S : Sig) : (srt : Fin (sorts S)) (c : Con (sorts S))
                        → c ∈ cns S srt → Set

data ⟦_⟧_ S where
    con : (srt : Fin (sorts S)) (c : Con (sorts S))
          (p : c ∈ cns S srt) → I-Args S srt c p → ⟦ S ⟧ srt

data I-Args S where
    arg : {srt : Fin (sorts S)} {c : Con (sorts S)}
          {p : c ∈ cns S srt} → args srt (⟦ S ⟧_) c →
          I-Args S srt c p
```

Given a signature `S` and a sort `srt`, ⟦S⟧ `srt` is the type standing for sort `srt` in `S`. For instance, ⟦ℕSig⟧ `zero` stands for ℕ in the example above, while ⟦NFNESig⟧ `zero` stands for `NF` and ⟦NFNESig⟧ `(suc zero)` stands for `NE` in `NF-NE`. To construct an element of ⟦S⟧ `srt`, we specify the sort number, constructor, position of the constructor in the list of constructors, and arguments for the constructor using the type `I-Args`. We could not simply add `args srt (⟦ S ⟧_) c` to the constructor `con` of ⟦_⟧ _ due to the usage of the type ⟦_⟧ _ in the function call, the same type we are just defining. We therefore define the data type `I-Args` mutually with ⟦_⟧ _.

The carriers part of the initial algebra Initial is thus complete. What remains is defining the constructors of these carriers, which is achieved using the function makeCons.

```
Initial : (S : Sig) → Alg S
Initial S = record { carriers =  λ srt → ⟦ S ⟧ srt ;
                     cons = λ srt c p → makeCons S srt c p }
```

Before we get into the definition of makeCons, we look more closely at what we want to define. We walk through the specific case of defining the initial algebra for ℕSig. We set the type of the only carrier to ⟦ℕSig⟧ zero and have a look at the type of the constructors we have to provide.

```
tstℕ : Alg ℕSig
tstℕ = record { carriers = λ {zero → ⟦ ℕSig ⟧ zero} ; -- N
                cons = λ {zero → λ c → λ {hd → {!!} ;  -- z
                                          (tl hd) → {!!}}} } --s
```

Using our previous definition of Alg, Agda tells us that the type of the first goal is ⟦ℕSig⟧ zero and the type of the second goal is ⟦ℕSig⟧ zero → ⟦ℕSig⟧ zero. The first constructor represents the zero constructor which has no arguments, while the second constructor represents suc which has one recursive argument. We start defining the constructors as follows.

```
tstℕ : Alg ℕSig
tstℕ = record { carriers = λ {zero → ⟦ ℕSig ⟧ zero} ; -- N
                cons = λ {zero → λ c →
                         λ {hd → con zero c hd (arg {!!}) ; -- z
                           (tl hd) → λ n' →
                           con zero c (tl hd) (arg {!!})}} } -- s
```

The zero constructor takes no arguments and is hence not a function. The type of arguments we have to provide is args zero (⟦ S ⟧_) (cn []), which computes to ⊤, hence we simply write tt. The suc constructor is a function taking n' : ⟦ℕSig⟧ zero. The type of arguments we have to provide for this constructor is args zero (⟦ S ⟧_) (cn (rec [] zero ∷ [])), which computes to ⟦ℕSig⟧ zero × ⊤. The arguments we provide should therefore be n' , tt.

```
tstℕ : Alg ℕSig
tstℕ = record { carriers = λ {zero → ⟦ ℕSig ⟧ zero} ; -- N
```

```
cons = λ {zero → λ c →
           λ {hd → con zero c hd (arg tt) ; -- z
             (tl hd) → λ n' →
            con zero c (tl hd) (arg (n' , tt))}} } -- s
```

We now move on to our implementation of makeCons. Our aim is to provide constructors like the ones we saw in the example above. makeCons takes a constructor and passes it on to makeConsAux. The function makeConsAux takes two lists of arguments: l_1 is the part of the constructor it has already processed, and l_2 is the unprocessed part of the constructor. For a constructor cn x, makeCons calls makeConsAux with l_1 = [] and l_2 = x, as initially the whole constructor is unprocessed. makeConsAux then processes the constructor one argument at a time, moving processed arguments from l_2 to l_1, while at the same time updating the arguments for l_1, which it obtains as inputs since these constructors are functions. Once it has gone through the whole constructor and l_2 is empty, it constructs an element of type ⟦S⟧ srt containing all the accumulated arguments.

```
makeConsAux : (S : Sig) (srt : Fin (sorts S))
              (l₁ l₂ : List (Arg (sorts S))) →
              cn (l₁ ++ l₂) ∈ cns S srt →
              args srt (⟦_⟧_ S) (cn l₁) →
              conType srt (⟦_⟧_ S) (cn l₂)
makeConsAux S srt l₁ [] p ars = con srt (cn l₁) p (arg ars)
makeConsAux S srt l₁ (x :: xs) p ars =
   λ a → makeConsAux S srt (l₁ ::ʳ x) xs p (argsSnoc S srt l₁ x ars a)

makeCons : (S : Sig) (srt : Fin (sorts S)) (c : Con (sorts S)) →
           c ∈ cns S srt → conType srt (⟦_⟧_ S) c
makeCons S srt (cn x) p = makeConsAux S srt [] x p tt
```

makeConsAux uses the helper function argsSnoc which, given arguments for cn l and an argument for the Arg x, returns arguments of type cn (l ::ʳ x).

```
argsSnoc : (S : Sig) (srt : Fin (sorts S))
           (l : List (Arg (sorts S))) (x : Arg (sorts S)) →
           args srt (⟦_⟧_ S) (cn l) → argType (⟦_⟧_ S) x →
           args srt (⟦_⟧_ S) (cn (l ::ʳ x))
```

```
argsSnoc S srt [] x ars ar = ar , ars
argsSnoc S srt (l :: ls) x (arl , arls) ar =
    arl , argsSnoc S srt ls x arls ar
```

It also uses two rewrite rules, which allow us to prove equalities and subsequently extend Agda's evaluation relation using these new rules. The first rewrite rule 'convinces' Agda that appending an empty list to any list leaves the list unchanged, while the second says that appending an element `a` to a list `xs` and then appending the list `ys` to the result is equivalent to prepending `a` to `ys`, and then appending this to `xs`.

```
appendNilPf : {A : Set} (l : List A) → l ++ [] ≡ l
appendNilPf [] = refl
appendNilPf (x :: xs) = cong (_::_ x) (appendNilPf xs)

postulate appendNil : {A : Set} (l : List A) → l ++ [] ≡ l
{-# REWRITE appendNil #-}

snocAppendPf : {A : Set} (xs ys : List A) (a : A) →
               (xs ::ʳ a) ++ ys ≡ xs ++ a :: ys
snocAppendPf [] ys a = refl
snocAppendPf (l :: ls) ys a =
    cong (_::_ l) (snocAppendPf ls ys a)

postulate snocAppend : {A : Set} (xs ys : List A) (a : A) →
                       (xs ::ʳ a) ++ ys ≡ xs ++ a :: ys
{-# REWRITE snocAppend #-}
```

These two rules allow us more flexibility when providing the proof `cn (l₁ ++ l₂) ∈ cns S srt` in `makeConsAux`.

5.4 The Iterator

Although we have constructed the carriers and constructors of the initial algebra, we have yet to construct its defining feature, a unique morphism from this algebra to any other algebra. In this section, we construct such a morphism, which is called the iterator, and in the next section we show that it is unique. An object in a category having a morphism from it to any other object is called *weakly initial*. By the end of this section, we will therefore have defined a weakly initial algebra.

Being a morphism, the iterator `It` is of type `Mor` and hence consists of two components, a function `f` from the carriers of `Initial S` to the carriers of the given algebra `A`, and an equality proof ensuring that `f` preserves structure.

```
It : (S : Sig) (A : Alg S) → Mor S (Initial S) A
It S A = record { f = λ srt → funcs S A srt ;
                  eq = λ srt c p {xs} → eqProof S A srt c p xs }
```

For the first field of our construction, we need to define a function `funcs` of the following type.

```
funcs : (S : Sig) (A : Alg S) (srt : Fin (sorts S)) →
        ⟦ S ⟧ srt → carriers A srt
```

Given a sort number `srt`, this function takes an element of the type standing for `srt` in the initial algebra, `⟦S⟧ srt`, and returns an element of the corresponding type `(carriers A) srt`. Let us illustrate this by an example. Consider the initial algebra for ℕ which we call `InitialℕAlg`, and the Boolean ℕ-algebra `BoolℕAlg'`.

```
InitialℕAlg : Initial ℕSig
⟦ ℕSig ⟧ zero : Set
zero : ⟦ ℕSig ⟧ zero
suc : ⟦ ℕSig ⟧ zero → ⟦ ℕSig ⟧ zero
```

```
BoolℕAlg' : Alg ℕSig
Bool : Set
true : Bool
not : Bool → Bool
```

In this scenario, `funcs` should map `zero` to `true` and `suc n` for `n : ⟦ℕSig⟧ zero` to `suc b` for `b : Bool`. This involves first converting arguments of type `⟦ℕSig⟧ zero` to arguments of type `Bool`, and then applying these arguments to the corresponding constructor in `BoolℕAlg'`. More generally, for an ℕ-algebra `X`, we need the following function.

```
fℕ' : (X : Alg ℕSig) → carriers (Initial ℕSig) zero →
      carriers X zero
fℕ' X (con .0F .(cn []) hd (arg ar)) = (cons X) zero (cn []) hd
fℕ' X (con .0F .(cn (rec [] 0F ∷ [])) (tl hd) (arg (n , tt))) =
    (cons X) zero (cn (rec [] zero ∷ [])) (tl hd) (fℕ' X n)
```

Note how we pattern match on the element of the initial algebra, and call the constructors of the algebra `X`, applying any relevant arguments after having mapped these arguments using `fℕ'` recursively.

The above example is fairly straightforward. For mutual inductive types with more than one sort, we would have a function like `fℕ'` for each sort, with their definitions calling each other recursively.

We now define `funcs` as follows.

```
funcs srt (con .srt c p (arg ar)) =
    apply S A srt c (cons A srt c p) (argsInitToCarr srt c ar)
```

`funcs` applies mapped arguments to the relevant constructors in the algebra `A`. The function `argsInitToCarr` takes care of the mapping of arguments.

```
argTypeInitToCarr : (a : Arg (sorts S)) → argType (⟦_⟧_ S) a →
                    argType (carriers A) a
argTypeInitToCarr (nrec set) arType = arType
argTypeInitToCarr (rec lst fin) arType = mapCon fin lst arType


argsInitToCarr : (c : Con (sorts S)) → args srt (⟦_⟧_ S) c →
                 args srt (carriers A) c
argsInitToCarr (cn []) ars = ars
argsInitToCarr (cn (x ∷ xs)) (arType , ars) =
    argTypeInitToCarr x arType , argsInitToCarr (cn xs) ars
```

`argsInitToCarr` goes through a constructor one argument at a time, calling `argTypeInitToCarr` at each step. We note that these functions are almost identical to the functions `map` and `mapArgType` respectively, which we defined in Section 5.2. Before explaining why we could not use these previously defined functions, we show our definition of our last function `mapCon`.

```
mapCon : (fin : Fin (sorts S)) (lst : List U) →
         conTypeAux (⟦_⟧_ S) lst fin →
         conTypeAux (carriers A) lst fin
mapCon fin [] cta = funcs fin cta
mapCon fin (x ∷ xs) cta = λ s → mapCon fin xs (cta s)
```

`mapCon` is itself almost identical to the function `mapConTypeAux`, also defined in Section 5.2. The reason we could not use the previously defined functions and had to write new ones is that they require a function of type

```
(srt : Fin (sorts S)) → (carriers A₁) srt → (carriers A₂) srt,
```

but this would be precisely the function `funcs` that we are trying to define. Had we used these functions in our definition of `funcs`, this would have been defined as

```
funcs srt (con .srt c p (arg ar)) =
    apply S A srt c (cons A srt c p) (map S (Initial S) A srt
    c funcs ar)
```

and since we would be using `funcs` as an argument to `map` in its own definition, we would get a non-termination error as Agda cannot make sure that we are calling `funcs` on a smaller argument. However, we obtain the same effect by defining the three functions shown above, with `mapCon`'s base case calling `funcs` (on a possibly different sort number than the one in the original `funcs` call, depending on whether `srt` is equal to `fin`). Because we call `mapCon` from `funcs`, and `funcs` from `mapCon`, these two functions are mutually defined.

We now focus on the second field of `It`, the equality proof. The equality that we need to prove is shown below.

```
eqProof : (S : Sig) (A : Alg S) (srt : Fin (sorts S))
          (c : Con (sorts S)) (p : c ∈ cns S srt)
          (xs : args srt (⟦_⟧_ S) c) →
          funcs S A srt (apply S (Initial S) srt c
          (makeCons S srt c p) xs) ≡
          apply S A srt c (cons A srt c p) (map S
          (Initial S) A srt c (funcs S A) xs)
```

The proof statement as is is complex and not straightforward to prove, so we will break it down into multiple sub-proofs. We will prove this in a sequence of steps from top to bottom, with the left hand side of the equality as the top of the proof and the right hand side as the bottom of the proof, so that we can simplify the bottom and make our way up, and simplify the top and make our way down, so long as we can make both ends meet in the middle. This can be written in Agda by importing the `≡-Reasoning` module. So far, we have the following.

```
begin
 funcs S A srt (apply S (Initial S) srt c (makeCons S srt c p) ars)
    ?
 apply S A srt c (cons A srt c p) (map S (Initial S) A srt c
 (funcs S A) ars)
■
```

We need to figure out a sequence of steps to prove the equality of these two statements. If we look at the bottom of the proof closely, we notice that this is precisely what we said `funcs` would have been written as had we used our previously defined functions `map`, `mapArgType`, and `mapConTypeAux`. This intuition drives us to attempt to provide a proof that the bottom part of our proof is equal to our actual definition of `funcs`. This proof would go in the second ≡⟨ ? ⟩ symbol below.

```
begin
 funcs S A srt (apply S (Initial S) srt c (makeCons S srt c p) ars)
 ≡⟨ ? ⟩
 apply S A srt c (cons A srt c p) (argsInitToCarr S A srt c ars)
 ≡⟨ ? ⟩
 apply S A srt c (cons A srt c p) (map S (Initial S) A srt c
 (funcs S A) ars)
■
```

Indeed, it is easy to prove that `argsInitToCarr` behaves like `map` and that `argTypeInitToCarr` behaves like `mapArgType`, although these rely on our proof that `mapCon` behaves like `mapConTypeAux`.

```
pfMapArgType : (srt : Fin (sorts S)) (x : Arg (sorts S))
               (at : argType (⟦_⟧_ S) x) →
               argTypeInitToCarr srt x at ≡
               mapArgType S (Initial S) A srt x funcs at
pfMapArgType srt (nrec x) at = refl
pfMapArgType srt (rec lst fin) at = pfMapConTypeAux srt fin lst at

eqTuple : {A B : Set} {a a' : A} {b b' : B} → a ≡ a' → b ≡ b'
                                            → (a , b) ≡ (a' , b')
eqTuple {_} {_} {a} {.a} {b} {.b} refl refl = refl
```

```
pfMap : (srt : Fin (sorts S)) (c : Con (sorts S))
        (ar : args srt (⟦_⟧_ S) c) →
        argsInitToCarr srt c ar ≡ map S (Initial S) A srt c funcs ar
pfMap srt (cn []) ar = refl
pfMap srt (cn (x ∷ xs)) (fst , snd) =
    eqTuple (pfMapArgType srt x fst) (pfMap srt (cn xs) snd)
```

The proof that mapCon behaves like mapConTypeAux is less straightforward since, because of the way the two functions are defined, in the inductive case we have to prove something of the form λ a → f a ≡ λ a → g a knowing that f a ≡ g a for all a. This principle is known as *functional extensionality* and is not provable in intensional type theory, and hence in Agda using the standard libraries. Within this setting, refl is the only constructor of the equality type, which equates definitionally equal objects, but λ a → f a and λ a → g a are not definitionally equal. To get around this, we could use Agda's Cubical mode which views (proof-relevant) equalities as paths on the unit interval, and using which functional extensionality is easily proved. After trying to use the Cubical libraries, the problem with this approach is that it adds restrictions to Agda's termination checker due to enabling the without-K option, which tells Agda not to assume the K axiom, which roughly states that any equality proof is equivalent to refl. This results in our previous definition of funcs not passing the termination checker, which is still under development for the Cubical mode and might still lack some features. Therefore, we opt for the other option, which is to assume this principle by declaring it as a postulate, and the rest of the proof follows easily.

```
postulate funExt : ∀ {ℓ} {A : Set ℓ} {B : A → Set ℓ}
                   {f g : (x : A) → B x} →
                   ((x : A) → f x ≡ g x) → f ≡ g

pfMapConTypeAux : (srt fin : Fin (sorts S)) (l : List U)
                  (cta : conTypeAux (⟦_⟧_ S) l fin) →
                  (mapCon fin l cta) ≡
                  (mapConTypeAux S (Initial S) A srt fin l funcs cta)
pfMapConTypeAux srt fin [] cta = refl
```

```
pfMapConTypeAux srt fin (x :: xs) cta =
    funExt (λ s → pfMapConTypeAux srt fin xs (cta s))
```

We can now fill in the missing sub-proof in the main proof by applying `cong` to `pfMap`. `congruence` states that if two expressions are equal, they remain equal after applying the same function to them (Wadler et al., 2020).

```
begin
 funcs S A srt (apply S (Initial S) srt c (makeCons S srt c p) ars)
 ≡⟨ ? ⟩
 apply S A srt c (cons A srt c p) (argsInitToCarr S A srt c ars)
 ≡⟨ cong (apply S A srt c (cons A srt c p)) (pfMap S A srt c ars) ⟩
 apply S A srt c (cons A srt c p) (map S (Initial S) A srt c (funcs S A) ars)
■
```

The bottom can now be reduced in one step to the left hand side of the definition of `funcs`.

```
begin
 funcs S A srt (apply S (Initial S) srt c (makeCons S srt c p) ars)
 ≡⟨ ? ⟩
 funcs S A srt (con srt c p (arg ars))
 ≡⟨ refl ⟩
 apply S A srt c (cons A srt c p) (argsInitToCarr S A srt c ars)
 ≡⟨ cong (apply S A srt c (cons A srt c p)) (pfMap S A srt c ars) ⟩
 apply S A srt c (cons A srt c p) (map S (Initial S) A srt c (funcs S A) ars)
■
```

All that remains now to complete our proof is to show that `(apply S (Initial S) srt c (makeCons S srt c p) ars)` results in `(con srt c p (arg ars))`. Intuitively, this makes sense since `apply` uses the constructors of our initial algebra, defined using `makeConsAux` by going through the constructor `c` and accumulating its arguments, constructing an element of type `⟦S⟧ srt` using `con`, and placing the accumulated arguments in `arg ars`, the result of which is `con srt c p (arg ars)`. We prove this equality using `apply≡con` shown below.

```
apply≡con : (c : Con (sorts S)) (p : c ∈ cns S srt)
            (ars : args srt (⟦_⟧_ S) c) →
```

```
            apply S (Initial S) srt c (makeCons S srt c p) ars ≡
            con srt c p (arg ars)
apply≡con (cn []) p tt = refl
apply≡con (cn (x :: xs)) p (fst , snd) =
    apply≡conMca x xs p fst snd
```

apply≡con calls an auxiliary function apply≡conMca, which in turn calls another auxiliary function, and so on. We will not list all of this code here as the process is quite mechanical and verbose, although the full proof is available in the supplementary material. It suffices to say that we use these intermediate functions to analyse each step of our construction of con srt c p (arg ars) and prove properties about the functions constructing it. The least obvious part is perhaps the definition of the function appArgs, which acts as a repeated version of the function argsSnoc, and is used to prove properties about it.

```
appArgs : (l₁ l₂ : List (Arg (sorts S))) →
          args srt (⟦_⟧_ S) (cn l₁) →
          args srt (⟦_⟧_ S) (cn l₂) →
          args srt (⟦_⟧_ S) (cn (l₁ ++ l₂))
appArgs [] l₂ a₁ a₂ = a₂
appArgs (x :: xs) [] a₁ a₂ = a₁
appArgs (x :: xs) (y :: ys) a₁ a₂ =
    proj₁ a₁ , appArgs xs (y :: ys) (proj₂ a₁) a₂
```

Once all the intermediate proofs are completed, we can finally complete our main proof by once again applying congruence, this time to apply≡con.

```
begin
 funcs S A srt (apply S (Initial S) srt c (makeCons S srt c p) ars)
 ≡⟨ cong (funcs S A srt) (apply≡con S srt c p ars) ⟩
 funcs S A srt (con srt c p (arg ars))
 ≡⟨ refl ⟩
 apply S A srt c (cons A srt c p) (argsInitToCarr S A srt c ars)
 ≡⟨ cong (apply S A srt c (cons A srt c p)) (pfMap S A srt c ars) ⟩
apply S A srt c (cons A srt c p) (map S (Initial S) A srt c (funcs S A) ars)
∎
```

5.5 Uniqueness of the Iterator

In the previous section we constructed a weakly initial algebra for a given signature. In this section, we prove that this algebra is unique, making it an initial algebra.

The statement that we want to prove is shown below. Any morphism from the initial S-algebra to another S-algebra is actually equivalent to the iterator.

```
uIt : (S : Sig) (A : Alg S) (f : Mor S (Initial S) A) → f ≡ It S A
```

Since f and It S A are both morphisms, which are defined as record types, we need to equate two records by equating each one of their fields. Doing this is, however, not as straightforward as simply proving

```
mor≡intro' : (S : Sig) (A : Alg S) (m₁ m₂ : Mor S (Initial S) A) →
                f m₁ ≡ f m₂ → eq m₁ ≡ eq m₂ → m₁ ≡ m₂.
```

Indeed, this type does not type check. The field eq in Mor depends on the other field f, so eq m_1 depends on f m_1 while eq m_2 depends on f m_2, meaning that eq m_1 and eq m_2 have different types and cannot be related by ≡. Obviously, we know that to get to that part of the type, we would have proved that f m_1 ≡ f m_2, so eq m_1 and eq m_2 *do* have the same type, but we have not expressed this to Agda yet and hence it gives us an error.

To express equality of dependent record types, we first look at equality for Σ-types, of which record types are a generalisation. The equality of Σ-types is a Σ-type of equalities. To prove that (a , b) ≡ (a’ , b’), we provide a proof that a ≡ a’, and given this, we prove that the value/proof obtained by substituting a with a’ in a predicate B, whose proof that it holds for a is b, is equivalent to b’.

```
Σ≡intro : ∀ {α β}{A : Set α}{B : A → Set β}{a a' : A}{b : B a}{b' : B a'}
          → (Σ (a ≡ a') λ p → subst B p b ≡ b')
          → (a , b) ≡ (a' , b')
Σ≡intro (refl , refl) = refl
```

To prove that two morphisms mor fun e and mor fun’ e’ are equal, we write an ≡-introduction rule similar to the above, where a = fun, a’ = fun’, b = e and b’ = e’ (here we are omitting the implicit arguments in the type).

```
mor≡intro : (p : fun ≡ fun') → (subst (λ fun → (srt : Fin (sorts S))
             (c : Con (sorts S)) (p : c ∈ (cns S) srt)
             (xs : args srt (carriers A₁) c) →
             (fun srt) (apply S A₁ srt c ((cons A₁) srt c p) xs) ≡
             apply S A₂ srt c ((cons A₂) srt c p)
             (map S A₁ A₂ srt c fun xs)) p e) ≡ e' →
             mor fun e ≡ mor fun' e'
mor≡intro refl refl = refl
```

We employ `mor≡intro` to prove `uIt`, so the first thing we have to do is prove that for a given morphism `m`, `f m ≡ f (It S A)`.

```
uItF : (S : Sig) (A : Alg S) (srt : Fin (sorts S)) (i : ⟦ S ⟧ srt)
       (m : Mor S (Initial S) A) → f m srt i ≡ f (It S A) srt i
```

Proving `uItF` is similar to proving `eqProof` from Section 5.4, so we will not be listing all the code here. Most importantly, we used the function `apply≡con` defined in that section, as well as functions almost identical to `pfMap`, `pfMapArgType`, and `pfMapConTypeAux`, this time written in terms of `f m` for a morphism `m` instead of `funcs`. Our counterpart to `pfMapConTypeAux`, `pfMapConTypeAux'`, uses `uItF` in its base case, hence these two functions are defined mutually, as well as `funExt` in its inductive case. Even though we are defining very similar functions more than once, the reason we cannot generalise these functions by defining them so they have an argument of type `(f : (srt : Fin (sorts S)) → ⟦ S ⟧ srt → carriers A srt)` and swapping out the `f` as required, is that doing so would result in a non-termination error, as described already in Section 5.4.

```
uItF S A srt (con .srt c p (arg ar)) (mor fm eqm) =
   subproof S A srt c p ar (mor fm eqm)
  (trans (sym (cong (fm srt) (apply≡con S srt c p ar))) (eqm srt c p ar))
```

Now that we have proved that the first fields `f` of the two records are equivalent, we need to prove the equivalence of the second fields `eq`. Since we are using the definition of equality that can only be constructed in one way, two equality proofs having the same type are automatically equivalent—they must both be `refl`. To make this explicit and prove our goal `uIt`, we can take several approaches. One option is to alter our `Alg` definition to add the constraint that any carrier has to be what is called an *h-set* in homotopy type theory, or in other words, it has to have the property that any two of its elements can be equivalent in at most one way. To achieve this, we can either express this explicitly as a type, or we can assign carriers the type `Prop`, which is like `Set` except all the elements of a type having type `Prop` are definitionally equal. We decided against using one of these options as this would affect many of the definitions that depend on `Alg`, although we might opt for one of them in our future work when we have more time to make these changes. What we did here instead was use the principle of *uniqueness of identity proofs (UIP)* directly in the proof of `uIt`.

```
UIP : ∀ {a} {A : Set a} {x y : A} (p q : x ≡ y) → p ≡ q
UIP refl refl = refl

uIt : (S : Sig) (A : Alg S) (f : Mor S (Initial S) A) → f ≡ It S A
uIt S A (mor fm eqm) with
   (funExt S A (λ s → funExt S A (λ i → uItF S A s i (mor fm eqm))))
uIt S A (mor .(funcs S A) eqm) | refl =
   mor≡intro refl (funExt (λ srt → funExt (λ c → funExt (λ p →
   funExt (λ xs → UIP (eqm srt c p xs) (eqProof S A srt c p xs))))))
```

`funExt` was used in the proof of equivalence of both `f` and `eq` to, for the first case, turn a type `(srt : Fin (sorts S)) (i : ⟦S⟧ srt) (m : Mor S (Initial S) A) → f m srt i ≡ funcs S A srt i` into an equality of type `f m ≡ funcs S A`, and similarly for `eq`.

Having proved that the iterator is unique means that we have concluded our constructive proof of the statement ‘every inductive type has an initial algebra’. Throughout this chapter, we have also provided a full specification of simple and mutual inductive types.

6

Constructing WI-types

Our next objective was to reduce all simple and mutual inductive types to a singular inductive type. This chapter details our construction of indexed W-types, or WI-types, and a starting point for reducing inductive types to WI-types. This chapter focusses on the indexed variant of W-types instead of simple W-types because in the future, we aim to extend our work to include more general inductive types, like inductive families, which cannot be represented by W-types directly, but can be represented by WI-types (which can then be reduced to W-types). Our work here is thus better suited to future extensions than if we had just considered W-types, and in any case any W-type representation of an inductive type can be easily converted to a WI-type representation.

6.1 WI-Types Introduction and Examples

WI-types are the indexed version of W-types, to which they have been shown to be reducible (Altenkirch and Morris, 2009; Altenkirch et al., 2015), so showing that simple and mutual inductive types are reducible to WI-types automatically implies they are reducible to W-types. In order to construct a reduction from inductive types to WI-types, i.e. to show that the WI-type is able to represent any inductive type, we first assume the existence of the WI-type, written in Agda as follows.

```
data WI (I:Set) (S:I → Set) (P:(i:I) → S i → I → Set) : I → Set where
    sup : (i:I) (s:S i) → ((j:I) → P i s j → WI I S P j) → WI I S P i
```

We also assume the existence of the WI-type's eliminator, which allows us to define functions out of the WI-type. (In reality, the WI-type and its eliminator can be derived from the W-type and its eliminator, so we should have assumed the latter instead, but this would have complicated our constructions so we leave it for future work.) Next, we take a particular inductive type's signature `S` and construct the types `I`, `S`, and `P` above to obtain the WI-type representation of the inductive type. We then also derive the constructors for this type, at which point we will have obtained the WI-type algebra associated to `S`. Finally, we construct a morphism from this algebra to any other `S`-algebra and show that this morphism is unique, proving that the WI-type algebra is isomorphic to the initial algebra of `S` and hence to the original inductive type. Although we have not achieved all of the steps in this process, we go through the ones that we have achieved, and present some ideas on how to implement the steps that follow. This first section introduces WI-types and presents WI-type representations of types we have already seen.

To model any inductive type using the WI-type, we need to define `I`, `S`, and `P` for the type. `I : Set` is a type representing all the sorts of the inductive type. `S : I → Set` refers to a constructor of a given sort by specifying the types of the constructor's non-recursive arguments. Having specified a sort `i` and a constructor `S i`, `P : (i : I) → S i → I → Set` expresses the number of recursive arguments to be passed to the constructor `S i` that are of the type corresponding to sort `j : I`.

We provide some of our inductive type examples expressed as WI-types. We begin with the representation for ℕ. Firstly, ℕ only has one sort, therefore `I₂ = ⊤` (to represent two sorts we use the type ⊤ ⊎ ⊤, for three sorts ⊤ ⊎ ⊤ ⊎ ⊤, and so on). Neither of this sort's constructors has non-recursive arguments, and since there are two of them, `S₂ tt = ⊤ ⊎ ⊤`. Lastly, the first constructor `zero` has no recursive arguments, hence `P₂ tt (inj₁ tt) tt = ⊥`, but the second constructor `suc` has one, hence `P₂ tt (inj₂ tt) tt = ⊤`. Using the sets we just defined, ℕ's constructors would then be written as `zero''` and `suc''` below.

```
I₂ = ⊤

S₂ : I₂ → Set
S₂ tt = ⊤ ⊎ ⊤

P₂ : (i : I₂) → S₂ i → I₂ → Set
```

```
P₂ tt (inj₁ tt) tt = ⊥
P₂ tt (inj₂ tt) tt = ⊤

zero'' : WI I₂ S₂ P₂ tt
zero'' = sup tt (inj₁ tt) λ {tt → λ ()}

suc'' : WI I₂ S₂ P₂ tt → WI I₂ S₂ P₂ tt
suc'' n = sup tt (inj₂ tt) (λ {tt → λ {tt → n}})
```

Upon choosing the first constructor in `zero”` with `inj₁ tt`, when having to input the recursive arguments for the constructor, we end up at the empty map `λ ()`. This represents that recursion has ended, or that we have reached an element with no subtree/s if we look at the WI-type as a well-ordered tree. The situation for `suc”` is different, as there we pass the recursive argument `n`.

`Lam` expressed as a WI-type is similar to ℕ with just one sort. Note however that this time, the first two constructors have non-recursive arguments and this is reflected in S_4. Also note how the non-recursive arguments are passed in the definition of the constructors `var”` and `abs”`. (We could have used ⊤ ⊎ ⊤ instead of `Bool` in P_4, they are in fact isomorphic as they are both types with two elements.)

```
I₄ = ⊤

S₄ : I₄ → Set
S₄ tt = String ⊎ String ⊎ ⊤

P₄ : (i : I₄) → S₄ i → I₄ → Set
P₄ tt (inj₁ s) tt = ⊥
P₄ tt (inj₂ (inj₁ s)) tt = ⊤
P₄ tt (inj₂ (inj₂ tt)) tt = Bool

var'' : String → WI I₄ S₄ P₄ tt
var'' s = sup tt (inj₁ s) λ {tt → λ ()}

abs'' : String → WI I₄ S₄ P₄ tt → WI I₄ S₄ P₄ tt
abs'' s l = sup tt (inj₂ (inj₁ s)) λ {tt → λ {tt → l}}
```

```
app'' : WI I₄ S₄ P₄ tt → WI I₄ S₄ P₄ tt → WI I₄ S₄ P₄ tt
app'' m n  = sup tt (inj₂ (inj₂ tt)) λ {tt → λ {true → m ; false → n}}
```

Our last example is NF-NE expressed as a WI-type. This type now has two sorts, hence I_9 = Bool. Our convention is that the false value represents NF while true represents NE. The rest of the definitions follow similarly to before, except this time we always have to consider two cases for elements of type I_9.

```
I₉ = Bool

S₉ : I₉ → Set
S₉ false = ⊤ ⊎ String -- NF
S₉ true = String ⊎ ⊤ -- NE

P₉ : (i : I₉) → S₉ i → I₉ → Set
P₉ false (inj₁ tt) false = ⊥ -- ne has 0 NF recursive args
P₉ false (inj₁ tt) true = ⊤ -- ne has 1 NE recursive arg
P₉ false (inj₂ s) false = ⊤ -- lam has 1 NF recursive arg
P₉ false (inj₂ s) true = ⊥ -- lam has 0 NE recursive args
P₉ true (inj₁ s) false = ⊥ -- var has 0 NF recursive args
P₉ true (inj₁ s) true = ⊥ -- var has 0 NE recursive args
P₉ true (inj₂ tt) false = ⊤ -- app has 1 NF recursive arg
P₉ true (inj₂ tt) true = ⊤ -- app has 1 NE recursive arg

ne' : WI I₉ S₉ P₉ true → WI I₉ S₉ P₉ false
ne' e = sup false (inj₁ tt) λ {true → λ {tt → e} ; false → λ ()}

lam' : String → WI I₉ S₉ P₉ false → WI I₉ S₉ P₉ false
lam' s f = sup false (inj₂ s) λ {true → λ () ; false → λ {tt → f}}

var'NE : String → WI I₉ S₉ P₉ true
var'NE s = sup true (inj₁ s) (λ {true → λ () ; false → λ ()})

app'NE : WI I₉ S₉ P₉ true → WI I₉ S₉ P₉ false → WI I₉ S₉ P₉ true
app'NE e f = sup true (inj₂ tt) λ {true → λ {tt → e} ; false → λ {tt → f}}
```

6.2 The Carriers

Given an inductive type's signature, our current aim is to construct an algebra for that signature using its representation as a WI-type. Recall that an algebra `Alg` has two fields: carriers and constructors. We set the carriers of the algebra to be `WI I S P`, where `I`, `S`, and `P` have to be defined in general for every signature. This section details our definition of these three types, forming the carriers of our algebra.

The first type `I : Set` is straightforward to define. Since `I` represents the number of sorts of an inductive type, we simply define `I` as `Fin (sorts S)` for a signature `S`, the type with `sorts S` elements. We can then assign each sort of the inductive type to an element of this type.

The second type `S : I → Set` is passed a sort number, looks at that sort's constructors, and forms a sum type of these constructors, representing the different paths we could take to construct an element of the sort. The sum type also encodes the types of non-recursive arguments to be passed to the constructors. To look at a signature `S`'s constructors for a given sort `i`, we use `cns S i`.

```
makeS : (S : Sig) → Fin (sorts S) → Set
makeS S i = listConToSetNrec S (cns S i)
```

We pass `cns S i` to the function `listConToSetNrec`, which returns the sum type we just described. The case for the empty list, which is reached either immediately when dealing with the empty type (because it has no constructors), or else at the end of the list of constructors for a particular sort, returns the empty type ⊥ to signal there are no other ways to construct the type.

```
listConToSetNrec : (S : Sig) → List (Con (sorts S)) → Set
listConToSetNrec S [] = ⊥
listConToSetNrec S (x ∷ xs) = conToSetNrec S x ⊎ listConToSetNrec S xs
```

`listConToSetNrec` in turn calls the functions `conToSetNrec` and `argToSetNrec`, which handle individual constructors and arguments respectively. The base case for `conToSetNrec` returns ⊤. This case is reached either when a constructor has no arguments, in which case ⊤ signifies an option to construct the sort without having to provide any non-recursive arguments, or at the end of a constructor's list of arguments. In this case, we attach a ⊤ at the end of the type, for which we can easily provide the element `tt`. We could have avoided adding ⊤ at the end by further pattern matching on `xs`, but this would have decreased modularity and made our later functions and proofs that rely on these definitions more complex. `argToSetNrec` returns the argument's type for non-recursive arguments, and a ⊤ for recursive ones.

```
argToSetNrec : (S : Sig) → Arg (sorts S) → Set
argToSetNrec S (nrec t) = El t
argToSetNrec S (rec lst fin) = ⊤

conToSetNrec : (S : Sig) → Con (sorts S) → Set
conToSetNrec S (cn []) = ⊤
conToSetNrec S (cn (x :: xs)) = argToSetNrec S x × conToSetNrec S (cn xs)
```

We can now compare the `S` types we defined for our examples earlier with the `S` types generated by `makeS`. For ℕ, `S` was defined as ⊤ ⊎ ⊤, while `makeS ℕSig zero` evaluates to ⊤ ⊎ ⊤ × ⊤ ⊎ ⊥. The extra ⊤s in the latter is due to the base case of `conToSetNrec` which we explained above. For `Lam`, `S` was defined in the examples as `String` ⊎ `String` ⊎ ⊤, and `makeS LamSig zero` gives `String` × ⊤ ⊎ `String` × ⊤ × ⊤ ⊎ ⊤ × ⊤ × ⊤ ⊎ ⊥. `NF-NE`'s `S` for `NF` was defined as ⊤ ⊎ `String`, and `makeS NFNESig zero` gives ⊤ × ⊤ ⊎ `String` × ⊤ × ⊤ ⊎ ⊥, while for `NE` it was defined as `String` ⊎ ⊤ and `makeS NFNESig (suc zero)` gives `String` × ⊤ ⊎ ⊤ × ⊤ × ⊤ ⊎ ⊥. Once again, the functions could have easily been defined to be closer to the previous `S` values, i.e. having no extra ⊤s and just an extra ⊎ ⊥ at the end, but we chose this representation to facilitate our later definitions and to increase modularity.

The third and last type we need to define is `P : (i : I) → S i → I → Set`. `P` takes the sort `i` we are constructing, a constructor of that sort, and a sort `j`, and returns a type for the recursive arguments of type `j` that are needed for this constructor. To construct this type, we once again have to access the list of constructors of sort `i` in the signature `S` using `cns S i`.

```
makeP : (S : Sig) (i : Fin (sorts S)) → makeS S i → Fin (sorts S) → Set
makeP S i s j = listConToSetRec S (cns S i) s j
```

`makeP` calls the function `listConToSetRec`. Since we are not passing a constructor of type `Con (sorts S)` for signature `S`, but we need the constructor of this type to be able to return its recursive argument types, this function has to figure out which constructor we are referring to using only the `S i` value. We can do this by noting that when our `S i` value is of the form $\texttt{inj}_1$ `x`, we are referring to the first constructor in the `cns S i` list, while if it is of the form $\texttt{inj}_2$ `x`, we recursively have to check the tail of the list and `x`. This is due to how we defined the inductive case of `listConToSetNrec`. For instance, consider the type `Lam` with the three constructors `var : String → Lam`, `abs : String → Lam → Lam`, and `app : Lam → Lam → Lam`, represented in that order in `cns LamSig zero`. Because we constructed `S` based on `cns LamSig zero` so that `makeS LamSig zero = String × ⊤ ⊎ String × ⊤ × ⊤ ⊎ ⊤ × ⊤ × ⊤ ⊎ ⊥`, having an `S i` value of type $\texttt{inj}_1$ `("x" , tt)` means we are constructing `var`, having $\texttt{inj}_2$ `(`$\texttt{inj}_1$ `("x" , tt , tt))` means we are constructing `abs`, and having $\texttt{inj}_2$ `(`$\texttt{inj}_2$ `(`$\texttt{inj}_1$ `(tt , tt , tt)))` means we are constructing `app`.

```
listConToSetRec : (S : Sig) (xs : List (Con (sorts S)))
                  (xsp : listConToSetNrec S xs) → Fin (sorts S) → Set
listConToSetRec S (x ∷ xs) (inj₁ xp) j = conToSetRec S x j
listConToSetRec S (x ∷ xs) (inj₂ xsp) j = listConToSetRec S xs xsp j
```

The function `conToSetRec` returns the sum type of recursive arguments of a given constructor, having the type of a given sort. This is why in the auxiliary function `argToSetRec`, we have to check whether the recursive argument we are considering is of type sort `j` or some other sort. For instance, the constructor `ne : NE → NF` in `NF-NE` has a recursive argument of type `NE` but no recursive arguments of type `NF`, so we need to ensure that this recursive argument is not considered when enquiring about the recursive arguments of type `NF`.

```
recArg : List U → Set
recArg [] = ⊤
recArg (x ∷ xs) = El x × recArg xs


argToSetRec : (S : Sig) → Arg (sorts S) → Fin (sorts S) → Set
argToSetRec S (nrec x) j = ⊥
argToSetRec S (rec lst fin) j with fin =Fin j
argToSetRec S (rec lst fin) j | false = ⊥
```

```
argToSetRec S (rec lst fin) j | true = recArg lst

conToSetRec : (S : Sig) → Con (sorts S) → Fin (sorts S) → Set
conToSetRec S (cn []) j = ⊥
conToSetRec S (cn (x :: xs)) j = argToSetRec S x j ⊎ conToSetRec S (cn xs) j
```

The definition of the predicate =Fin is shown below.

```
_=Fin_ : {n : ℕ} → Fin n → Fin n → Bool
zero =Fin zero = true
zero =Fin suc _ = false
suc _ =Fin zero = false
suc m =Fin suc n = m =Fin n
```

We can once again compare the P types we defined for our examples with the P types generated by makeP. For LamSig, we have the examples below.

Examples	Using makeP
P_4 tt (inj₁ s) tt = ⊥	makeP LamSig zero (inj₁ ("x" , tt)) zero ⟹ ⊥ ⊎ ⊥
P_4 tt (inj₂ (inj₁ s)) tt = ⊤	makeP LamSig zero (inj₂ (inj₁ ("x" , tt , tt))) zero ⟹ ⊥ ⊎ ⊤ ⊎ ⊥
P_4 tt (inj₂ (inj₂ tt)) tt = Bool	makeP LamSig zero (inj₂ (inj₂ (inj₁ (tt , tt , tt)))) zero ⟹ ⊤ ⊎ ⊤ ⊎ ⊥

To check that we are calculating the recursive arguments correctly when having more than one sort, in our examples we have that P_9 false (inj₁ tt) false = ⊥, expressing that ne has no NF recursive arguments, and P_9 false (inj₁ tt) true = ⊤, expressing that ne has one NE recursive argument. Correspondingly, we have that makeP NFNESig zero (inj₁ (tt , tt)) zero gives us ⊥ ⊎ ⊥ (NF recursive arguments), and that makeP NFNESig zero (inj₁ (tt , tt)) (suc zero) gives ⊤ ⊎ ⊥ (NE recursive arguments).

6.3 The Constructors

Having defined makeS and makeP, we can define the carriers of our algebra using WI defined earlier. In this section, we define the function makeConsW that builds the constructors of these carriers, so that we can define our algebra

WAlg as follows.

```
WAlg : (S : Sig) → Alg S
WAlg S  = record { carriers = WI (Fin (sorts S)) (makeS S) (makeP S) ;
                   cons = λ srt c p → makeConsW S srt c p }
```

makeConsW takes a constructor and passes it on to makeConsWAux, much in the same way makeCons passes the constructor to makeConsAux in Section 5.3. Indeed, makeConsWAux is similar to makeConsAux in taking two lists of arguments l_1 and l_2, where l_1 is the part of the constructor it has already processed and l_2 is the part yet to be processed. However, because WI is constructed differently to ⟦_⟧_ S in that non-recursive and recursive arguments are stored separately, we cannot simply use args srt (⟦_⟧_ S) (cn l_1) as the type for our arguments. Instead, we know the non-recursive arguments have type conToSetNrec S (cn l_1), and the recursive arguments are stored in the function (j : Fin (sorts S)) → conToSetRec S (cn l_1) j → WI (Fin (sorts S)) (makeS S) (makeP S) j.

```
makeConsWAux : (S : Sig) (srt : Fin (sorts S)) (l₁ l₂ : List (Arg (sorts S)))
               → (p : (cn (l₁ ++ l₂) ∈ cns S srt))
               → conToSetNrec S (cn l₁)
               → ((j : Fin (sorts S)) → conToSetRec S (cn l₁) j →
               WI (Fin (sorts S)) (makeS S) (makeP S) j) →
               conType srt (WI (Fin (sorts S)) (makeS S) (makeP S))
               (cn l₂)
makeConsWAux S srt l₁ [] p nrec_ars rec_ars =
    sup srt (makeSi S srt l₁ (cns S srt) p nrec_ars)
    (makeWI S srt l₁ p nrec_ars rec_ars)
makeConsWAux S srt l₁ (x ∷ xs) p nrec_ars rec_ars =
    λ ar → makeConsWAux S srt (l₁ ∷ʳ x) xs p
    (conToSetNrecSnoc S srt l₁ x nrec_ars ar)
    (conToSetRecSnoc S srt l₁ x rec_ars ar)


makeConsW : (S : Sig) (srt : Fin (sorts S)) (c : Con (sorts S)) → c ∈ cns S srt
            → conType srt (WI (Fin (sorts S)) (makeS S) (makeP S)) c
makeConsW S srt (cn x) p = makeConsWAux S srt [] x p tt (λ j → λ ())
```

Once makeConsWAux has parsed the entire constructor and l_2 = [], makeSi is used to produce the non-recursive arguments, while `makeWI` is used for the recursive ones. `makeSi` is fairly straightforward and uses the $\texttt{inj}_1/\texttt{inj}_2$ reasoning we discussed when constructing `listConToSetRec`.

```
module _(S : Sig) (srt : Fin (sorts S)) (l : List (Arg (sorts S))) where


    makeSi : (xs : List (Con (sorts S))) → cn l ∈ xs →
              conToSetNrec S (cn l) → listConToSetNrec S xs
    makeSi (.(cn l) :: xs) hd nrec_ars = inj₁ nrec_ars
    makeSi (x :: xs) (tl p) nrec_ars = inj₂ (makeSi xs p nrec_ars)
```

`makeWI` transforms the function we have of type

```
(j : Fin (sorts S)) → conToSetRec S (cn l₁) j → WI (Fin (sorts S))
(makeS S) (makeP S) j
```

into one which fits the type we need, which is

```
(j : Fin (sorts S)) → makeP S srt (makeSi (cns S srt) p nrec_ars) j
→ WI (Fin (sorts S)) (makeS S) (makeP S) j.
```

It builds the required function from the given one by pattern matching on the location of the fully processed constructor $\texttt{cn l}_1$ in `cns S srt`, because it depends on the `makeSi` value which also pattern matches on this location. Its definition is omitted here as it is fairly mechanical, but can be found in the supplementary material.

The function `conToSetNrecSnoc` updates the non-recursive arguments at each step of `makeConsWAux`. It produces an element of type `conToSetNrec S (cn (l ::`^{r}` x))`. It checks if the snocced argument `x` is recursive or not. If it is non-recursive, it adds the incoming argument `ar` at the end of the tuple of non-recursive arguments. If it is recursive, it adds a `tt` at the end of the tuple instead.

`conToSetRecSnoc` rewrites the function of type

```
(j : Fin (sorts S)) → conToSetRec S (cn l) j → WI (Fin (sorts S))
(makeS S) (makeP S) j)
```

into a function of type

```
(j : Fin (sorts S)) → conToSetRec S (cn (l ::ʳ x)) j → WI (Fin (sorts S))
(makeS S) (makeP S) j,
```

to accomodate for the snocced argument x. It does so by first checking whether x is recursive or not. If it is non-recursive, the process is straightforward as we have no recursive arguments to add. If it is recursive of the form `rec lst fin`, the process is more complex as we have to leave the entries in the above function for $j \neq$ `fin` unchanged, and associate the new recursive argument `ar` to the relevant sort `j`.

The code for `conToSetNrecSnoc` and `conToSetRecSnoc` can be found in the supplementary material. To illustrate how the functions we mentioned in this section work together, we show the steps in generating a constructor for `lam : Str → NF → NF` of the inductive type `NF-NE`.

```
    makeConsW NFNESig zero (cn (nrec string :: rec [] zero ::
    [])) (tl hd)
⟹ makeConsWAux NFNESig zero [] (nrec string :: rec [] zero ::
    []) (tl hd) tt (λ j → λ ())
⟹ λ s → makeConsWAux NFNESig zero (nrec string :: [])
    (rec [] zero :: []) (tl hd) (s , tt)
    (λ j → λ {(inj₁ ()); (inj₂ ())})
⟹ λ s → λ f → makeConsWAux NFNESig zero (nrec string ::
    rec [] zero :: []) [] (tl hd) (s , tt , tt)
    (λ {zero → {λ {(inj₂ (inj₁ tt)) → f} ;
    (suc zero) → λ {(inj₁ ()) ; (inj₂ (inj₁ ())) ;
    (inj₂ (inj₂ ()))}})
⟹ sup zero (inj₂ (inj₁ (s , tt , tt)))
    (λ {zero → {λ {(inj₂ (inj₁ tt)) → f} ;
    (suc zero) → λ {(inj₁ ()) ; (inj₂ (inj₁ ())) ;
    (inj₂ (inj₂ ()))}})
```

6.4 The Iterator

Now that we have `WAlg S` for a given signature `S`, the next step in our reduction is to construct the iterator, a morphism from `WAlg S` to any other `S`-algebra.

```
WIt : (S : Sig) (A : Alg S) → Mor S (WAlg S) A
WIt S A = record { f = λ srt → funcsW S A srt ;
                   eq = λ srt c p xs → {!!} }
```

This construction has unfortunately not been completed, but throughout this section we describe our work so far and some ideas for going forward.

The first thing we need to construct is a function `funcsW` from the carriers of `WAlg S` to the carriers of an `S`-algebra `A`.

```
funcsW : (S : Sig) (A : Alg S) (srt : Fin (sorts S)) →
         WI (Fin (sorts S)) (makeS S) (makeP S) srt → carriers A srt
```

Similarly to `funcs` from Section 5.4, what `funcsW` must do is to first convert arguments of `WI (Fin (sorts S)) (makeS S) (makeP S) srt` into arguments of `carriers A srt` by recursively calling `funcsW` on these arguments, and then apply the arguments to the corresponding constructors in `A`. The example below illustrates what we want to achieve, i.e. a generalisation of the functions `fNF”` and `fNE”`.

```
fNF'' : (A : Alg NFNESig) → WI (Fin (sorts NFNESig)) (makeS NFNESig)
        (makeP NFNESig) zero → carriers A zero
fNE'' : (A : Alg NFNESig) → WI (Fin (sorts NFNESig)) (makeS NFNESig)
        (makeP NFNESig) (suc zero) → carriers A (suc zero)


-- ne
fNF'' A (sup .(zero) (inj₁ (tt , tt)) f) =
    (cons A) zero (cn (rec [] (suc zero) ∷ [])) hd
    (fNE'' A (f (suc zero) (inj₁ tt)))
-- lam
fNF'' A (sup .(zero) (inj₂ (inj₁ (s , tt , tt))) f) =
    (cons A) zero (cn (nrec string ∷ rec [] zero ∷ [])) (tl hd) s
    (fNF'' A (f zero (inj₂ (inj₁ tt))))
```

```
-- var
fNE'' A (sup .(suc zero) (inj₁ (s , tt)) f) =
    (cons A) (suc zero) (cn (nrec string :: [])) hd s
-- app
fNE'' A (sup .(suc zero) (inj₂ (inj₁ (tt , tt , tt))) f) =
   (cons A) (suc zero) (cn (rec [] (suc zero) :: rec [] zero :: [])) (tl hd)
   (fNE'' A (f (suc zero) (inj₁ tt) )) (fNF'' A (f (zero) (inj₂ (inj₁ tt))))
```

We note that when calling `fNF”` or `fNE”` recursively, it is not immediately obvious how we can provide the element of type `WI`. So for example, in the first case of `fNF”`, we obtain the `WI` element using `f (suc zero) (inj₁ tt)`. We call `f` with the argument `suc zero` because we need an argument of type `WI (suc zero)` (representing `NE`), and the other argument `inj₁ tt` is obtained by looking at the result of `makeP NFNESig zero (inj₁ (tt , tt)) (suc zero)`, the type for recursive arguments of type `WI (suc zero)` in the first constructor of `WI (zero)`, which evaluates to $\top \uplus \bot$.

`funcsW` is defined similarly to `funcs`, using the previously defined `apply`, but calls some new auxiliary functions.

```
funcsW S A srt (sup .srt s p) =
   apply S A srt c (cons A srt c pf) (makeArgs S A srt (cns S srt) s p)
       where
           c,pf = findCon S (cns S srt) s
           c = proj₁ c,pf
           pf = proj₂ c,pf
```

To call `apply`, we need to find out which constructor `sup srt s p` corresponds to, together with its location in `cns S srt` for signature `S`. We do this using the function `findCon`, which pattern matches on `s` and returns a tuple with the constructor and its location in the list of constructors `cns S srt`.

```
findCon : (S : Sig) (l : List (Con (sorts S))) → listConToSetNrec S l →
          Σ (Con (sorts S)) (λ c → c ∈ l)
findCon S (x :: xs) (inj₁ _) = x , hd
findCon S (x :: xs) (inj₂ y) = proj₁ rest , tl (proj₂ rest)
    where
        rest = findCon S xs y
```

funcsW then calls makeArgs, which goes through cns S srt until it arrives at the constructor that s corresponds to, and then calls argsWToCarr on this constructor. argsWToCarr's job is to then extract the non-recursive and recursive arguments from the s and p in sup srt s p, and transform them into arguments for the S-algebra A.

```
argsWToCarr : (srt : Fin (sorts S)) (c : Con (sorts S)) (s : conToSetNrec S c)
        → ((j : Fin (sorts S)) → conToSetRec S c j → WI (Fin (sorts S))
        (makeS S) (makeP S) j) → args srt (carriers A) c
argsWToCarr srt (cn []) s p = s
argsWToCarr srt (cn (x ∷ xs)) (fst , snd) p =
    argTypeWToCarr x fst (λ j ar → p j (inj₁ ar)) ,
    argsWToCarr srt (cn xs) snd (λ j cr → p j (inj₂ cr))

makeArgs : (srt : Fin (sorts S)) (l : List (Con (sorts S)))
           (s : listConToSetNrec S l) → ((j : Fin (sorts S)) →
           listConToSetRec S l s j → WI (Fin (sorts S)) (makeS S)
           (makeP S) j) → args srt (carriers A) (proj₁ (findCon S l s))
makeArgs srt (x ∷ xs) (inj₁ y) p = argsWToCarr srt x y p
makeArgs srt (x ∷ xs) (inj₂ y) p = makeArgs srt xs y p
```

argsWToCarr above is comparable to argsInitToCarr from Section 5.4. This function calls argTypeWToCarr, which handles individual arguments and is comparable to argTypeInitToCarr, and this in turn calls mapConW, which handles lists of arguments for recursive function arguments and is comparable to mapCon.

```
mapConW : (lst : List U) (fin : Fin (sorts S)) (s : ⊤) →
          (recArg lst → WI (Fin (sorts S)) (makeS S) (makeP S) fin)
          → conTypeAux (carriers A) lst fin
mapConW [] fin s p = funcsW fin (p s)
mapConW (x ∷ xs) fin s p = λ el_x → mapConW xs fin s (λ ra → p (el_x , ra))

argTypeWToCarr : (x : Arg (sorts S)) → argToSetNrec S x → ((j : Fin (sorts S))
                 → argToSetRec S x j → WI (Fin (sorts S)) (makeS S)
                 (makeP S) j) → argType (carriers A) x
argTypeWToCarr (nrec x) s p = s
argTypeWToCarr (rec lst fin) s p = mapConW lst fin s (p fin)
```

Similarly to `mapCon`, `mapConW` calls `funcsW`. This is the recursive call necessary to map arguments of the WI-type to arguments of the carriers of `A`. However, this time this leads to a non-termination error. The structure and sequence of the functions in this section is the same as that in Section 5.4, the only difference being the passing around of a function representing the argument `((j : I) → P i s j → WI I S P j)` of the `sup` constructor of the WI-type. Given a sort number, this function returns the recursive arguments having the type of that sort. Because we are passing around a function, Agda cannot ensure that the arguments are getting structurally smaller.

One way of circumventing this problem is to come up with a data type to replace the function we are passing around. A simple example of this idea is replacing a function of type `{n : ℕ} → Fin n → A`, where `A : Set`, with the data type `Vec A n`. Indeed, these both associate a number between 0 and $n-1$ to an element of type `A`. Our particular case requires a more refined data type as it involves dependent types. One possible option is to use a data type like the following, accompanied by a function for access of data.

```
data HVec : (n : ℕ)(A : Fin n → Set) → Set₁ where
    [] : HVec ℕ.zero (λ ())
    _::_ : {n : ℕ}{A : Fin (ℕ.suc n) → Set} → A zero →
            HVec n (λ i → A (suc i)) → HVec (ℕ.suc n) A


appH : {n : ℕ}{A : Fin n → Set} → HVec n A → (i : Fin n) → A i
appH {ℕ.suc n} {A} (a :: as) zero = a
appH {ℕ.suc n} {A} (a :: as) (suc i) = appH {n} {(λ i → A (suc i))} as i
```

At present, we still have some issues incorporating this representation into our functions, so we leave this for future work. In our supplementary material, we use the `{-# TERMINATING #-}` pragma so Agda switches off termination checking for this code block. This is a 'cheat' that will be fixed in the future, and should not be used under normal circumstances as this gives no guarantees that our code is correct. However, when manually analysing the structure of our functions, it is clear that the non-recursive arguments `s` being passed to `p` in `mapConW` have become smaller than their original value in the initial call of `funcsW`, so our code should in fact terminate even though Agda cannot guarantee this.

7

Conclusion and Future Work

This dissertation contributes towards the formalisation of simple and mutual inductive types left incomplete by more general work. We specified a small 'theory of signatures' in which we can express any simple or mutual inductive type. Given a signature from this theory, we specified what algebras and morphisms are for the signature, constructed its initial algebra, and constructed a unique morphism from this algebra to any other algebra of the signature. Moreover, we looked into WI-types, and constructed a WI-type algebra for any given signature. We then described our attempt at constructing the iterator for this algebra, which unfortunately has not been completed, but goes a fair distance to construct a reduction from simple and mutual inductive types to W-types.

Our work advances the long-term goal of creating a small as possible trusted code base responsible for software verification. On the one hand, this means that the end user has to assume as little as possible to provide behavioural guarantees for their code, and on the other hand it prevents malicious users from taking advantage of the complexities of a large code base. Moreover, studying the metatheory of Martin-Löf Type Theory is essential for its validity as an alternative foundation of mathematics to set theory, and for proofs shown in this setting to be deemed reliable.

Since all of our formalisations took place in Agda, a proof-assistant in which code is type checked not run, the way to evaluate our results is to ensure that our code compiles in Agda. This constitutes a proof of our results thanks to the propositions as types paradigm. All of our code type checks except for the non-termination error in the WI-type iterator functions in Section 6.4, and as we have already discussed there, we aim to solve this issue in future work by replacing the function causing the error by a data type. Some other areas of improvement we also aim to tackle in future work are using Cubical libraries for an alternative equality type, using which we would be able to prove UIP, as well as derive the WI-type and its eliminator from the W-type and its eliminator, and not using pattern matching on WI-types but instead using only the WI-type eliminator. These improvements will further reduce our axioms and trusted code base, taking us closer to our long-term goal.

Alongside the improvements to our existing constructions, a possible direction for future work is the generalisation of our work to more general inductive types, such as inductive families and inductive-inductive types. We already know from Altenkirch and Morris (2009) and Altenkirch et al. (2015) that inductive families are reducible to WI-types and hence to W-types, so this reduction in Agda should be possible. As for inductive-inductive types, some work has been done showing a reduction which supports simpler versions or specific inductive-inductive type examples (Forsberg, 2014; Hugunin, 2019), so this would be more challenging. Another class of types to consider are nested inductive types, which we have only briefly mentioned in this dissertation, but to which we could also extend our constructions.

Bibliography

Abbott, M., Altenkirch, T., and Ghani, N. (2003). Categories of containers. In *Proceedings of Foundations of Software Science and Computation Structures*.

Abbott, M., Altenkirch, T., and Ghani, N. (2005). Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27. Applied Semantics: Selected Topics.

Abbott, M. G. (2003). *Categories of containers*. PhD thesis, University of Leicester, England, UK.

Altenkirch, T., Ghani, N., Hancock, P., McBride, C., and Morris, P. (2015). Indexed containers. *Journal of Functional Programming*, 25:e5.

Altenkirch, T. and Morris, P. (2009). Indexed containers. In *Proceedings - Symposium on Logic in Computer Science*, pages 277–285.

Awodey, S., Gambino, N., and Sojakova, K. (2012). Inductive types in homotopy type theory. In *2012 27th Annual IEEE Symposium on Logic in Computer Science*, pages 95–104. IEEE.

Church, A. (1940). A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, 5(2):56–68.

Dybjer, P. (1994). Inductive families. *Formal Aspects of Computing*, 6:440–465.

Dybjer, P. (1997). Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoretical Computer Science*, 176(1):329–335.

Dybjer, P. (2018). *An Introduction to Programming and Proving in Agda (incomplete draft)*. http://www.cse.chalmers.se/~peterd/papers/AgdaLectureNotes2018.pdf [Accessed: 20-08-2020].

Dybjer, P., Lindström, S., Palmgren, E., and Sundholm, G. (2012). *Epis-*

temology versus ontology. Essays on the philosophy and foundations of mathematics in honour of Per Martin-Löf. Based on the conference, "Philosophy and foundations of mathematics: Epistemological and ontological aspects", Uppsala, Sweden, May 5–8, 2009.

Forsberg, F. N. (2014). Inductive-inductive definitions.

Gambino, N. and Hyland, M. (2003). Wellfounded trees and dependent polynomial functors. volume 3085.

Hofmann, M. (1996). Conservativity of equality reflection over intensional type theory. In Berardi, S. and Coppo, M., editors, *Types for Proofs and Programs*, pages 153–164, Berlin, Heidelberg. Springer Berlin Heidelberg.

Howard, W. (1980). The Formulae-as-Types Notion of Construction.

Hugunin, J. (2019). Constructing inductive-inductive types in cubical type theory. In Bojańczyk, M. and Simpson, A., editors, *Foundations of Software Science and Computation Structures*, pages 295–312, Cham. Springer International Publishing.

Kaposi, A., Kovács, A., and Altenkirch, T. (2019). Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.*, 3(POPL).

Martin-Löf, P. (1971). Hauptsatz for the intuitionistic theory of iterated inductive definitions. In *Studies in Logic and the Foundations of Mathematics*, volume 63, pages 179–216. Elsevier.

Martin-Löf, P. (1972). An intuitionistic theory of types. *Twenty-Five Years of Constructive Type Theory.*

Martin-Löf, P. (1982). Constructive mathematics and computer programming. In *Studies in Logic and the Foundations of Mathematics*, volume 104, pages 153–175. Elsevier.

Martin-Löf, P. and Sambin, G. (1984). *Intuitionistic type theory*, volume 9. Bibliopolis Naples.

Milewski, B. (2018). *Category Theory for Programmers.* Blurb, Incorporated.

Norell, U. and Chapman, J. (2009). Dependently typed programming in Agda. In *Advanced Functional Programming, volume 5832 of LNCS*, pages 230–266. Springer.

Russell, B. (1903). *Principles of Mathematics*. University Press, Cambridge. Appendix B: The Doctrine of Types.

The Agda Team (2020). Agda's documentation. https://agda.readthedocs.io/en/v2.6.1/ [Accessed: 20-08-2020].

The Univalent Foundations Program (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

Troelstra, A. (1991). History of constructivism in the twentieth century.

Wadler, P., Kokke, W., and Siek, J. G. (2020). *Programming Language Foundations in Agda*. Available at http://plfa.inf.ed.ac.uk/20.07/.

Index