Portland State University

PDXScholar

Dissertations and Theses Dissertations and Theses

Spring 6-6-2017

Fully Generic Programming Over Closed Universes of Inductive-Recursive Types

Larry Diehl
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Computer Sciences Commons

Let us know how access to this document benefits you.

Recommended Citation

Diehl, Larry, "Fully Generic Programming Over Closed Universes of Inductive-Recursive Types" (2017). *Dissertations and Theses.* Paper 3647.
https://doi.org/10.15760/etd.5531

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Fully Generic Programming

Over Closed Universes of Inductive-Recursive Types

by

Larry Diehl

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Tim Sheard, Chair
James Hook
Mark P. Jones
Andrew Tolmach
Robert Bass

Portland State University
2017

ABSTRACT

Dependently typed programming languages allow the type system to express arbitrary propositions of intuitionistic logic, thanks to the Curry-Howard isomorphism. Taking full advantage of this type system requires defining more types than usual, in order to encode logical correctness criteria into the definitions of datatypes. While an abundance of specialized types helps ensure correctness, it comes at the cost of needing to redefine common functions for each specialized type.

This dissertation makes an effort to attack the problem of code reuse in dependently typed languages. Our solution is to write generic functions, which can be applied to any datatype. Such a generic function can be applied to datatypes that are defined at the time the generic function was written, but they can also be applied to any datatype that is defined in the *future*. Our solution builds upon previous work on generic programming within dependently typed programming.

Type theory supports generic programming using a construction known as a *universe*. A universe can be considered the model of a programming language, such that writing functions over it models writing generic programs in the programming language. Historically, there has been a trade-off between the expressive power of the modeled programming language, and the kinds of generic functions that can be written in it. Our dissertation shows that no such trade-off is necessary, and that we can write future-proof generic functions in a model of a dependently typed programming language with a rich collection of types.

ACKNOWLEDGMENTS

I would like to thank Tim Sheard, my advisor, for giving me the freedom to explore my own research interests, for always being available to listen and provide constructive feedback, and for instilling in me the importance of thoroughly explaining background material, supplemented by plenty of examples.

I would also like to thank my parents, for supporting my decision to pursue academic interests, despite needing to abandon a lucrative job and career in software development.

Finally, I would like to thank Conor McBride, for inspiring me to work on the topic of generic programming. This inspiration is in part due to his academic publications and artifacts resulting from the Epigram programme, and is in part due to him warmly and enthusiastically welcoming a naive industry programmer.

TABLE OF CONTENTS

Abstract i

Acknowledgments ii

List of Figures ix

Color Conventions x

Part I Prelude 1

Chapter 1 Introduction 2

1.1 Dependently Typed Languages & Motivation . . . 3
1.1.1 Curry-Howard Isomorphism . . . 3
1.1.2 Indexed Types . . . 5
1.1.3 Motivation . . . 6
1.2 A Taste of Fully Generic Programming . . . 6
1.2.1 Traditional Generic Programming . . . 7
1.2.2 Fully Generic Programming . . . 9
1.2.3 Universes . . . 11
1.2.4 Fully Generic versus Deriving . . . 14
1.3 Class of Supported Datatypes . . . 14
1.3.1 Dependent Algebraic Types . . . 14
1.3.2 Indexing versus Induction-Recursion . . . 15
1.3.3 Smallness versus Largeness . . . 16
1.4 Thesis . . . 18
1.4.1 Thesis Statement . . . 18
1.4.2 Contributions . . . 19
1.4.3 Outline . . . 19

Chapter 2 Types & Universes **24**
2.1 Types . . . 25
2.1.1 Function Types . . . 25
2.1.2 Non-Inductive Types . . . 26
2.1.3 Inductive Types . . . 27
2.1.4 Parameterized Types . . . 27
2.1.5 Indexed Types . . . 28
2.1.6 Type Families . . . 31
2.1.7 Derived Types . . . 31
2.1.8 Infinitary Types . . . 34
2.1.9 Inductive-Recursive Types . . . 35
2.1.10 Algebraic Types . . . 37
2.1.11 Computational Families . . . 38
2.1.12 Open Types . . . 39
2.1.13 Closed Types . . . 39
2.2 Universes . . . 40
2.2.1 Universe Model . . . 40
2.2.2 Open Universes . . . 41
2.2.3 Closed Universes . . . 43
2.2.4 Inductive Universes . . . 44
2.2.5 Non-Inductive Universes . . . 44
2.2.6 Subordinate Universes . . . 45
2.2.7 Autonomous Universes . . . 45
2.2.8 Derived Universes . . . 46
2.2.9 Parameterized Universes . . . 47

Chapter 3 Generic Programming **50**
3.1 Parametric Polymorphism . . . 51
3.1.1 Parametric over Types . . . 51
3.1.2 Parametric over Levels . . . 51
3.2 Ad Hoc Polymorphism . . . 52
3.2.1 Ad Hoc by Overloading . . . 52
3.2.2 Ad Hoc by Coercion . . . 53
3.2.3 Ad Hoc by Overloading & Coercion . . . 54
3.3 Abstractness & Concreteness . . . 54
3.3.1 Abstract Types . . . 55

3.3.2 Concrete Types . . . 55
3.3.3 Abstract Data Types . . . 56
3.3.4 Fully Generic Programming . . . 56
3.4 Totality . . . 57
3.4.1 Non-Dependent Domain Change . . . 57
3.4.2 Non-Dependent Codomain Change . . . 58
3.4.3 Dependent Domain Change . . . 58
3.4.4 Dependent Codomain Change . . . 59
3.4.5 Domain Predicates versus Domain Supplements . . . 60

Chapter 4 Closed Type Theory **62**
4.1 Closed Vector Universe . . . 63
4.1.1 Closed Vector Types . . . 63
4.1.2 Fully Generic Functions . . . 65
4.2 Closed Algebraic Universe . . . 69
4.2.1 Closed Well-Order Types . . . 70
4.2.2 Open Well-Order Types . . . 71
4.2.3 Inadequacy of Well-Orders . . . 73

Part II Open Type Theory **76**

Chapter 5 Open Algebraic Universes **77**
5.1 Open Non-Dependent Types . . . 79
5.1.1 Categorical Model . . . 79
5.1.2 Formal Model . . . 81
5.1.3 Examples . . . 86
5.2 Open Infinitary Types . . . 91
5.2.1 Categorical Model . . . 91
5.2.2 Formal Model . . . 93
5.2.3 Examples . . . 96
5.3 Open Dependent Types . . . 99
5.3.1 Categorical Model . . . 99
5.3.2 Formal Model . . . 102
5.3.3 Examples . . . 106
5.4 Open Inductive-Recursive Types . . . 111
5.4.1 Categorical Model . . . 112

5.4.2 Formal Model . . . 117
5.4.3 Examples . . . 122
5.4.4 Agda Model . . . 131

Part III Closed Type Theory **133**

Chapter 6 Closed Algebraic Universe **134**
6.1 Open Inductive-Recursive Types . . . 135
6.1.1 Formal Model . . . 136
6.1.2 Source of Openness . . . 137
6.2 Closed Inductive-Recursive Types . . . 139
6.2.1 Formal Model . . . 139
6.2.2 Examples . . . 142
6.2.3 Kind-Generalized Universes . . . 150
6.3 How to Close a Universe . . . 152
6.3.1 Procedure . . . 152
6.3.2 Example Procedure Run . . . 153
6.4 Types versus Kinds . . . 157
6.4.1 Open Types and Kinds . . . 157
6.4.2 Gratuitous Kinds . . . 160
6.4.3 Types versus Descriptions . . . 161
6.4.4 Kind-Parameterized Types . . . 162

Chapter 7 Fully Generic Functions **167**
7.1 Fully Generic Count . . . 169
7.1.1 Generic Types . . . 169
7.1.2 Counting All Values . . . 172
7.1.3 Counting Algebraic Arguments . . . 173
7.1.4 Examples . . . 176
7.2 Fully Generic Lookup . . . 185
7.2.1 Generic Types . . . 186
7.2.2 Looking Up All Values . . . 188
7.2.3 Looking Up Algebraic Arguments . . . 190
7.2.4 Splitting Functions . . . 193
7.2.5 Examples . . . 194
7.3 Fully Generic AST . . . 197

7.3.1 Generic Types 199
7.3.2 Marshalling Initial Algebras 200
7.3.3 Marshalling All Values 201
7.3.4 Marshalling Algebraic Arguments 202
7.3.5 Generic Template 204

Chapter 8 Closed Hierarchy of Universes 206

8.1 Closed Hierarchy of Well-Order Types 209
8.1.1 Formal Model 209
8.1.2 Examples 213
8.1.3 Agda Model 215
8.2 Closed Hierarchy of Inductive-Recursive Types 220
8.2.1 Agda Model 220
8.2.2 Examples 228
8.3 Leveled Fully Generic Functions 235
8.3.1 Counting in Universe Zero 236
8.3.2 Counting in Universe One 238
8.3.3 Leveled Generic Template 241

Part IV Postlude 243

Chapter 9 Related Work 244

9.1 Fixed Open or Closed Universes 244
9.2 Extendable Open or Closed Well-Order Universes 246
9.3 Extendable Open Algebraic Universes 247
9.4 Previous Work 250

Chapter 10 Future Work 253

10.1 Universe Polymorphism 253
10.2 Large Induction-Recursion 254
10.3 Induction-Induction 255
10.4 High-Level Generic Programming 255
10.5 Efficient Implementation 256
10.6 Termination of Intensional Closed Type Theory 256
10.7 Inductive Definitions over Infinitary Domain 257

Chapter 11 Conclusion 258

References		**260**
Appendices		**267**
Appendix A	**Open Non-Algebraic Types**	**267**
Appendix B	**Open Universe of Algebraic Types**	**268**
Appendix C	**Closed Universe of Algebraic Types**	**269**
Appendix D	**Closed Hierarchy of Universes**	**270**
Appendix E	**Internalized Constructor Signatures**	**273**

LIST OF FIGURES

7.1 The natural number 1, as a closed algebraic type. 178
7.2 The natural number 2, as a closed algebraic type. 179
7.3 The length-1 vector of pairs of strings `[("a", "x")]`, as a closed algebraic type. 181
7.4 The length-2 vector of pairs of strings `[("a", "x"), ("b", "y")]`, as a closed algebraic type. 184
7.5 Definitions of the helper splitting functions (splitΣ, splitσ, and splitδ) used in Section 7.2.2 and Section 7.2.3. The helpers are all just shallow wrappers around the splitFin function (Section 7.2.4). 195
7.6 The inductive-recursive component of the length-1 vector of pairs of strings `[("b", "y")]`, as a closed algebraic type. This figure depicts the inductive-recursive first component of the vector encoded as a dependent pair (the second component is the length constraint). . . . 197

8.1 Closed natural number definitions in universe level 0. 236
8.2 Fully generic counting of values (count) and algebraic arguments (counts) in universe level 0. 237
8.3 Fully generic counting of values (Count) and algebraic arguments (Counts) in universe level 1. 239
8.4 Fully generic counting of types (CountSet) and algebraic arguments (CountDesc) in universe level 1. 240

COLOR CONVENTIONS

This dissertation can be read in black and white, but it benefits from being read in color. The main programming language used in this dissertation is Agda, which is dependently typed. Agda does not use any syntactic conventions, like capitalization, to distinguish identifiers of various program elements, like datatypes, definitions, and constructors (this is partially due to the fact most program elements can be legally used at both the *type* and *value* levels of the dependently typed language).

Knowing which program element an identifier stands for depends on the *environment*. In other words, readers of the black and white version of the dissertation can check previous definitions to see what an identifier was declared as, in order to understand a particular piece of code. Readers of the colored version of the dissertation can understand a piece of code by being aware of color conventions (described below), without needing to consult previous definitions of identifiers. The Agda system keeps track of what identifiers were declared as in the environment, allowing the appropriate color to be emitted when displaying syntax highlighted code (because the *syntax* highlighting depends on the environment, it may make sense to think of the output of Agda as *semantics* highlighted code).

We use the following Agda source code highlighting color conventions: Keywords are orange, `comments` are red (and prefixed by a dash), `strings` are red (and enclosed in quotes), datatypes are dark blue, definitions are light blue, constructors are green, record projections are pink, and *variables* are purple.

Part I

Prelude

Chapter 1

INTRODUCTION

In this dissertation we expand the class of functions that can be written generically for all types, in a type-safe manner, within a dependently typed language [3, 39]. Below, we contrast traditional generic programming [5, 32] with the approach we describe in this thesis, which we call *fully generic programming*.

Traditional Generic Programming Traditional generic programming captures the pattern of folding an algebra through the *inductive* occurrences of an algebraic datatype. For example, we could write a generic size function, that can be applied to *any* datatype. For any constructor of any datatype, size returns 1 plus the sum of:

- The number of non-inductive arguments.
- The *recursive* size of the inductive arguments.

Applying size to a single-element list containing a pair of booleans (`(True, False):[]`) results in 3: the sum of the cons constructor, the pair, and the nil constructor. Because the pair is a non-inductive argument from the perspective of the list, its size is counted atomically as 1 (it would be counted as 1 even if it were a value of an *inductive* datatype other than lists, like a tree, since it is not inductive with respect to lists).

Fully Generic Programming Fully generic programming (Section 3.3.4) captures the pattern of folding an algebra through both the *non-inductive* and *inductive* occurrences of an algebraic datatype. For example, we could write a fully

generic count function that can also be applied to *any* datatype. For any constructor of any datatype, count returns 1 plus the sum of:

- The *recursive* count of the non-inductive arguments.
- The *recursive* count of the inductive arguments.

Applying count to a single-element list containing a pair of booleans (`(True, False):[]`) results in 5: the sum of the cons constructor, the pair (,) constructor, the two booleans constructors, and the nil constructor. Notably, count (unlike size) additionally recurses through the components of the pair.

In the remainder of this introduction we provide an overview of dependently typed languages and motivate our work (Section 1.1), give an example of fully generic programming over a limited collection of datatypes (Section 1.2), describe the class of datatypes that we have been able to extend the fully generic programming approach to (Section 1.3), and finally cover our thesis statement and contributions (Section 1.4).

1.1 DEPENDENTLY TYPED LANGUAGES & MOTIVATION

A standard dependently typed language [38, 45] is *purely functional* (meaning an absence of side effects), *total* (meaning all inductively defined functions terminate and cover all possible inputs), and has a type system that captures the notion of *dependency*. In this thesis we use the dependently typed language Agda [47] for all of our developments.[1]

1.1.1 Curry-Howard Isomorphism

A single term language is used to write programs at the value and type levels. The combination of total programming at the type level and a notion of dependency

[1] This dissertation is written as a literate Adga program. The literate Agda source file and accompanying code can be found at: `https://github.com/larrytheliquid/thesis`

between types allows any proposition of intuitionistic logic to be expressed as a type. A value (or equivalently, a total program) inhabiting a type encoding a proposition serves as its intuitionistic proof. This correspondence between values and types, and proofs and propositions, is known as the *Curry-Howard isomorphism* [33]. For example, below we compare universally quantified propositions to dependent function types, and existentially quantified propositions to dependent pair types.

$$\forall x.\ \mathrm{P}(x) \approx (x : A) \to \mathsf{P}\ x$$

$$\exists x.\ \mathrm{Q}(x) \approx \Sigma\ A\ (\lambda\ x \to \mathsf{Q}\ x)$$

Using the Curry-Howard isomorphism, we can encode logical *preconditions* and *postconditions* at the type level. For example, below we give the type of a lookup function over lists with a *precondition* constraining the natural number (n) index to be less than the length of the list (xs) being looked up. This allows an otherwise partial lookup function to be defined totally by preventing out-of-bounds indexing.

$$\mathsf{lookup} : (n : \mathbb{N})\ (xs : \mathsf{List}\ A) \to n < \mathsf{length}\ xs \to A$$

As another example, we give the type of an append function over lists with a *postcondition* constraining the length of the output list (zs) to be equal to the sum of the lengths of the input lists (xs and ys).

$$\mathsf{append} : (xs\ ys : \mathsf{List}\ A) \to \Sigma\ (\mathsf{List}\ A)\ (\lambda\ zs \to \mathsf{length}\ zs \equiv \mathsf{length}\ xs + \mathsf{length}\ ys)$$

The types of lookup and append correspond to the following two logical propositions respectively.

$$\forall n, xs.\ n < |xs| \Rightarrow \exists x.\ \top$$

$$\forall xs, ys.\ \exists zs.\ \ |zs| = |xs| + |ys|$$

1.1.2 Indexed Types

The less-than ($<$) precondition and equality ($\equiv$) postcondition in the examples above are *relations* in the language of logic, and are called *indexed types* [19, 20] in the language of dependent types. Indexed types are (commonly) types whose arguments are values (rather than types). For example, less-than ($<$) takes two natural number ($\mathbb{N}$) arguments, and equality ($\equiv$) takes two values of some type A. Rather than constraining a datatype (like lists) using relations after-the-fact, we can create more specific (i.e., *indexed*) variants of datatypes that encode certain properties before-the-fact.

For example, the type of vectors ($\mathsf{Vec}\ A\ n$) is like a length-indexed version of lists. Compared to lists, the type former of vectors gains an additional natural number parameter (n) constraining its length. Because the property of the length of a vector is encoded at the type level, we can write a variant of append where calls to length have been replaced by an index.

$$\mathsf{append} : (m\ n : \mathbb{N})\ (xs : \mathsf{Vec}\ A\ m)\ (ys : \mathsf{Vec}\ A\ n) \to \mathsf{Vec}\ A\ (m + n)$$

Additionally, the explicit equality proof ($\equiv$) postcondition can be dropped in favor of expressing the postcondition directly in the index position of the output vector. In other words, the *extrinsic* equality postcondition has been dropped in favor of an *intrinsic* property about the codomain of append.

Another example of an indexed type is the type of finite sets ($\mathsf{Fin}\ n$), indexed by a natural number constraining the size of the finite set. A finite set is like a subset of the natural numbers from 0 to n - 1. This subset property (whose maximum value is n - 1) is the perfect datatype to act as an *intrinsic* version of the *extrinsic* less-than ($<$) precondition of lookup. Hence, we can rewrite an intrinsic-precondition version of lookup using vectors and finite sets as follows.

$$\mathsf{lookup} : (n : \mathbb{N})\ (xs : \mathsf{Vec}\ A\ n)\ (i : \mathsf{Fin}\ n) \to A$$

1.1.3 Motivation

Programmers of non-dependently typed languages already struggle with the issue of needing to define logically similar versions of functions (like count, lookup, etc.) for their various algebraic types (e.g., natural numbers, lists, binary trees, etc.). This problem is more pronounced in a dependently typed language, where programmers also define indexed variants of types (e.g., finite sets, vectors, balanced binary trees, etc.) that intrinsically capture preconditions and postconditions.

Rather than punishing programmers for creating new datatypes, our **motivation** is to reward them with *fully generic functions* (like count, lookup, etc.), which are new mechanisms for *code reuse*. Fully generic functions are predefined once-and-for-all to work with any datatype of the language, whether it is defined now or will be defined in the future. Programmers defining new types should be able to *apply* fully generic functions to them, and programmers should also be able to *define* new fully generic functions themselves.

1.2 A TASTE OF FULLY GENERIC PROGRAMMING

Generic programming in dependently typed languages [3, 39] is accomplished using a construction known as a universe (Section 2.2). Rather than explaining how universes work in detail (which we do in Section 2.2) in this introduction, we develop our dependently typed Agda examples using universes in parallel with examples in Haskell [35] using type classes [30, 36]. Later we learn why our analogy with Haskell type classes makes sense, as *ad hoc polymorphism* (Section 3.2) is a form of generic programming.

In the following, we first develop the size function using traditional generic programming (in Haskell and Agda), and then develop the count function using *fully* generic programming (albeit over a fixed and small language, and also in Haskell and Agda), both described in the introduction.

1.2.1 Traditional Generic Programming

Recall (from the introduction) that size returns the sum of all inductive constructors, inductive arguments, and non-inductive arguments. Notably, size *only* recurses into inductive constructor arguments.

Haskell In Haskell, we start by defining a type class (`Size`) for the `size` function.

```
class Size a where
  size :: a -> Int
```

The `size` of a boolean is just 1. This is because it has no other non-inductive or inductive arguments to sum.

```
instance Size Bool where
  size b = 1
```

The `size` of a pair is 3, which is the sum of the pair constructor (1) and both of its non-inductive arguments (1 + 1).

```
instance Size (a, b) where
  size (a, b) = 3
```

The `size` of an empty list is just 1, because it has no arguments. The `size` of a "cons" is the sum of the "cons" constructor (1), its single non-inductive argument (1), and the *recursive* `size` of its single inductive argument.

```
instance Size [a] where
  size [] = 1
  size (x : xs) = 2 + size xs
```

Note that the `Size` type class is just ad hoc polymorphism by *overloading* (Section 3.2.1), as each of its instances can be defined independently because they only recurse into inductive arguments.

Agda In Agda, we start by declaring a new type (Size), which is a syntactic reification of the types we wish to generically program size for. Unlike the Haskell version, we must choose the types for which we will provide "instances" upfront.

```
data Size : Set₁ where
  ‘Bool : Size
  ‘Pair : (A B : Set) → Size
  ‘List : (A : Set) → Size
```

Each constructor of Size is not a type, but rather an encoding of a type. Next, we define a function (⟦_⟧) that interprets each encoded Size type as an actual Agda type (i.e., a Set).

```
⟦_⟧ : Size → Set
⟦ ‘Bool ⟧ = Bool
⟦ ‘Pair A B ⟧ = A × B
⟦ ‘List A ⟧ = List A
```

We can generically define size as a *dependent* function from a code (A : Size), to a value of the encoded type (⟦ A ⟧), to a number (ℕ). We case-analyze the first (Size) argument of size to distinguish each different "instance". After that, each second argument and body follows the same logic as the instances in the Haskell version above.

```
size : (A : Size) → ⟦ A ⟧ → ℕ
size ‘Bool b = 1
size (‘Pair A B) (a , b) = 3
size (‘List A) nil = 1
size (‘List A) (cons x xs) = 2 + size (‘List A) xs
```

A significant difference with the Haskell version is that we supply the encoded type explicitly in recursive calls (i.e., ‘List A in the cons case).[2]

[2] It is possible to make this an implicit argument so the Agda surface language also infers it. However, the argument would still be explicit in the underlying core language to which the surface language elaborates.

1.2.2 Fully Generic Programming

Recall (from the introduction) that `count` returns the sum of all inductive constructors, non-inductive constructors, inductive arguments, and non-inductive arguments. Notably, `count` recurses into inductive *and* non-inductive constructor arguments.

Haskell Again, we start by defining a Haskell type class (`Count`) for the `count` function.

```
class Count a where
  count :: a -> Int
```

The `count` of a boolean is still 1, because it has no arguments.

```
instance Size Bool where
  size b = 1
```

The `count` of a pair is the sum of the pair constructor (1), and the *recursive* `count` of both of its non-inductive arguments. Notably, `count` (unlike `size`) recurses into its non-inductive arguments.

```
instance (Count a, Count b) => Count (a, b) where
  count (a, b) = 1 + count a + count b
```

The `count` of an empty list is still 1. The `count` of a "cons" is the sum of the "cons" constructor (1), the *recursive* `count` of its single non-inductive argument, and the *recursive* `count` of its single inductive argument. Notably, `count` (unlike `size`) recurses into its non-inductive argument.

```
instance (Count a) => Count [a] where
  count [] = 1
  count (x : xs) = 1 + count x + count xs
```

The `Count` instances for pairs and lists are able to recurse into their non-inductive arguments because they have type class premises for their type parameters (e.g., the left of the arrow in `(Count a) => Count [a]` in the list instance). This allows instances of one type to recurse into instances of other types, and is called ad hoc polymorphism by *coercion* (Section 3.2.2). The etymology of the name is the idea that `count` for lists can be defined by "coercing" the meaning of `count` for the parameter type of the lists.

Agda In Agda, we declare a new type (Count), reifying the types over which we will generically program count. Unlike Size, Count is an *inductive* type, as the arguments to ‘Pair and ‘List are inductive (i.e., the A and B arguments have type Count below, but they have type Set in the Size datatype).

```
data Count : Set where
  ‘Bool : Count
  ‘Pair : (A B : Count) → Count
  ‘List : (A : Count) → Count
```

The types encoded by Count are interpreted (by the ⟦_⟧ function) as actual Agda types. The ⟦_⟧ function interprets the *inductive* arguments of ‘Pair and ‘List (representing datatype parameters) *recursively*.

```
⟦_⟧ : Count → Set
⟦ ‘Bool ⟧ = Bool
⟦ ‘Pair A B ⟧ = ⟦ A ⟧ × ⟦ B ⟧
⟦ ‘List A ⟧ = List ⟦ A ⟧
```

In Haskell, the `Count` instances for pairs and lists have `Count` type class premises for their type parameters. This allows `count` to recurse into non-inductive arguments of the parameterized types. In Agda, count can recurse into non-inductive arguments (in addition to the inductive arguments) because its parameterized types

are encoded inductively in Count.

```
count : (A : Count) → ⟦ A ⟧ → ℕ
count `Bool b = 1
count (`Pair A B) (a , b) = 1 + count A a + count B b
count (`List A) nil = 1
count (`List A) (cons x xs) = 1 + count A x + count (`List A) xs
```

The logic of count closely follows that of the `count` instances, except encoded types are explicitly supplied in recursive calls. Significantly, count has access to Count type encodings (A and B) in the pair (,) and cons cases, and these type encodings are supplied to recursive calls of non-inductive arguments (a, b, and x). Finally, count still recurses into the inductive argument xs in the cons case using the encoded type `List A.

1.2.3 Universes

In Agda, generic programming (like the count function) is accomplished using a *universe* (Section 2.2). A universe is the combination of a type of *codes* for types (e.g., Count) and a *meaning* function (e.g., ⟦_⟧) mapping codes to actual types. Generic functions (over all types of the universe) are dependent function parameterized over all type codes (Code below) and the meaning (Meaning below) of the particular code supplied.

```
(c : Code) (m : Meaning c) → ...
```

Fixed Types Universe In the count example (using the Count universe), we have a seen how to perform a limited version of *fully generic programming*, in which recursion into both *non-inductive* and *inductive* arguments is possible. The problem with the Count universe is that it is *fixed* to a particular collection of types, chosen ahead of time.

We can add more types (as in Section 2.2.3) to this universe (like natural

numbers, vectors, finite sets, dependent pairs, dependent functions, etc), naming the new type of codes Type, until it contains enough types to model a dependently typed language with a primitive collection of built-in types. Fully generic programming over this universe then models fully generic programming over the entire language modeled by the universe:

$$(A : \mathsf{Type})\ (a : [\![\ A\ ]\!]) \to \ldots$$

However, most modern dependently typed language allow users to declare new algebraic datatypes. The Type universe does not model a language with datatype declarations, as users can only work with the built-in types that have been *fixed* ahead of time.

Extensible Algebraic Types Universe Alternatively, we may define a universe that models algebraic datatypes (as in Section 5.3). We call the type of codes for this universe Desc, as they *describe* algebraic datatype declarations. The meaning function for this universe, named μ, interprets a declaration as the declared type.[3] The Desc universe models an *extensible* collection of algebraic datatypes. Generic programming over this universe allows users to write functions that can be applied to any algebraic datatype a user might declare (whether the type is already declared now or will be declared in the future):

$$(D : \mathsf{Desc})\ (x : \mu\ D) \to \ldots$$

Actually, dependently typed languages can only contain the *strictly positive* (Section 2.1.8) subset of algebraic datatypes (this restriction keeps the language total, hence consistent as a logic under the Curry-Howard isomorphism). A consequence of defining Desc as a strictly positive datatype is that generic programming

[3] As we see in the next section, another way to think about Desc is a reification of pattern functors from initial algebra semantics, whose least fixed point is calculated by μ.

over it corresponds to *ordinary generic programming* (like the size function), in which recursion is restricted to *inductive* arguments.

Fixed Types *Closed Under* Algebraic Extension Universe A primary contribution of this thesis is defining a universe that combines the fixed collection of built-in types universe (Type) with the extensible collection of algebraic datatypes universe (Desc), in a way that supports *fully generic programming* (while remaining consistent under the Curry-Howard isomorphism).

One important property of what makes fully generic programming possible in Count is that the arguments to its codes (i.e., the arguments to ‘Pair and ‘List) are *inductive*. This makes Count a universe of booleans *closed under* pair formation and list formation. Closure properties are an important defining feature of a universe.

The key to defining our combined universe it to make the Type universe not only closed under expected types (like dependent pairs and dependent functions), but also closed under *algebraic datatype formation* (μ) from datatype declarations (Desc). The details of how to make this work are beyond the scope of this introduction (see Section 6.2 for the full construction). However, the central idea has to do with defining the Type and Desc universes *mutually*. Thus, fully generic programming over this mutual universe corresponds to writing *mutually* dependent functions over the following type signatures:

$$(A : \mathsf{Type})\ (a : [\![\ A\ ]\!]) \rightarrow ...$$

$$(D : \mathsf{Desc})\ (x : \mu\ D) \rightarrow ...$$

Essentially, in our mutual universe Type is closed under Type formers (like ‘μ) that can have Desc arguments, and Desc is closed under Desc formers that can have Type arguments. The consequence of our *closed* universe is that it models a dependently typed language supporting datatype declarations *and* fully generic programming.

1.2.4 Fully Generic versus Deriving

Finally, we would like to make an analogy: Having access to fully generic functions (e.g., `count`) defined for all possible types is like `deriving` a type class instance for a datatype in Haskell. In both cases, users get to declare a new datatype and have access to functions operating over it (i.e., fully generic `count` or derived `count`) for free.

The big difference is that users of a closed but extensible dependently typed language (like a variant of Agda) may define fully generic functions themselves. Furthermore, because these are ordinary dependent functions defined within the language, they are ensured to be type-safe. In contrast, users of a non-dependently typed language like Haskell must rely on compiler writers to provide them with derivable functions for a fixed collection of type classes.

1.3 CLASS OF SUPPORTED DATATYPES

Previously (in Section 1.2) we introduced the idea of *fully generic programming* over a mutually defined universe, encoding a fixed collection of primitive types *and* an extensible collection of algebraic datatypes. This section addresses the following question: What properties of algebraic datatypes should we support to adequately describe all possible types definable in a dependently typed language like Agda?

We explain why we choose *inductive-recursive* types, instead of indexed types, as the answer to this question. Non-expert readers may wish to skim this section and come back to it after finishing Part I: Prelude.

1.3.1 Dependent Algebraic Types

We certainly want to support algebraic datatypes with *dependencies* between their arguments. In a non-dependent language like Haskell, the types of all arguments to constructors of an algebraic datatype can be defined independently. In Agda, the

types of subsequent constructor arguments can depend on the *values* of previous constructor arguments. There are 2 common generic encodings (i.e., semantic models) of dependent algebraic datatypes:

◇ **Containers** (Section 4.2.2) These are data structures that represent types using an analogy of *shapes* (capturing inductive structure) and *positions* (capturing contained values). The least fixed points of containers [1] are *well-orderings* [21], or W types.

◇ **Dependent Polynomials** (Section 5.3) These are *pattern functors* [26] from initial algebra semantics, whose least fixed point is returned by the μ operator. The Desc type of Section 1.2 is a syntactic reification of dependent polynomial pattern functors, whose meaning function is μ when considered as a universe of dependent algebraic types.

A universe closed under W types, supporting fully generic programming, is trivial to define (Section 4.2.1). Unfortunately, while W types adequately encode algebraic types in Extensional Type Theory (as implemented by NuPRL [9]), they inadequately [40] (Section 4.2.3) encode first-order algebraic types in Intensional Type Theory (as implemented by Agda [47]). For this reason, we choose **dependent polynomials** to model dependent algebraic types.

1.3.2 Indexing versus Induction-Recursion

Besides supporting algebraic types with dependencies between arguments, Agda also supports algebraic types capturing *intrinsic* correctness properties. There are 2 main special kinds of algebraic types used to capture intrinsic correctness properties:

◇ **Indexed Types** (Section 2.1.5) These are collections of algebraic types, indexed by some type I, such that each type in the collection may vary for any particular value of I. For example, Vectors of A values are indexed by

the natural numbers and map to lists whose lengths are constrained to equal the natural number index.

- **Inductive-Recursive Types** (Section 2.1.9) These are algebraic datatypes mutually defined with a *decoding function* whose domain is the algebraic type and codomain is some type O. For example, Arithmetic expressions (Section 2.1.9) of "Big Pi" formulae are an inductive-recursive type, mutually defined with an evaluation function (as their decoding function) returning the number they encode. The upper bound of "Big Pi" arithmetic expressions is calculated using the mutually defined evaluation function.

Somewhat surprisingly, indexed types [19, 20] and inductive-recursive types [22, 23] define isomorphic classes of datatypes [31]. That is, any indexed type (like Vec) can be defined as an inductive-recursive type, and any inductive-recursive type (like Arith) can be defined as an indexed type.

Thus, picking either indexed or inductive-recursive types is adequate to capture all of the algebraic types we would like to encode in our closed universe. We choose **inductive-recursive** types because there is little research on using them to even do traditional generic programming, so we hope to make inductive-recursive types more popular by providing more examples of programming with them.

1.3.3 Smallness versus Largeness

There are two more significant reasons why picking induction-recursion to showcase generic programming is important. The first is merely an issue of encoding, but the second emphasizes that the isomorphism between indexed and inductive-recursive does not scale to "large" cases, defined below:

- **Intensionality** Even though indexed and inductive-recursive types are isomorphic, encoding "naturally" inductive-recursive types (like Arith) as indexed types means reasoning about the low-level encoding rather than the

high-level intended type definition. Similarly, writing generic functions over inductive-recursive types produces more "natural" results when applied to "naturally" inductive-recursive types, as opposed to encoded indexed types.

- **Largeness** In this thesis we only cover *small* closed universe fully generic programming, meaning the codomain of the inductive-recursive decoding function is a type (like the natural numbers). In contrast, *large* inductive-recursive types may have kinds (Set) as the codomain of their decoding functions. The isomorphism between indexed and inductive-recursive types no longer applies in the large case. Therefore, fully generic programming over small inductive-recursive types may serve as a guide for how to do it in the large case (where one cannot simply use indexed types and apply the isomorphism).

Our arguments (the intensionality of functions and the lack of an isomorphism in the large case) could also be used to justify choosing indexed types (where we consider "naturally" indexed types and large type indices). Once again, we choose inductive-recursive because they are less studied in the generic programming literature.

Finally, because the isomorphism fails in the large case, the ideal choice would be to use **indexed inductive-recursive** [24] algebraic types. These are a 3rd option for expressing intensional correctness properties of datatypes, where both indexing and induction-recursion are expressed naturally.[4] While it is *not technically challenging* to extend our work on fully generic programming over closed universes to indexed inductive-recursive types, we do not do this for *pedagogical* reasons. The necessary background material to explain this combined approach, and the resulting complexity it introduces in generic functions and examples, would

[4] Interestingly, even indexed inductive-recursive types are isomorphic to indexed types and inductive-recursive types in the small case [31].

obscure our lessons on how to define closed universes and perform fully generic programming.

1.4 THESIS

Now we cover our thesis statement, contributions, and outline the remainder of the dissertation.

1.4.1 Thesis Statement

Fully generic programming, supporting functions defined by recursion into all non-inductive and inductive constructor arguments of all types in the universe, is possible over a universe that:

- (Section 6.2) Models a **dependently typed language** (or type theory, supporting the Curry-Howard isomorphism) with datatype declarations.
- (Section 5.4) Adequately (in intensional type theory) models **small inductive-recursive algebraic types** via initial algebra semantics (in contrast to the inadequate model of first-order types in the universe of Section 4.2).
- (Section 3.3.4) Supports the elimination of all (i.e., inductive and non-inductive) values by:
  - (Section 2.2.4) being **inductively defined**, allowing types to be closed under other types.
  - (Section 2.2.3) being **closed**, by not containing values defined using Set.
  - (Section 2.2.7) being **autonomous**, by only containing values whose types are in the universe.
  - (Section 3.3.2) being **concrete**, by only containing types that have some elimination principle.

1.4.2 Contributions

We make the following 3 *primary contributions* to the field of generic programming using dependently typed languages:

1. **Defining** (Chapter 6) a *closed universe* (in Section 6.2), as an adequate model of a dependently typed language with datatype declarations for inductive-recursive types, supporting fully generic programming. Additionally, we define a procedure (in Section 6.3) to close *any* universe.
2. **Examples** (Chapter 7) of writing *fully generic functions* over all *values* of our universe, including count (in Section 7.1), lookup (in Section 7.2), and *marshalling* (ast, in Section 7.3) to an abstract syntax tree.
3. **Extending** (Chapter 8) our closed universe to a *closed hierarchy of universes* (in Section 8.2), supporting fully generic functions over all *types* in the universe hierarchy (in addition to values), via fully generic programming at any universe *level* (in Section 8.3).

1.4.3 Outline

The dissertation is broken up into 4 parts, the **Prelude**, a part on **Open Type Theory**, a part on **Closed Type Theory**, and the **Postlude**:

Part I: Prelude

The prelude reviews background information on dependently typed programming, and serves as a mini-version of our dissertation, in a simplified but unfortunately inadequate setting.

Chapter 1: Introduction This chapter concludes the introduction. We already reviewed dependently typed languages, and how code reuse serves as our motivation (Section 1.1). We also demonstrated what fully generic programming looks

like in a limiting setting, and compared how it works in Agda with how it works in Haskell (Section 1.2). Finally, we explained why we chose inductive-recursive types as the class of algebraic types we wish to write fully generic functions over (Section 1.3).

Chapter 2: Types & Universes We review the concept of types (Section 2.1) and universes (Section 2.2) in type theory. In particular, we classify both types and universes according to a detailed account of various properties they can have.

Chapter 3: Generic Programming We clarify what we mean by *generic programming* (i.e., programming over many types, using various forms of polymorphism [51]), because the meaning of this term is overloaded. We compare and contrast generic programming as parametric polymorphism (Section 3.1) and ad hoc polymorphism (Section 3.2). Additionally, we introduce the idea of *concreteness* (Section 3.3) to help clarify what we mean by *fully* generic programming. Programming total functions in type theory can be non-trivial, especially as the class of types we program over expands during generic programming, so we review techniques to make total programming possible (Section 3.4).

Chapter 4: Closed Type Theory This chapter contains examples of closed type theories (i.e., those that do not contain Set) supporting fully generic programming. We present (Section 4.1) the closed type theory of *Closed Vector Types*, modeling a language with a built-in collection of types related to vector operations. We show how to write a fully generic sum function over the language of *Closed Vector Types*. Then we present (Section 4.2) the closed type theory of *Closed Well-Order Types*, modeling a language with algebraic datatype declarations. Unfortunately, while this closed universe model is easy to define and supports fully generic programming, the W type it uses to model algebraic types is inadequate for our purposes. Even though W types are inadequate for our purposes, it is helpful

to understand a closed universe of dependent types in this simpler setting, before understanding the more complicated (but adequate) version in Chapter 6.

Part II: Open Type Theory

In this part we focus on modeling algebraic datatypes in *open type theory*, whose collection of types grows as more types are declared. While algebraic types defined using W are inadequate (in open type theory and closed type theory), types defined using *initial algebra semantics* are not. This part explains how to model initial algebra semantics in type theory (by defining the Desc and μ types), which is much more involved than defining the W type.

Chapter 5: Open Algebraic Universes In this chapter we progress through a series of initial algebra semantics for incrementally more expressive classes of datatypes, starting with non-dependent algebraic types and ending with inductive-recursive types. We motivate the (formal) type theory models with their category theory equivalents. We also give examples of modeling values, not just types, using initial algebra semantics.

Part III: Closed Type Theory

In this part we switch back to closed type theory, returning back to the setting from which we diverged in Section 4.2.1, but this time using an adequate equivalent of the language of *Closed Well-Order Types*. We also go one step further, defining a closed *hierarchy* of closed types.

Chapter 6: Closed Algebraic Universe In this chapter we define the closed type theory of *Closed Inductive-Recursive Types*. This adapts the previous initial algebra semantics from an open type theory setting to a closed type theory setting. We define the *Closed Inductive-Recursive Types* in Agda, serving as a formal model

of a closed dependently typed language supporting datatype declarations.

Chapter 7: Fully Generic Functions In this chapter we provide examples of writing fully generic functions over *Closed Inductive-Recursive Types*. These functions can be applied to values of any type in our model, can recurse into non-inductive and inductive arguments, and can eliminate any value in our model. Significantly, our generic functions are examples of how to deal with dependencies among *inductive* arguments, as such dependencies only exist for inductive-recursive types.

Chapter 8: Closed Hierarchy of Universes Up to this point we have worked with a closed type theory modeling the first universe of a hierarchy, which contains values but not types. In this chapter we show how to extend a closed type theory to a hierarchy of universes, which contains types (in addition to values) at every level of the hierarchy beyond the first. The chapter reviews how to model a hierarchy of *Closed Well-Order Types*, and then defines a model of the hierarchy of *Closed Inductive-Recursive Types*. We highlight the subtleties necessary to adequately define a hierarchy containing algebraic types modeled using initial algebra semantics.

In this chapter we also show how to extend *fully generic functions* to also be *universe-level generic*. We call such functions *leveled fully generic functions*, and show that they can be applied to any type at any level of the universe hierarchy. Importantly, leveled fully generic programming is possible because our universe hierarchy model is closed (i.e., the hierarchy still does not contain Set, but additionally does not contain Level).

Part IV: Postlude

Finally, we address **Chapter 9: Related Work**, **Chapter 10: Future Work**, and summarize our dissertation in **Chapter 11: Conclusion**.

Major Ideas

Each chapter (besides the introduction and the postlude chapters) is preceded by a paragraph titled **Major Ideas**. This paragraph explains the purpose of the chapter, and anything unconventional, to help prevent readers from getting lost in the details and forgetting the motivation. This paragraph may assume ideas explained within the chapter, so it may be necessary to reference the major ideas as the chapter is read.

Chapter 2

TYPES & UNIVERSES

A *type* is a collection of values, and a *universe* is a collection of types (possibly closed under certain type formers). In this section we review different classes of types (e.g., indexed types, infinitary types, etc). This allows us to be clear about what each class adds to the expressive power of a language (i.e., what sorts of new values we can construct.)

We also review properties of both types and universes (e.g., inductiveness, openness, etc). These properties determine how we can use values (i.e., what elimination principles are valid for them). With a thorough understanding of classifications and properties of types and universes, we can precisely describe which classifications and properties we need to perform the main goal of this thesis, *fully generic programming* (Chapter 3) within *closed type theory* (Chapter 4).

Major Ideas The purpose of this chapter is to review mostly standard terminology used to classify types and universes in DTT. Expert readers may wish to skip this review. One deviation we make from standard terminology is calling a universe *inductively defined* (Section 2.2.4). If the datatype of codes of a universe is inductive, then the universe represented by the codes is "closed under" certain type formers. Nevertheless, we call the universe *inductive* to not confuse that concept with the idea of a *closed* universe (one not defined in terms of Set, as described in Section 2.2.3).

We also introduce some new terminology for describing universes, namely subordinate (Section 2.2.6) and autonomous (Section 2.2.7) universes. Although the

concept of open-versus-closed types and universes is well established, we focus on this uncommon distinction in this dissertation (we do so because a closed universe is essential for fully generic programming). By defining all of these concept, we can precisely capture the universe properties that are necessary to perform *fully generic programming* in Section 3.3.4.

2.1 TYPES

In programming languages, a *type* is a construct used to capture the notion of a collection of *values*. In this section we introduce many different properties of types so that we may precisely describe types in future parts of this thesis. As the primary motivation of a functional programming language is writing functions, we also accompany datatype definitions with example functions operating over said types.

2.1.1 Function Types

Dependently typed functional languages include dependent functions as a primitive. The codomain of a dependent function type may depend on a value of its domain.

$$(a : \mathsf{A}) \to \mathsf{B}\ a$$

Values of function types are lambda expressions, such as the lambda expression in the body of the identity function (*id*) below.

```
id : (A : Set) → A → A
id = λ A → λ a → a
```

2.1.2 Non-Inductive Types

A *non-inductive* type is any type that is not recursively defined. A type can have one or more constructors used to introduce its values. The definition of a non-inductive type does not mention itself in the types of any of the arguments to its constructors.

Functions are an example of a non-inductive type because the domain and codomain of a λ does not recursively mention the function type. Booleans are another example because the true and false constructors do not have arguments. Below is the type of booleans, defined with the negation function not as an example.

```
data Bool : Set where
  false true : Bool

not : Bool → Bool
not false = true
not true = false
```

An even simpler example is the unit type, which only has a single constructor without any arguments.

```
data ⊤ : Set where
  tt : ⊤
```

A non-trivial example is the "maybe" type specialized to booleans (MaybeBool). The just constructor has an argument, but its type is Bool rather than the type being defined (MaybeBool).

```
data MaybeBool : Set where
  nothing : MaybeBool
  just : Bool → MaybeBool
```

2.1.3 Inductive Types

An *inductive* type mentions itself in its definition. That is, at least one constructor has one argument whose type is the type being defined. For example, below is the type of natural numbers (defined with the addition function + as an example). The successor constructor of the type of natural numbers takes a natural number argument, making it inductive.

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

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

An alternative definition of an inductive type is a collection of values closed under certain value constructors (e.g., ℕ as zero closed under suc).

2.1.4 Parameterized Types

A *parameterized* type is a collection of types, parameterized by some type A, such that the collection is uniformly defined for each of its types regardless of what A is.

For example, below the type of disjoint unions (⊎) is *non-dependent*, *non-inductive*, and parameterized by two types A and B. We define the type of disjoint unions along with a function to case-analyze them.

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

case : {A B C : Set} → A ⊎ B → (A → C) → (B → C) → C
case (inj₁ a) f g = f a
```

```
case (inj₂ b) f g = g b
```

Dependent pairs (Σ) are another example. They are *dependent*, *non-inductive*, and parameterized by a type A and a function type B (whose domain is A and codomain is Set). We define the type of dependent pairs along with its dependent projections.

```
data Σ (A : Set) (B : A → Set) : Set where
  _,_ : (a : A) (b : B a) → Σ A B

proj₁ : ∀{A B} → Σ A B → A
proj₁ (a , b) = a

proj₂ : ∀{A B} (ab : Σ A B) → B (proj₁ ab)
proj₂ (a , b) = b
```

A third example is the type of polymorphic lists. They are *non-dependent*, *inductive*, and parameterized by some type A. The example function append combines two lists into a single list.

```
data List (A : Set) : Set where
  nil : List A
  cons : A → List A → List A

append : ∀{A} → List A → List A → List A
append nil ys = ys
append (cons x xs) ys = cons x (append xs ys)
```

2.1.5 Indexed Types

An *indexed* type is a collection of types, indexed by some type I, such that each type in the collection may vary for any particular value of I. For example, the type of vectors (Vec), or length-indexed lists. Vectors are *indexed* by a natural number n (representing their length) and also *parameterized* by some type A.

```
data Vec (A : Set) : ℕ → Set where
```

```
    nil : Vec A zero
    cons : ∀{n} → A → Vec A n → Vec A (suc n)
```

Below we encode the 2-length vector of booleans `[true,false]` and the 3-length vector of natural numbers `[1,2,3]` using Vec.

```
bits : Vec Bool 2
bits = cons true (cons false nil)

nums : Vec ℕ 3
nums = cons 1 (cons 2 (cons 3 nil))
```

The example function append ensures that the length of the output vector is the sum of the lengths of the input vectors.

```
append : ∀{A n m} → Vec A n → Vec A m → Vec A (n + m)
append nil ys = ys
append (cons x xs) ys = cons x (append xs ys)
```

Another example is the type of finite sets (Fin), indexed by the natural numbers. For each natural number n, the type Fin n represents the subset of natural numbers from 1 to n.[1]

```
data Fin : ℕ → Set where
    here : ∀{n} → Fin (suc n)
    there : ∀{n} → Fin n → Fin (suc n)
```

The type Fin 3 encodes the finite set `{1,2,3}`. Below we construct the numbers 1, 2, and 3 as values of the Fin 3 type.

```
one : Fin 3
one = here

two : Fin 3
two = there here
```

[1] Note that the finite set Fin 0 is uninhabited, as the subset of natural numbers from 1 to 0 does not have any values.

```
three : Fin 3
three = there (there here)
```

We give an example function using finite sets, named prod, which computes the product of a list of n natural numbers. However, we represent a list of numbers as a function from Fin n to $\mathbb{N}$. The idea is that each member of the finite set maps to a number (a member of our "list"). For example, the list `[1,2,3]` is represented as the function below.[2]

```
nums : Fin 3 → ℕ
nums here = 1
nums (there here) = 2
nums (there (there here)) = 3
nums (there (there (there ())))
```

Once again, prod takes this functional list representation as an input and returns the mathematical product of all members of the "list". [3]

The base case represents the empty list, for which we return the number one (the identity of the product operation). The recursive case multiplies the current number at the head position of the list (accessed by applying f to the here constructor of finite sets) with the recursive call on the tail of the list (we compute the tail of a list represented as a function by composing the function with the there constructor of finite sets).

```
prod : (n : ℕ) (f : Fin n → ℕ) → ℕ
prod zero f = suc zero
prod (suc n) f = f here * prod n (f ∘ there)
```

Hence, prod applied to 3 and nums produces 6. This is the result of reducing

[2] Technically this is a length-3 vector rather than a list. However, prod also takes a natural number argument, and a dependent pair consisting of a number n and a vector of length n is isomorphic to a list. See Section 2.1.7 on derived types for more discussion.

[3] The final clause serves as a proof that Fin 3 has no inhabitants beyond three. The parentheses `()` serve as a witness that a value of the type that *would* be there there (Fin 0 in this case, which is Fin 3 minus 3 theres) is uninhabited. Such witnesses are required for type checking to be decidable in Agda.

the expression $1 \cdot 1 \cdot 2 \cdot 3$. Note that the first 1 is from the zero case of prod, and the second 1 is the first element of nums.

2.1.6 Type Families

A *type family* is a collection of types, represented as a function from some domain A to the codomain Set.

$$A \rightarrow \mathsf{Set}$$

Any parameterized datatype is a type family, for example the type of lists.

$$\mathsf{List} : \mathsf{Set} \rightarrow \mathsf{Set}$$

Any indexed type is also a type family, for example the type of vectors.

$$\mathsf{Vec} : \mathsf{Set} \rightarrow \mathbb{N} \rightarrow \mathsf{Set}$$

Although the type of vectors contains two arguments rather than one, it is isomorphic to an uncurried version with a single argument:

$$\mathsf{Vec} : \mathsf{Set} \times \mathbb{N} \rightarrow \mathsf{Set}$$

2.1.7 Derived Types

Thus far we have only seen *primitive* types. The type of functions already existed as a primitive in the language. We defined each other type using a datatype declaration, extending our language with a new primitive type. Alternatively, many types can be *derived* from existing types. A derived datatype should be isomorphic to the type we have in mind. Rather than writing a function for each derived type, we derive its constructors as examples of how the derived type is used. For example, we can derive the type of booleans as the disjoint union of two unit types.

$$\mathsf{Bool} : \mathsf{Set}$$
$$\mathsf{Bool} = \top \uplus \top$$

```
false : Bool
false = inj₁ tt

true : Bool
true = inj₂ tt
```

An *indexed type* can be derived as a function by *computing* an appropriate existing type from its index. This is because the type former of an indexed type (such as the type of vectors or finite sets) is a function.

For example, we can derive the indexed type of vectors of length n as a right-nested tuple of pairs containing n values of type A. Each occurrence of A represents a cons). The tuple terminates in the unit type, representing nil.

```
Vec : Set → ℕ → Set
Vec A zero = ⊤
Vec A (suc n) = A × Vec A n

nil : ∀{A} → Vec A zero
nil = tt

cons : ∀{A n} → A → Vec A n → Vec A (suc n)
cons x xs = x , xs
```

As another example, consider the type of finite sets. The finite set type can be derived as a right-nested tuple of disjoint unions of unit types, ending with a bottom type ($\bot$, the type without any constructors). This makes sense because the finite set of zero elements is uninhabited, and the finite set of any other number n offers a choice (of heres and theres) to index any sub-number of n. Here “choice” is interpreted as disjoint union.

```
Fin : ℕ → Set
Fin zero = ⊥
Fin (suc n) = ⊤ ⊎ Fin n
```

```
here : ∀{n} → Fin (suc n)
here = inj₁ tt

there : ∀{n} → Fin n → Fin (suc n)
there p = inj₂ p
```

Besides deriving vectors as a function whose type is *computed* from its index, we can also derive the type of vectors as a *constant* function. Vectors are a special case of a class of datatypes called *containers* [1], which are functions from datatype positions to contained values. Below, the type of vectors is represented as a *constant* function (i.e., one that does not vary for n) whose domain is a finite set of n elements, and whose codomain is A. Think of the function as an n-ary projection for each A value in the vector.

```
Vec : Set → ℕ → Set
Vec A n = Fin n → A

nil : ∀{A} → Vec A zero
nil ()

cons : ∀{A n} → A → Vec A n → Vec A (suc n)
cons x f here = x
cons x f (there p) = f p
```

Above, the nil function receives bottom ($\bot$) as an argument, so we need not define it. The cons function "extends" the function f by returning x if the finite set points to the head of the vector, and otherwise calls the "tail" by applying f to the sub-index p. Notice that in Section 2.1.5 the "list" argument to prod was actually this functional vector representation, so it could have been written as:

```
prod : (n : ℕ) (f : Vec ℕ n) → ℕ
```

Finally, we can derive *non-indexed* types from indexed types by using a *dependent pair*. The dependent pair acts like an existential, where the first component is a value from the index domain and acts as a witness, and the second component

is the indexed type former applied to the witness and acts like a predicate. For example, we can derive the type of lists from the type of vectors as follows.

```
List : Set → Set
List A = Σ ℕ (λ n → Vec A n)

nil : {A : Set} → List A
nil = zero , vnil

cons : {A : Set} → A → List A → List A
cons x (n , xs) = suc n , vcons x xs
```

The first component is zero for the nil constructor. For the cons constructor, the first component is the successor of the natural number n contained within the list being extended (the second argument to cons) represented as a pair.

2.1.8 Infinitary Types

An *infinitary* type is an inductive type where at least one constructor has one function argument whose codomain is the type being defined.[4] The domain can never be the type being defined because negative datatypes make type theory inconsistent [11]. For example, the datatype below is inconsistent with type theory.

```
{-# NO_POSITIVITY_CHECK #-}
data Neg : Set where
  neg : (Neg → Neg) → Neg
```

To motivate the definition of an infinitary type, consider the type of rose trees containing values in node positions and allowing each node to have any finite number of branches.

```
data Rose (A : Set) : Set where
```

[4] *Infinitary types* are also referred to as *generalized inductive definitions* [37].

```
rose : A → List (Rose A) → Rose A
```

Now recall the derived definitions of vectors and lists from Section 2.1.7.

```
Vec : Set → ℕ → Set
Vec A n = Fin n → A

List : Set → Set
List A = Σ ℕ (λ n → Vec A n)
```

If we expand this derived definition of lists (and the inner derived definition of vectors) in the definition of Rose above, we arrive at an alternative but isomorphic definition of rose trees.

```
data Rose (A : Set) : Set where
  rose : A → (n : ℕ) (f : Fin n → Rose A) → Rose A
```

Our new definition of rose trees is an example of an infinitary type, as it contains an argument (f) whose domain is a finite set but whose codomain is the type being defined (Rose).

2.1.9 Inductive-Recursive Types

An *inductive-recursive* type is a collection of values mutually defined with a function parameterized by said type. The mutually defined function is called the *decoding* function. An example of an inductive-recursive type is the type of arithmetic expressions Arith. Values of type Arith encode “Big Pi” mathematical arithmetic product equations up to some finite bound, such as the one below.

$$\prod_{i=1}^{3} i$$

The intuition is that this expression should evaluate to something (the number 6 in this case). The mutually defined (*decoding*) function is exactly the evaluation function. The type is defined as follows.

```
mutual
```

```
data Arith : Set where
  Num : ℕ → Arith
  Prod : (a : Arith) (f : Fin (eval a) → Arith) → Arith

eval : Arith → ℕ
eval (Num n) = n
eval (Prod a f) = prod (eval a) (λ i → eval (f i))
```

A literal number is represented using the Num constructor, evaluating to said number. A mathematical product is represented using the Prod constructor, where the first argument a is the upper bound of the product as an arithmetic expression (3 in the example above), and the second argument f is the body of the product (i in the example above) as a functional representation of a vector of arithmetic expressions. Note that Arith is also an *infinitary type*, as the codomain of f is Arith.

The length of the vector (the argument to Fin in the type of f) should be the evaluation of the upper bound a. Hence, the evaluation function eval must be mutually defined with the type Arith. The Prod constructor evaluates to the product computed with our prod function from Section 2.1.5. We can represent the mathematical equation given earlier as follows.

```
six : Arith
six = Prod (Num 3) (λ i → Num (num i))
```

The result of applying eval to the inductive-recursive (Arith) equation six is the natural number 6. An Arith equation may be nested in its upper bound (a) or body (codomain of f), but the lower bound is always 1. Note that above we define the expression six with the helper function num, which converts the finite set value i to a natural number using one-based indexing.

A more typical example of an inductive-recursive type is a *universe* modeling a dependently typed language, which we will see in Section 2.2.3.

2.1.10 Algebraic Types

An *algebraic* type is a type defined as the fixpoint of a suitable algebra. Although this fixpoint construction is not given directly, it is the semantics of types defined using data declarations. For example, the inductive type of lists is defined as the fixpoint below.

$$\text{List} \triangleq \lambda \text{A}.\ \mu \text{X}.\ 1 + \text{A} \cdot \text{X}$$

In the equation, X is used to ask for recursive arguments (such as the second argument to cons). A non-inductive type like booleans can also be defined by ignoring X.

$$\text{Bool} \triangleq \mu \text{X}.\ 1 + 1$$

We would like to emphasize that this definition of booleans corresponds to the semantics of defining Bool using a data declaration (as in Section 2.1.2). Although it looks syntactically similar to the *derived* definition of booleans using unit and disjoint union in Section 2.1.7, that derived definition is *not* algebraic because it is not defined with μ (either syntactically or semantically). However, some derived types *can* be algebraic if we internalize μ as a type former μ [12], and use this type former to derive type definitions. In the scope of this thesis, an algebraic type is one defined using a data declaration, a μ type former, or a W type former (introduced in Section 4.2.2). Although W types are not syntactically fixpoint constructions, they are semantically very similar so we still call them algebraic.

Finally, below is an example of an indexed type defined algebraically. The index is given as a lambda argument (n) just like the parameter (A). However, the nil and cons constructor must appropriately constrain the index argument (to zero or the successor of the previous vector respectively). Additionally, the recursive argument X takes the index as an argument.

$$\text{Vec} \triangleq \lambda \text{A}.\ \lambda \text{n}.\ \mu \text{X}.\ (\text{n} = \text{zero}) + ((\text{m} : \mathbb{N}) \cdot \text{A} \cdot \text{X}\ \text{m} \cdot \text{n} = \text{suc}\ \text{m})$$

Notice that in cons (i.e., the second summand) the index of the previous vector is given as an explicit argument (m), and the index (n) is constrained to be the successor of that argument.

2.1.11 Computational Families

A *computational family* is an indexed type defined by computing over its index. We have already seen a non-algebraic computational family, namely the derived type of vectors from Section 2.1.7.

```
Vec : Set → ℕ → Set
Vec A zero = ⊤
Vec A (suc n) = A × Vec A n
```

However, computational families can also be algebraic. In the previous section, vectors are algebraically defined by constraining the input index given as a lambda argument. As a computational algebraic family, we case-analyze the lambda index argument to determine the algebra that we take the fixpoint of rather than constraining the input.

$$
\begin{aligned}
\text{Vec} &\triangleq \lambda A.\ \lambda n.\ \mu X.\ \textbf{case}\ \text{n}\ \textbf{of} \\
\text{zero} &\mapsto 1 \\
\text{suc n} &\mapsto A \cdot X\ n
\end{aligned}
$$

Agda does not currently support a high-level syntax (like data) for defining computational algebraic families. Nonetheless, we semantically model them using an internalized μ type [12].

2.1.12 Open Types

An *open* type is any type whose definition mentions the type of types (Set).[5] In an *open type theory* datatype declarations add new types to the language, extending Set with additional type formers. Therefore the collection of type formers (values of type Set) is considered to be "open". Consequently, open languages must prohibit case analysis over Set, because a total function matching against currently defined types becomes partial when a new datatype is declared. One example of an open datatype is the type of heterogenous lists (HList).

```
data HList : Set₁ where
  nil : HList
  cons : {A : Set} → A → HList → HList

append : HList → HList → HList
append nil ys = nil
append (cons x xs) ys = cons x (append xs ys)
```

HList is an open type because its cons constructor has an argument A of type Set, and an argument a whose type is the open type A.

The parametric lists from Section 2.1.4 are another example of an open type, as the a argument in the cons constructor has type A. The type of lists parameterized by A is open because cons uses A, and A has type Set.

2.1.13 Closed Types

A *closed* type is any type whose definition does not mention Set. For example, if we specialize the type of parametric lists to booleans (as the type Bits) the source

[5] A type is open if its definition *directly* mentions Set, for example as an argument to one of its constructors. However, a type is also open if its definition *indirectly* mentions Set. For example, an argument to one of its constructors may be another open type (which is open because it either directly or indirectly mentions Set).

of openness (the parameter A of type Set) disappears.

```
Bits : Set
Bits = List Bool

all : Bits → Bool
all nil = true
all (cons false xs) = false
all (cons true xs) = all xs
```

2.2 UNIVERSES

A *universe* is a collection of *types*, possibly closed under certain type formers. Just as we accompanied types with example functions operating over them in Section 2.1, we accompany universes with example *generic functions* in this section. A *generic function* is any function defined over multiple types.

2.2.1 Universe Model

In a dependently typed language, a universe can be modeled as a type of codes (*representing* the actual types of the universe), and a meaning function (mapping each code to its actual type).

For example the BoolStar universe is comprised of the type of booleans, lists of booleans, lists of lists of booleans, and so on. In other words, it is the Kleene star version of Bits (non-nested lists of booleans) from Section 2.1.13. The type of codes is BoolStar, and its meaning function is ⟦_⟧. As a convention, we prefix constructors of the code type with a backtick to emphasize the distinction between a code (e.g., ‘Bool) and the actual type it denotes (e.g., Bool).

```
data BoolStar : Set where
  ‘Bool : BoolStar
  ‘List : BoolStar → BoolStar
```

```
⟦_⟧ : BoolStar → Set
⟦ ‘Bool ⟧ = Bool
⟦ ‘List A ⟧ = List ⟦ A ⟧
```

To get the actual universe type, we apply the dependent pair type former (Σ) to the codes and meaning function. Therefore, values of the universe are dependent pairs whose first component is a code and second component is a value (the type of the value is the meaning function applied to the code).

```
BoolStarU : Set
BoolStarU = Σ BoolStar ⟦_⟧
```

As a convention, we append the letter U to the type of codes to define the universe type. Our first example member of this universe represents the list of booleans `[true, false]`.

```
bits₁ : BoolStarU
bits₁ = ‘List ‘Bool , cons true (cons false nil)
```

Our second example universe value represents the list of lists of booleans `[[true], [false]]`.

```
bits₂ : BoolStarU
bits₂ = ‘List (‘List ‘Bool) , cons (cons true nil) (cons (cons false nil) nil)
```

Our example generic function over this universe is all, which returns true if all the booleans in any potential list and nested sublists are true.

```
all : (A : BoolStar) → ⟦ A ⟧ → Bool
all ‘Bool b = b
all (‘List A) nil = true
all (‘List A) (cons x xs) = all A x ∧ all (‘List A) xs
```

2.2.2 Open Universes

An *open* universe mentions Set in its type of codes or meaning function. Just as open types grow their collection of values when new types are declared, open

universes grow their collection of types when new types are declared.

An example open universe is DynStar, the universe of dynamic lists closed under list formation. A dynamic list may contain values of any type, but the type must be shared by all values.

```
data DynStar : Set₁ where
  `Dyn : Set → DynStar
  `List : DynStar → DynStar

⟦_⟧ : DynStar → Set
⟦ `Dyn A ⟧ = A
⟦ `List A ⟧ = List ⟦ A ⟧
```

Again, we can encode the actual Kleene star of dynamic types universe (rather than just its codes or meaning function) using a dependent pair.

```
DynStarU : Set₁
DynStarU = Σ DynStar ⟦_⟧
```

In our first example, we represent the list of booleans `[true, false]`. The `Dyn part of the first component of the pair indicates the type of values contained in our list, namely Bool.

```
bits₁ : DynStarU
bits₁ = `List (`Dyn Bool) , cons true (cons false nil)
```

Our second example represents the list of lists of natural numbers `[[1], [2]]`. This time, `Dyn is applied to the type of natural numbers (ℕ).

```
nums₂ : DynStarU
nums₂ = `List (`List (`Dyn ℕ)) , cons (cons 1 nil) (cons (cons 2 nil) nil)
```

A common function to define over parameterized lists is "concat", which flattens a list of lists to a single list. Ordinarily we might define multiple versions of this function, each flattening an increasing number of outer lists.

```
concat1 : {A : Set} → List (List A) → List A
```

```
concat2 : {A : Set} → List (List (List A)) → List A
concat3 : {A : Set} → List (List (List (List A))) → List A
```

Using the DynStar universe, we can define a generic concat function that flattens any number of outer lists. The return type of this function should be a List of As, where A is the dynamic type for the dynamic lists to be flattened. Thus, we first define a function Dyn to extract the dynamic type from a DynStar code by recursing down to the base case ‘Dyn.

```
Dyn : (A : DynStar) → Set
Dyn (‘Dyn A) = A
Dyn (‘List A) = Dyn A
```

Note that Dyn is a *computational family* (Section 2.1.11). Later in the thesis we introduce more specific terminology, calling Dyn a *computational argument family* (Section 3.4.3) that serves as a *domain supplement* (Section 3.4.5). Having defined Dyn, we can define a generic concat function to return a flattened list of dynamic universe values.

```
concat : (A : DynStar) → ⟦ A ⟧ → List (Dyn A)
concat (‘Dyn A) x = cons x nil
concat (‘List A) nil = nil
concat (‘List A) (cons x xs) = append (concat A x) (concat (‘List A) xs)
```

Note that a dynamic ‘Dyn value is flattened by turning it into a single-element list.

2.2.3 Closed Universes

A *closed* universe does not mention Set in its type of codes or meaning function. The BoolStar universe of Section 2.2.1 is an example of a closed universe.

As an edge case, consider the universe (HListStar) of heterogenous lists closed under list formation below.

```
data HListStar : Set where
```

```
    ‘HList : HListStar
    ‘List : HListStar → HListStar

⟦_⟧ : HListStar → Set₁
⟦ ‘HList ⟧ = HList
⟦ ‘List A ⟧ = List ⟦ A ⟧
```

Even though HListStar does not mention Set *directly* in its codes or meaning function, it does mention it *indirectly* because the ‘HList code maps to the open type HList (which mentions Set). Therefore, the HListStar universe is *open*!

2.2.4 Inductive Universes

We call a universe *inductive* if its types are closed over one or more type formers. For example, the BoolStar, DynStar, and HListStar universes above are inductive because they are closed under List formation (via the inductive ‘List code constructor).

2.2.5 Non-Inductive Universes

A universe is *non-inductive* if its types are not closed under any type formers. For example, the Truthy universe below represents types that we want to consider as boolean conditional values.

```
data Truthy : Set where
    ‘Bool ‘ℕ ‘Bits : Truthy

⟦_⟧ : Truthy → Set
⟦ ‘Bool ⟧ = Bool
⟦ ‘ℕ ⟧ = ℕ
⟦ ‘Bits ⟧ = List Bool
```

Below we define the isTrue operation, allowing us to consider any value of the universe as being true or false.

```
isTrue : (A : Truthy) → ⟦ A ⟧ → Bool
```

```
isTrue ‘Bool b = b
isTrue ‘ℕ zero = false
isTrue ‘ℕ (suc n) = true
isTrue ‘Bits nil = true
isTrue ‘Bits (cons false xs) = false
isTrue ‘Bits (cons true xs) = isTrue ‘Bits xs
```

2.2.6 Subordinate Universes

A universe is *subordinate* if one of its types contains a nested type that is not a member of the universe. Hence, a universe is subordinate if one of its types has a constructor with an argument whose type is not a member of the universe.

For example, the open HListStar universe from Section 2.2.3 is subordinate because it contains HList, which has a Set argument in the cons constructor, and Set is not a member of HListStar.

Closed universes can be subordinate too, for example the universe BitsStar contains lists of booleans closed under list formation. The ‘Bits values of this universe contain booleans in cons positions, but booleans are not members of the universe.

```
data BitsStar : Set where
  ‘Bits : BitsStar
  ‘List : BitsStar → BitsStar

⟦_⟧ : BitsStar → Set
⟦ ‘Bits ⟧ = List Bool
⟦ ‘List A ⟧ = List ⟦ A ⟧
```

2.2.7 Autonomous Universes

A universe is *autonomous* if all nested types of its types are also types in the universe. Hence, the type of every argument to every constructor of a universe type must also be a type in the universe.

For example, the closed BoolStar universe of Section 2.2.1 is closed because Bool does not have constructor arguments, and because the universe is closed under List formation (thus any sublist only contains types also in the universe).

Note that open universes can be autonomous. For example, DynStar from Section 2.2.2 includes all types A (of type Set) via the ‘Dyn constructor. Regardless of any other types (such as lists) in the universe, DynStar is autonomous because any type can be injected using ‘Dyn.

2.2.8 Derived Universes

Thus far we have constructed universes with certain properties from scratch, extending the *primitive* types of our language with a *primitive* universe. However, we can also *derive* a universe from any *type family* by considering the type of its indices as the codes and the type family itself as the meaning function. If we do this for the indexed type of finite sets (Fin), we get a universe (Pow) like powerset but without the empty set (because Fin zero is not inhabited).

```
Pow : Set
Pow = Σ ℕ Fin

one₁ : Pow
one₁ = 1 , here

one₂ : Pow
one₂ = 2 , here

two₂ : Pow
two₂ = 2 , there here
```

That is, for every natural number (each ℕ code) we get the subset of the natural numbers from zero to that number minus one (the Finite set).

We can use the same method to derive type of *dynamic* lists (DList) from the type of parameterized lists. Note that this is the type of dynamic lists, rather than

the Kleene star of dynamic values (DynStar from Section 2.2.2).

```
DList : Set₁
DList = Σ Set List

bits : DList
bits = Bool , cons true (cons false nil)

nums : DList
nums = ℕ , cons 1 (cons 2 nil)
```

Reflect on the fact that universes are modeled in type theory as a dependent pair consisting of codes and a meaning function. This pair is just another type, therefore whether we consider Pow and DList to be derived types (Section 2.1.7) or derived universes is merely a matter of perspective.

2.2.9 Parameterized Universes

A *parameterized* universe is a collection of universes, parameterized by some type A, such that the collection is uniformly defined for each universe regardless of what A is.

The model of a parameterized universe (i.e., its representation in type theory) may depend on its parameter in its codes, meaning function, or both. The Kleene star universes of booleans (BoolStar), heterogenous lists (HListStar) and bits (BitsStar) all have a similar structure, namely a specialized base type closed under list formation. Our example parameterized universe abstracts out the base type as a parameter.

```
data ParStar : Set where
  `Par : ParStar
  `List : ParStar → ParStar

⟦_⟧ : ParStar → Set → Set
⟦ `Par ⟧ X = X
```

```
⟦ `List A ⟧ X = List (⟦ A ⟧ X)
```

The `Par code represents the parameterized type, and is interpreted as the second argument to the meaning function. To more easily see how this is a "parameterized" universe, we give the type of the universe as a parameterized dependent pair below.

```
ParStarU : Set → Set
ParStarU X = Σ ParStar (λ A → ⟦ A ⟧ X)

bits₁ : ParStarU Bool
bits₁ = `List `Par , cons true (cons false nil)

bits₂ : ParStarU Bool
bits₂ = `List (`List `Par) , cons (cons true nil) (cons (cons false nil) nil)
```

We can still write concat by injecting values of the parameterized type into a singleton list as with DynStar (Section 2.2.2). Recall that concat for DynStar required a special function Dyn to extract the base type. When defining concat for ParStar, the base type is already an explicit parameter that we can refer to in the return type.

```
concat : ∀{X} (A : ParStar) → ⟦ A ⟧ X → List X
concat `Par x = cons x nil
concat (`List A) nil = nil
concat (`List A) (cons x xs) = append (concat A x) (concat (`List A) xs)
```

We've seen how to derive a universe from a *type family* in Section 2.2.8, but we can also derive a universe from a *universe family*. As an example, we derive the DynStar universe from the ParStar universe. In Section 2.2.2 we defined the type of DynStar codes as a primitive, whereas below we derive DynStar codes as the pair of ParStar and Set (the parameter type of ParStarU).

```
DynStarU : Set₁
DynStarU = Σ (ParStar × Set) (λ { (A , X) → ⟦ A ⟧ X })
```

```
bits₁ : DynStarU
bits₁ = (‘List ‘Par , Bool) , cons true (cons false nil)

bits₂ : DynStarU
bits₂ = (‘List (‘List ‘Par) , Bool) , cons (cons true nil) (cons (cons false nil) nil)
```

Chapter 3

GENERIC PROGRAMMING

Generic programming is the act of writing functions that can be applied to values of a collection of types (a *universe*). Given a collection of types, a *generic function* can be applied to values of any type in the collection. A *polymorphic function* universally quantifies over some collection of values and references an arbitrary member of that collection in its type signature. Therefore, generic functions are merely polymorphic functions. The type of the quantified variable can be seen as the codes of the universe, followed by the meaning function applied to a particular code, followed by the remainder of the type signature.

$$(c : \mathsf{Code})\ (m : \mathsf{Meaning}\ c) \rightarrow \ldots$$

We have already seen many generic functions fitting this pattern in Section 2.1 and Section 2.2. Below we reconsider some of them, while classifying them by different forms of polymorphism. In each of these examples, we emphasize the definition of Code (i.e., what the function is polymorphic over).

Major Ideas This chapter clarifies our definition of generic programming, relating it to parametric (Section 3.1) and ad hoc (Section 3.2) polymorphism. It also introduces non-standard terminology, namely the properties of abstractness (Section 3.3.1) and concreteness (Section 3.3.2), which can apply to both types and universes. This final bit of terminology allows us to precisely capture the universe properties (along with properties from Section 2.1 and Section 2.2) necessary to perform *fully generic programming* in Section 3.3.4.

We also include a section on dependently typed programming techniques used to write *total functions* (Section 3.4), which often become necessary when writing sufficiently complex generic functions. The techniques of Section 3.4 are primarily used in Chapter 7.

3.1 PARAMETRIC POLYMORPHISM

A *parametrically polymorphic* function is defined uniformly over its codes and their meanings. That is, the function does not inspect the type of codes and therefore does not behave differently for any code or its *interpretation* (i.e., it does not behave differently for different values in the type returned by the meaning function applied to a code).

3.1.1 Parametric over Types

A common form of parametric polymorphism is over types(i.e., where Code is defined to be Set).

```
append : {A : Set} → List A → List A → List A
append nil ys = ys
append (cons x xs) ys = cons x (append xs ys)
```

Notice that append over lists behaves the same way for any type A that it is applied to.

3.1.2 Parametric over Levels

Functions can also be defined parametrically over universe Levels.[1] Types in Agda are arranged in a hierarchy, where base types are classified by Set0, kinds are

[1] Here, a "universe" refers to all types, or all kinds, or all superkinds, etc. This use of the word universe is distinct from a type of codes and a meaning function. While these are related (the former is the image of the meaning function of the latter), the former refers to a level in a hierarchy of types, while the latter is a technical formal device used for generic programming or modeling a domain.

classified by Set1, superkinds are classified by Set2, and so on. Rather than defining different functions operating over types in each of these levels, we can define a single function level-polymorphically.

```
append : {ℓ : Level} {A : Set ℓ} → List A → List A → List A
append nil ys = ys
append (cons x xs) ys = cons x (append xs ys)
```

Note that append now behaves the same way for any type at any level that it is applied to.

3.2 AD HOC POLYMORPHISM

An *ad hoc polymorphic* function is defined non-uniformly over its codes or their meanings. That is, the function may inspect codes and its interpretation (the values in the type returned by the meaning function applied to a code).

3.2.1 Ad Hoc by Overloading

If the type of Codes for our universe is *algebraic* and *non-inductive*, then generic functions over the universe amount to a kind of syntactic overloading of function names.

For example, consider the isTrue function from Section 2.2.5 over the Truthy universe. Before defining isTrue for the universe, we can define versions of the function for each type in the universe.

```
isTrueBool : Bool → Bool
isTrueBool b = b

isTrueℕ : ℕ → Bool
isTrueℕ zero = false
isTrueℕ (suc n) = true

isTrueBits : List Bool → Bool
```

```
isTrueBits nil = true
isTrueBits (cons false xs) = false
isTrueBits (cons true xs) = isTrueBits xs
```

Now we can define isTrue by matching on each type code, and returning the appropriate function specialized to that type.

```
isTrue : (A : Truthy) → ⟦ A ⟧ → Bool
isTrue ‘Bool = isTrueBool
isTrue ‘ℕ = isTrueℕ
isTrue ‘Bits = isTrueBits
```

Another way to say this is that we can make recursive calls on interpretations, but not codes. For example, below we inline the specialized functions as is done in Section 2.2.5. The ‘Bits cases make recursive calls on inductive values, but the codes stay constant in recursive calls.

```
isTrue : (A : Truthy) → ⟦ A ⟧ → Bool
isTrue ‘Bool b = b
isTrue ‘ℕ zero = false
isTrue ‘ℕ (suc n) = true
isTrue ‘Bits nil = true
isTrue ‘Bits (cons false xs) = false
isTrue ‘Bits (cons true xs) = isTrue ‘Bits xs
```

3.2.2 Ad Hoc by Coercion

If the type of Codes for our universe is *algebraic*, *inductive*, and *autonomous*, then generic functions over the universe can make recursive calls on both codes and their interpretations. Because we can make recursive calls on types of our universe, we can effectively *coerce* recursive values of our universe to an appropriate output type.

For example, consider the concat function from Section 2.2.2 over the DynStar universe. Each value and subvalue of this dynamic Kleene star universe can be

coerced to a dynamic list.

$$
\begin{array}{l}
\mathsf{concat} : (A : \mathsf{DynStar}) \to [\![\ A\ ]\!] \to \mathsf{List}\ (\mathsf{Dyn}\ A) \\
\mathsf{concat}\ (\text{`}\mathsf{Dyn}\ A)\ x = \mathsf{cons}\ x\ \mathsf{nil} \\
\mathsf{concat}\ (\text{`}\mathsf{List}\ A)\ \mathsf{nil} = \mathsf{nil} \\
\mathsf{concat}\ (\text{`}\mathsf{List}\ A)\ (\mathsf{cons}\ x\ xs) = \mathsf{append}\ (\mathsf{concat}\ A\ x)\ (\mathsf{concat}\ (\text{`}\mathsf{List}\ A)\ xs)
\end{array}
$$

3.2.3 Ad Hoc by Overloading & Coercion

Ad hoc polymorphic functions may also be a hybrid of the overloading and coercion styles. For example, if universe Codes are *algebraic*, *inductive*, and *subordinate* then we can recurse on the codes and interpretations for the autonomous types in the universe (coercion), but can only recurse on the interpretations of the subordinate types (overloading). For example, consider the all function for the BitsStar universe of Section 2.2.6.

$$
\begin{array}{l}
\mathsf{all} : (A : \mathsf{BitsStar}) \to [\![\ A\ ]\!] \to \mathsf{Bool} \\
\mathsf{all}\ \text{`}\mathsf{Bits}\ \mathsf{nil} = \mathsf{true} \\
\mathsf{all}\ \text{`}\mathsf{Bits}\ (\mathsf{cons}\ \mathsf{false}\ xs) = \mathsf{false} \\
\mathsf{all}\ \text{`}\mathsf{Bits}\ (\mathsf{cons}\ \mathsf{true}\ xs) = \mathsf{all}\ \text{`}\mathsf{Bits}\ xs \\
\mathsf{all}\ (\text{`}\mathsf{List}\ A)\ \mathsf{nil} = \mathsf{true} \\
\mathsf{all}\ (\text{`}\mathsf{List}\ A)\ (\mathsf{cons}\ x\ xs) = \mathsf{all}\ A\ x \wedge \mathsf{all}\ (\text{`}\mathsf{List}\ A)\ xs
\end{array}
$$

The `Bits cases only recurse over the interpretation (keeping the code constant), hence they are defined by overloading. The `List cases recurse both over the codes and the interpretation, hence they are defined by coercion.

3.3 ABSTRACTNESS & CONCRETENESS

In this section we define additional non-standard terminology for types and universes, namely *abstractness* and *concreteness*. We cover these properties here rather than in Section 2.1 on types and Section 2.2 on universes because the terminology (as it is used) and our emphasis on it is unique to this thesis.

3.3.1 Abstract Types

An *abstract* type is any type that does not have an elimination principle. For example, *open* types (Section 2.1.12) mentioning Set are abstract because you cannot pattern match on values of Set (or otherwise eliminate Set).

Types mentioning Level (used to indicate which level in a hierarchy a type inhabits) are also abstract. Once again, Agda types are arranged in a hierarchy, where base types are classified by Set0, kinds are classified by Set1, superkinds are classified by Set2, and so on. Rather than defining different datatypes inhabiting each of these levels, we can define a single datatype that can be instantiated at any level.

In the example below, we define parameterized lists that can be instantiated at any level in the type hierarchy. The definition also enforces the constraint that a list must inhabit the same level as its type parameter.

```
data List {ℓ : Level} (A : Set ℓ) : Set ℓ where
  nil : List A
  cons : A → List A → List A
```

Just like Set, Agda does not expose an elimination principle for Level (thus you cannot, for example, pattern match on levels in a function definition).[2]

3.3.2 Concrete Types

A *concrete* type is any type that does have an elimination principle. For the purpose of this thesis, this will mean any type that does not mention Set or Level. Therefore, concrete types have the special properties that all of its values and subvalues may be eliminated. Algebraic datatypes are concrete and they may be eliminated by pattern matching. Function types are concrete and they may be eliminated by application.

[2] Nevertheless, we can write parametrically level-polymorphic functions over List as in Section 3.1.2.

3.3.3 Abstract Data Types

This thesis does not consider abstract data types (ADT's), but we touch upon them briefly here to relate them to our terminology of abstract and concrete types.

An ADT allows the user to expose a type former, constructors, and elimination principles while hiding their implementation. For example, a dictionary may be exposed as a list of key/value pairs but internally be implemented as a balanced binary search tree. Therefore, an ADT defined to expose its type former and constructors, but not its elimination principle, is *abstract* by our definition. However, if such an ADT also exposed its elimination principle we would call it *concrete* (despite the fact that the ADT would be hiding its true implementation).

3.3.4 Fully Generic Programming

We call a universe abstract if at least one of the types it contains is abstract. We call a universe concrete if all of the types it contains are concrete. This brings us to the primary ambition of this thesis, which we call *fully generic programming*. A fully generic function is a special kind of ad hoc polymorphic function by coercion (Section 3.2.2) with the additional property that the universe is *concrete*.[3] By consequence, fully generic functions can eliminate any value or subvalue (including both inductive and non-inductive constructor arguments) and recurse on any universe code.

For example, the BoolStar universe (Section 2.2.1) allows us to define nor as a fully generic function (returning true if all values and subvalues contain false).

```
nor : (A : BoolStar) → ⟦ A ⟧ → Bool
nor `Bool true = false
nor `Bool false = true
nor (`List A) nil = true
```

[3] Because the universe is both *inductively defined* (Section 2.2.4) and *concrete* (Section 3.3.2), it is also *closed* (Section 2.2.3).

$\mathsf{nor}\ (\text{`}\mathsf{List}\ A)\ (\mathsf{cons}\ x\ xs) = \mathsf{nor}\ A\ x \wedge \mathsf{nor}\ (\text{`}\mathsf{List}\ A)\ xs$

In addition to making recursive calls on codes and interpretations (thanks to an algebraic, inductive, and autonomous universe) for the ‘List cases, we can also pattern match on all [sub]codes and all [sub]values (thanks to concreteness). Compare this to concat for DynStar in Section 3.2.2. The concat function cannot pattern match on the interpretation of the ‘Dyn base case because the type is Set (the source of abstractness). By contrast, nor can match on the interpretation of ‘Bool by distinguishing between true and false (because Bool is a concrete type).

3.4 TOTALITY

Functions written in dependent type theory (DTT) must be *total* (defined over all inputs). Thus, partial functions written in traditional languages cannot be directly encoded as functions in DTT. In this section, we explain a general technique for altering the type signature of a partial function so that it may be encoded as a total function in DTT. We use the head function as our running example of a partial function that we wish to encode in a total language.

$\mathsf{head} : \{A : \mathsf{Set}\} \to \mathsf{List}\ A \to A$

Applying head to a non-empty list should return the first element, but applying head to an empty list should be undefined. Below we explain how to encode head as a total function by altering either the domain or codomain, first by using non-dependent types and then by taking advantage of dependent types.

3.4.1 Non-Dependent Domain Change

In a non-dependent language, we could write head as a total function by adding an extra A argument to the domain. This extra argument serves as a “default”

argument to return in the (otherwise partial) empty list case.

```
head : {A : Set} → List A → A → A
head nil y = y
head (cons x xs) y = x
```

3.4.2 Non-Dependent Codomain Change

In a non-dependent language, we could also write head as a total function by changing the return type to Maybe A. This allows us to dynamically model partiality by failing with nothing in the empty list case, and succeeding with just in the non-empty list case.

```
head : {A : Set} → List A → Maybe A
head nil = nothing
head (cons x xs) = just x
```

3.4.3 Dependent Domain Change

Without dependent types we can add a default argument to head. Unfortunately, a user must supply this default argument even if they are taking the head of a non-empty list. With dependent types, we can add an extra *dependent* argument to head. The type of the extra argument depends on the input list, and is defined below as a *computational argument type* (a type family defined as a computation as in Section 2.1.11, in an argument position of a function).

```
HeadArg : {A : Set} → List A → Set
HeadArg {A = A} nil = A
HeadArg (cons x xs) = ⊤
```

If the input list is empty, HeadArg computes to A, the type of the default argument that is required. If the input list is non-empty, HeadArg returns the unit type (⊤). Because a value of type unit can always be trivially constructed, this is equivalent to not having an extra argument at all when the input list is non-empty.

Now we can define head with HeadArg as its extra argument.

```
head : {A : Set} (xs : List A) → HeadArg xs → A
head nil y = y
head (cons x xs) tt = x
```

Notice that head only receives the default argument y in the empty list case. Otherwise, it receives the trivial tt constructor of the unit type.

3.4.4 Dependent Codomain Change

Without dependent types we can change the codomain to dynamically model partiality using the Maybe type. Dependent types allow us to statically enforce partiality. We define the return type to be a *computational return type* (a type family defined as a computation, in the return type position of a function).

```
HeadRet : {A : Set} → List A → Set
HeadRet nil = ⊤
HeadRet {A = A} (cons x xs) = A
```

If the input list is non-empty, HeadRet computes the standard return type A. However, if the list is empty then HeadRet computes the unit type. A function returning unit may as well be undefined, as its output is uniquely determined to be tt.

```
head :{A : Set} (xs : List A) → HeadRet xs
head nil = tt
head (cons x xs) = x
```

Rather than dynamically enforcing partiality by returning a nothing failure value for non-empty lists, head is statically "partial" as its definition for the empty list case is uniquely determined.

3.4.5 Domain Predicates versus Domain Supplements

We have seen two different ways (in Section 3.4.3 and Section 3.4.4) to make head total using dependent types, first by adding missing data (the default argument A), and second by effectively making the function undefined for its "partial" cases.

A more common approach is to directly model partiality as a computational argument type that *requests* an argument of the empty type for the empty list case. This in contrast to modeling partiality as a computational return type (Section 3.4.4) that *returns* unit for the empty list case.

```
HeadArg : {A : Set} → List A → Set
HeadArg {A = A} nil = ⊥
HeadArg (cons x xs) = ⊤
```

This allows us to leave the empty list case undefined, as a value of type ⊥ is known not to exist.

```
head : {A : Set} (xs : List A) → HeadArg xs → A
head nil ()
head (cons x xs) tt = x
```

It is clear that the computational argument type HeadArg above acts a *domain predicate*, refining the domain of all lists to be undefined for the empty list by asking the user to provide a value of the empty type (⊥). Compare this to the version of HeadArg in Section 3.4.3, which requests an extra argument (A) in the empty list case. The Section 3.4.3 HeadArg is also technically a domain predicate, as it restricts the input of all lists to supply additional data (A) in the empty list case (i.e., head is no longer defined for all lists, only those with additional data). However, this usage of the word "predicate" feels unnatural, as predicates are associated with logically restricting a domain (rather than requesting additional data). For this reason, we prefer to call the Section 3.4.3 HeadArg a *domain supplement* (this is a non-standard term that we are introducing). Thus, we have two options when embedding a partial function in type theory:

1. Use a domain predicate to restrict the domain, avoiding definitions for the partial cases. For example, adding an empty type argument or returning unit.
2. Use a domain supplement to request additional data, computing results for the partial cases using the additional data. For example, returning a default value provided as an additional argument.

This thesis focuses more on the second option. Functions made total using domain supplements are more interesting than ones using domain predicates, as the supplement adds computationally relevant data rather than just restricting the domain to be undefined for certain cases. Thus, a domain supplement is like a proof-relevant version of a domain predicate (even though both technically restrict the domain).

Conclusion

We have seen how to encode partial functions within total type theory by modifying the domain or codomain of a function, with and without the benefits afforded by dependent typing. Previously, when writing ordinary functions over types (Section 2.1), and especially when writing generic functions over universes (Section 2.2), we deliberately chose examples that were naturally total to avoid using the techniques of this section.

However, as we write generic programs over larger universes (those containing more types), it often becomes necessary to use computational argument or return types to make generic functions total. This is particularly true when writing fully generic functions (Section 3.3.4), as it might not be possible to define them for certain values of a universe without domain supplements.

Chapter 4

CLOSED TYPE THEORY

A *closed type theory* is a dependently typed language with a built-in collection of types (i.e., *primitives*) that will never be extended. Such a type theory can be modeled as a *fully closed universe.* To qualify as a closed type theory, we require that its collection of types is at least closed under dependent function (Π) formation.

It is reasonable to assume that programming in a closed type theory would be limiting, as programmers can only work with the types that are built into the theory (compared to an *open* theory where users may *extend* the language with custom types). However, it turns out that a closed theory with an appropriate collection of primitives can be used to *model* any custom type using only the primitives. Hence, instead of extending an open theory with custom types using datatype declarations, isomorphic versions of custom types may be formed in a closed theory from its primitives.

Therefore, a closed type theory can model a dependently typed programming language supporting custom types. Assume that we make the universe model of such a theory algebraic, inductive, autonomous, and concrete. This language supports *fully generic programming* (Section 3.3.4), allowing programmers to write functions over all types of the language, including custom types![1]

[1] By analogy, consider generic functions (like equality via `Eq` or comparison via `Ord`) that a language like Haskell can derive for any appropriate type using the `deriving` keyword. While users of Haskell are limited to deriving the generic functions built into the compiler, users of such a closed type theory may write their own generic functions operating over any appropriate type.

Major Ideas This chapter gives two examples of closed universes that can serve as models of dependently typed languages. The first universe models a language with a fixed collection of of built-in types related to vectors (Section 4.1). The second universe models a language supporting user-declared types, by including the type of well-orderings (W) as a built-in type (Section 4.2.1). Although we could perform fully generic programming over this universe, the universe is inadequate for our purposes (Section 4.2.3). Nevertheless, it is easy to understand, and is good background material for the universe of user-declared types in Chapter 6 that we actually use for fully generic programming (which replaces the built-in well-order type W with a built-in fixpoint type μ_1).

4.1 CLOSED VECTOR UNIVERSE

In this section we present one example of a closed type theory, which we call the universe of *Closed Vector Types*. This universe contains some standard types along with some types specifically for writing programs operating over vectors. The *Closed Vector Types* universe is an example of a simple closed type theory (or programming language) with a fixed set of primitives that does *not* support custom user-defined types.

4.1.1 Closed Vector Types

Below is the formal model (that is, the model within type theory) of the *Closed Vector Types* universe. It has standard types like the empty type ($\bot$), the unit type ($\top$), booleans (Bool) and is closed under dependent pair formation (Σ) and dependent function (Π) formation. However, we call it the *Closed Vector Types* universe because it also includes types for writing vector-manipulating programs, namely the natural numbers ($\mathbb{N}$) and finite sets (Fin), and is closed under vector

(Vec) formation.

```
data `Set : Set where
  `⊥ `⊤ `Bool `ℕ : `Set
  `Fin : ℕ → `Set
  `Vec : `Set → ℕ → `Set
  `Σ `Π : (A : `Set) (B : ⟦ A ⟧ → `Set) → `Set

⟦_⟧ : `Set → Set
⟦ `⊥ ⟧ = ⊥
⟦ `⊤ ⟧ = ⊤
⟦ `Bool ⟧ = Bool
⟦ `ℕ ⟧ = ℕ
⟦ `Fin n ⟧ = Fin n
⟦ `Vec A n ⟧ = Vec ⟦ A ⟧ n
⟦ `Σ A B ⟧ = Σ ⟦ A ⟧ (λ a → ⟦ B a ⟧)
⟦ `Π A B ⟧ = (a : ⟦ A ⟧) → ⟦ B a ⟧
```

Recall our naming convention of prefixing universe code constructors (e.g., `Bool) with a backtick to distinguish the code (the "quoted" version of the type) from the actual type it models (in this case Bool, which is the result of applying the meaning function to the code). For closed type theories we establish the new naming convention of prefixing the type of codes (e.g., `Set) with a backtick. Thus the type of the meaning function is a function whose domain is `Set (a "quoted" type of types) and whose codomain is Set (the actual type of types). This promotes our definition-level quoting analogy (⟦ `Bool ⟧ = Bool) to the type signature level (⟦_⟧ : `Set → Set).

Finally, notice that `Set is inductive-recursive (Section 2.1.9), as its `Σ and `Π constructors refer to the meaning function in their codomain argument (B). Any universe modeling a dependently typed language is similarly inductive-recursive, as the universe must have Π types to qualify as a model for DTT.

4.1.2 Fully Generic Functions

Just like the fully closed BoolStar universe of Section 3.3.4, ‘Set also supports writing fully generic functions. Fully generic functions, over a universe model of a closed type theory, model pattern matching on types (Set) by pattern matching on codes (‘Set) instead. Therefore, a generic function over all values of all types is modeled by matching on a code, then a value from the interpretation of that code, followed by any additional arguments and the return type.

$$(A : \text{‘Set}) \to [\![\ A\ ]\!] \to \ldots$$

Thus pattern matching on types is supported in a closed type theory, because we know ahead of time that the collection of types will never be extended (hence total functions over types never become partial).

Fully Generic Sum without Function Body

Our example fully generic function is sum, summing up all natural numbers (and values that can be coerced into natural numbers) contained within a value of the closed vector universe.[2]

$$\text{sum} : (A : \text{‘Set})\ (a : [\![\ A\ ]\!]) \to \mathbb{N}$$

Let’s begin with summing values of base types (non-function type formers, like Bool): Summing a natural number just means returning it. We can never receive a value of the empty type, so we need not define that case. The sum of unit is 0. The sum of a boolean is 0 if it is false, and 1 if it is true (true is the second constructor of Bool, just as suc zero is the second value in the ordered natural numbers). The sum of a finite set Fin value is 0 for the here index, and the successor of the previous

[2] The sum function is different from the count function (from Section 1.2.2, which sums the total number of nodes). Instead, sum returns the sum of all values that have been interpreted as natural numbers by coercion.

index for the there case.

```
sum ‘ℕ n = n
sum ‘⊥ ()
sum ‘⊤ tt = 0
sum ‘Bool true = 0
sum ‘Bool false = 1
sum (‘Fin (suc n)) here = 0
sum (‘Fin (suc n)) (there p) = suc (sum (‘Fin n) p)
```

We sum a vector by adding together all of the values it contains, where each value is interpreted as a natural number by recursive application of sum. The empty vector contains no values, so its sum is 0.

```
sum (‘Vec A zero) nil = 0
sum (‘Vec A (suc n)) (cons x xs) = sum A x + sum (‘Vec A n) xs
```

A pair is like a two-element vector, so we sum a pair by adding its components. Finally, the sum of a function is 0.

```
sum (‘Σ A B) (x , xs) = sum A x + sum (B x) xs
sum (‘Π A B) f = 0
```

Defining the sum of a function to be 0 may seem unsatisfying, as its body contains other values of our closed vector universe. Consider the types of variables in context when defining the function case of sum:

```
B : ⟦ A ⟧ → ‘Set
f : (x : ⟦ A ⟧) → ⟦ B x ⟧
```

The sum cases for pairs and functions are actually for *dependent* pairs (‘Σ) and functions (‘Π). Notice that, in our definition of sum for pairs, the recursive call of sum for the second component applies the codomain B to an ⟦ A ⟧ value. Luckily, the first component of the pair (x) is exactly the value we need. If we wanted to provide an alternative definition of sum for functions, we would have no such luck because the value we are summing (f) is itself a function (i.e., there is no x in

sight). In the next section we change our definitions to end up with an x in the function case that we can pass to both B and f.

Fully Generic Sum with Function Body

Step back for a moment and consider what the sum of a function should mean. One interpretation is to consider the sum of a function to be many possible sums, one for each argument in the domain of the function. Under this interpretation our sum is missing an argument, one that provides a domain value for each occurrence of a function in the closed vector universe value we wish to sum.

In other words, we consider our previous definition of sum to be a partial function (we cannot appropriately define the function case). In Section 3.4.5 we learned how to create a total function from a partial one by adding a *domain supplement*. Below, we define the computational argument family Sum to be used as a domain supplement for sum.

The most significant case is the supplement for functions, in which we request the interpretation of A as an additional argument (let's call it x). When defining sum we will want to use x to recursively sum the body of the function, so we also recursively request a supplement for the codomain of our function (B) applied to x.

```
Sum : (A : `Set) → ⟦ A ⟧ → Set
Sum (`Π A B) f = Σ ⟦ A ⟧ (λ x → Sum (B x) (f x))
```

A pair may contain functions in either of its components, so we recursively request supplements for its domain and codomain. An empty vector requires no supplement, but a non-empty vector requires one for each of its elements. Finally, the base types do not need supplemental values to sum them, so their supplement is a trivial unit value.

```
Sum (`Σ A B) (x , xs) = Sum A x × Sum (B x) xs
Sum (`Vec A zero) nil = ⊤
```

```
Sum ('Vec A (suc n)) (cons x xs) = Sum A x × Sum ('Vec A n) xs
Sum A a = ⊤
```

Below the domain of sum's type is altered to take the supplement Sum as an additional argument.

```
sum : (A : 'Set) (a : ⟦ A ⟧) → Sum A a → ℕ
```

The only change we make to defining sum for base types and Fin is to ignore the additional trivial unit value (tt), and to supply it as an argument in the recursive case of 'Fin.

```
sum '⊥ () tt
sum '⊤ tt tt = 0
sum 'Bool false tt = 0
sum 'Bool true tt = 1
sum 'ℕ n tt = n
sum ('Fin (suc n)) here tt = 0
sum ('Fin (suc n)) (there p) tt = suc (sum ('Fin n) p tt)
```

Summing pairs and vectors is also relatively unchanged. For pairs the only difference is that we thread along the left component of the supplement (y) when summing the left component of the pair (x), and the right component of the supplement (ys) when summing the right component of the pair (xs). Summing a non-empty vector is similar to summing a pair, and summing an empty vector remains 0.

```
sum ('Σ A B) (x , xs) (y , ys) = sum A x y + sum (B x) xs ys
sum ('Vec A zero) nil tt = 0
sum ('Vec A (suc n)) (cons x xs) (y , ys) = sum A x y + sum ('Vec A n) xs ys
```

Finally, we can sum a function by summing its body. Its body is obtained by applying the function f to x (whose type is the interpretation of A), readily available in the domain supplement. Of course, the body may have additional

functions to sum, so we thread along the domain supplement y for any of those.

```
sum (‘Π A B) f (x , y) = sum (B x) (f x) y
```

Conclusion

The *Closed Vector Types* universe has enough types to write a lot of interesting functions, but the specific collection of types that our closed type theory contains is arbitrarily chosen. What if we later decide we also want binary trees? By definition we cannot add custom types to a closed type theory (and if we did it would violate the totality of generic functions over the original universe). Next (in Section 4.2) we see how to model a closed type theory that *does* support custom user-defined types.

4.2 CLOSED ALGEBRAIC UNIVERSE

On one hand, we would like a closed type theory because it supports fully generic programming (Section 3.3.4) via pattern matching on types (modeled by pattern matching on *codes* of types). On the other hand, we want to support custom user-defined types (like an open type theory) that may not be present in the closed collection of types we fixed ahead of time.

What if our closed theory had enough primitive base types and type families to simulate adding new algebraic datatypes to the language? That is, we want to support translating any “new” type declaration into an isomorphic type defined in terms of our closed collection of primitive types. In this section we present such a theory and call it the *Closed Well-Order Types* universe.

4.2.1 Closed Well-Order Types

The type of *well-orderings* (W) is used to define the semantics of inductive datatypes in type theory, and is the key to solving our problem. After pruning some derivable types from the previous universe and adding W types, we get a closed type theory (the *Closed Well-Order Types* universe) that can internally represent any type that would normally extend the language in an open type theory. Before explaining what W types are and how they can be used to derive inductive types (Section 4.2.2), we use them below to define a closed type theory universe supporting custom user-defined types.

```
data `Set : Set where
  `⊥ `⊤ `Bool : `Set
  `Σ `Π `W : (A : `Set) (B : ⟦ A ⟧ → `Set) → `Set

⟦_⟧ : `Set → Set
⟦ `⊥ ⟧ = ⊥
⟦ `⊤ ⟧ = ⊤
⟦ `Bool ⟧ = Bool
⟦ `Σ A B ⟧ = Σ ⟦ A ⟧ (λ a → ⟦ B a ⟧)
⟦ `Π A B ⟧ = (a : ⟦ A ⟧) → ⟦ B a ⟧
⟦ `W A B ⟧ = W ⟦ A ⟧ (λ a → ⟦ B a ⟧)
```

The closed type theory above consisting of the empty type (⊥), the unit type (⊤), and booleans (Bool) closed under dependent pair (Σ) formation, dependent function (Π) formation, and well-order (W) formation allows us to model datatype declarations. We show how to model datatype declarations by translating them into W types and other primitive types in Section 4.2.2. In Section 4.2.3 we show that the universe of this section is sufficient for all such translations.

4.2.2 Open Well-Order Types

The type of well-orderings [21] (W) can be used to model inductive datatype declarations as well-founded trees.[3] It is defined below, where the A parameter encodes non-inductive arguments for each constructor of an algebraic datatype, and the cardinality of $B\ a$ encodes the number of inductive arguments for each constructor.[4] The W type of well-orderings is *open* due to its two open type parameters, A and B (in contrast, the arguments of the closed ‘W constructor are closed ‘Set types.).

```
data W (A : Set) (B : A → Set) : Set where
  sup : (a : A) (f : B a → W A B) → W A B
```

We show how to model the semantics of inductive datatypes using W by:

1. Starting with a high-level inductive datatype declaration.
2. Translating between a series of isomorphic datatype declarations.
3. Finally reaching a datatype declaration that can be encoded using a W type.

As an example of elaborating a datatype declaration to a W type, we begin with the Tree type below. In the series of paragraphs that follow, we change the definition of Tree by applying isomorphisms. Our binary Tree type begins with leaves containing A values and binary branches containing B values in the middle of each branch.

```
data Tree (A B : Set) : Set where
  leaf : A → Tree A B
  branch : Tree A B → B → Tree A B → Tree A B
```

[3] The etymology of “well-orderings” comes from W being the constructive version of the classical notion of a well-order. A well-order interprets a set as an ordinal α and a relation specifying which ordinals are less than α. However, in this thesis we focus on the more practical interpretation of W types as a means to define algebraic datatypes.

[4] Besides cardinailty, the content of the B parameter also determines the domain of infinitary arguments [37].

`A × B → C ≅ A → B → C` Replace multiple arguments of constructors by a single *uncurried* argument. Single argument constructors remain unchanged.

```
data Tree (A B : Set) : Set where
  leaf : A → Tree A B
  branch : Tree A B × B × Tree A B → Tree A B
```

`A × B ≅ B × A` By commutativity of pairs, rearrange inductive constructor arguments to all appear at the end.

```
data Tree (A B : Set) : Set where
  leaf : A → Tree A B
  branch : B × Tree A B × Tree A B → Tree A B
```

`A × B ≅ Π Bool (λ b → if b then A else B)` A non-dependent pair can be defined as a dependent function from a boolean to each component of the pair. Replace all pairs of inductive constructor arguments with such a dependent function whose domain cardinality is equal to the number of inductive arguments for that constructor (i.e., Bool for 2 inductive arguments and ⊥ for 0 inductive arguments).

```
data Tree (A B : Set) : Set where
  leaf : A × (⊥ → Tree A B) → Tree A B
  branch : B × (Bool → Tree A B) → Tree A B
```

`(A → C) ⊎ (B → C) ≅ A ⊎ B → C` Replace the collection of constructors with a single constructor. The new constructor's argument type is the tuple of right-nested disjoint unions formed from the argument types of each old constructor.

```
data Tree (A B : Set) : Set where
  list : (A × (⊥ → Tree A B))
```

```
    ⊎ (B × (Bool → Tree A B))
    → Tree A B
```

(A × B) ⊎ (A' × B') ≅ Σ (A ⊎ A') (λ x → if isLeft x then B else B') Replace the disjoint union of pairs whose domain is non-inductive arguments and codomain is inductive arguments, with a single pair whose domain is the disjoint union of non-inductive arguments and codomain is the disjoint union of inductive arguments.[5]

```
data Tree (A B : Set) : Set where
  list : Σ (A ⊎ B) (λ x → if isLeft x
    then (⊥ → Tree A B)
    else (Bool → Tree A B))
    → Tree A B
```

data ≅ W Encode the final datatype declaration as a W type by using the first component of the pair for the A parameter, and the domains of each function in the second component of the pair for the B parameter.

```
Tree : Set → Set → Set
Tree A B = W (A ⊎ B) λ x → if isLeft x
  then ⊥
  else Bool
```

4.2.3 Inadequacy of Well-Orders

It would seem like W is a sufficient datatype to represent any inductive datatype a user would define. Any *Open Well-Order Type* (i.e., any *open* algebraic type defined using W) can be translated to a *Closed Well-Order Type*, or a value of type ‘Set, by using the sufficient collection of primitive ‘Set constructors.

[5] For datatypes with infinitary arguments, B and B' may depend on A and A' respectively, so the `if` conditional is replaced by `case` analysis.

For example, below we derive closed disjoint unions in terms of closed dependent pairs (‘Σ) and booleans (‘Bool), and then translate the open Tree type to a closed version defined using the closed well-ordering (‘W) type former.

```
_‘⊎_ : ‘Set → ‘Set → ‘Set
A ‘⊎ B = ‘Σ ‘Bool (λ b → if b then A else B)

‘Tree : ‘Set → ‘Set → ‘Set
‘Tree A B = ‘W (A ‘⊎ B) λ x → if (not (proj₁ x))
  then ‘⊥
  else ‘Bool
```

If W were adequate for our purposes, then this thesis could focus on writing fully generic functions over the *Closed Well-Order Type* universe (‘Set) of Section 4.2. We could write functions similar to sum from Section 2.2.3, except they would work for any custom user-defined type! W types can be extended to support definitions of indexed types (Section 2.1.5), which are isomorphic to inductive-recursive (Section 2.1.9) types. However, there is one major issue:

Inadequacy The base cases of inductively defined datatypes using W have an infinite number of intensionally distinct values. Recall that the base case leaf had ⊥ → `Tree A B` as its inductive argument. Because the domain of the function is bottom, we can write it many different ways (i.e., elim⊥, elim⊥ ∘ elim⊥, etc.). Even though all leaves containing such functions are extensionally equivalent, it is inadequate [40] to have an infinite number of intensionally (or, definitionally) distinct canonical forms for the model of Tree (whose initial declaration was first-order).[6]

[6] McBride also explains [40] that W types are inadequate for representing inductive types in Observational Type Theory (OTT) [4], where evidence of extensional equality is internalized in the types of the theory (unlike Extensional Type Theory, where the evidence is at the judgmental level). In OTT, coercion between extensionally equal values requires explicit evidence of the extensional equality, but this evidence is erased when coercing between equal values (rather than neutral terms, and also assuming that the evidence is a value rather than a neutral term). In

W types are inadequate for our purposes because we are interested in dependently typed languages (like Agda) implementing intensional type theory, rather than extensional type theory. For this reason, we represent algebraic datatypes using initial algebra semantics (instead of W types), as covered in Chapter 5. In Chapter 6 we define a universe suitable for modeling closed type theory (i.e., a dependently typed language supporting fully generic programming), using closed initial algebra semantics, and analogous to the *Closed Well-Order Types* universe of Section 4.2.

OTT, the induction principle of an inductive type can be derived from the induction principle of W. However, coercion only erases the equality evidence used in the definition of the derived induction principle for one of the infinitely many base cases of an inductive type (for example, zero as elim⊥ would be erased, but elim⊥ ∘ elim⊥, and subsequent compositions of elim⊥, would not).

Part II

Open Type Theory

Chapter 5

OPEN ALGEBRAIC UNIVERSES

In Section 4.2.2 we derived custom user-defined types as *well-founded trees*, or *well-orderings* (W types). W types can be used to model datatype declarations in a *closed type theory* (Section 4.2.1), without actually extending the metalanguage as done in *open type theory*. Unfortunately, W types are inadequate (Section 4.2.3) models of first-order canonical terms.

In this chapter we present an adequate alternative to modeling datatype declarations, using *initial algebra semantics*. In initial algebra semantics a datatype is modeled as the *least fixed point* (or *fixpoint* for short) of a *pattern functor*. First, we define an initial algebra semantics for datatypes in the language of category theory, denoting types by their *categorical model*. Then, we show the equivalent initial algebra semantics in the language of type theory (as implemented by Agda), denoting types by their *formal model*. We do not fully define the constructions in the categorical model, but rather appeal to widely known concepts to inspire and elucidate the equivalent constructions in the formal model. For example, the syntax of *pattern functors* from the categorical model becomes the type of descriptions (Desc) in the formal model, and the *fixpoint* operator from the categorical model becomes the μ type (parameterized by Desc) in the formal model.

This chapter defines the initial algebra semantics for a series of progressively more expressive classes of datatypes. All formal models in the series are expressed as an *open* universe. The series ends with a formal model for *inductive-recursive* types, which can also be used to model *indexed* types. In Chapter 6 we adapt the formal model of inductive-recursive types as an *open* universe (Section 5.4) to a

closed universe (Section 6.2), suitable for fully generic programming.

Major Ideas The purpose of this chapter is to define the type of fixpoints (μ_1) used to model inductive-recursive types. This fixpoint type is added as a built-in type to our closed universe of user-declared types in Chapter 6, over which we perform fully generic programming in Chapter 7. This chapter reviews initial algebra semantics for datatypes, and does not contain any novel technical contributions. But, we build up to defining inductive-recursive fixpoints by starting from fixpoints for non-dependent types (Section 5.1), then moving to infinitary non-dependent types (Section 5.2), then moving to dependent types (Section 5.3), and finally arriving at fixpoints for inductive-recursive types (Section 5.4).

Our non-technical contribution is relating initial algebra semantics for these progressively more complex classes of datatypes using common terminology, while providing both a categorical and formal model of each class of datatypes. The model of non-dependent types in Section 5.1 is the same as the model given by Norell [48]. We make a minor extension of that model in Section 5.2 to support infinitary types. The model of inductive-recursive types in Section 5.4 is due to Dybjer and Setzer [22, 23]. In Section 5.3 we present a restriction of the model of Dybjer and Setzer to support dependent and infinitary types, but not inductive-recursive types. This restriction is somewhat interesting because its functors are still defined as a sequence of dependent pairs, ending in the unit type, or a "dependent tuple". More conventionally, dependent polynomials are not restricted to such a dependent tuple format. We only use dependent tuple functors for dependent types in Section 5.3 so that the explanation of functors for inductive-recursive types in Section 5.4 progresses naturally from the explanation of functors for dependent types in Section 5.3.

5.1 OPEN NON-DEPENDENT TYPES

In this section we review the initial algebra semantics for *non-dependent* and potentially *inductive* (Section 2.1.3) types. We begin with the categorical model, and then transition to the formal model (i.e., within type theory) by converting abstract mathematical constructs to concrete datatypes (analogous to how we model the abstract notion of a universe as concrete code and meaning function types in Section 2.2.1).[1] Henceforth, when we say "categorical model" or "formal model", we omit clarifying that these models are used as an *initial algebra semantics* of types.

5.1.1 Categorical Model

The categorical model of an inductive datatype is the *least fixed point* of a polynomial equation represented as a *pattern functor* ($F : \mathbf{Set} \to \mathbf{Set}$). The pattern functor is an endofunctor from the category of sets to itself. We are only concerned with the object map of the pattern functor, which maps a **Set** (representing a type) to another **Set**.

The input of the pattern functor (conventionally named X) represents the inductive set being defined, and its output must be a set formed by *polynomial* set constructions. The polynomial set constructions are denoted 1, $(+)$, $(\cdot)$, and X, and represent the unit set, the sum of two sets, the product of two sets, and inductive occurrences of the set. Hence, algebraic datatypes can be encoded as sums-of-products by using pattern functors, where "pattern" means that the functors are restricted to the language of polynomial set expressions.

[1] Here the words "abstract" and "concrete" have their general meanings, not the technical meanings we defined in Section 3.3.

Natural Numbers For example, consider the datatype declaration for the natural numbers.

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

The categorical model of the ℕ type is the following fixpoint equation.

$$\mathbb{N} \triangleq \mu X.\ 1 + X$$

The plus operator $(+)$ represents a choice between constructors, and is analogous to the disjoint union type $(\uplus)$. Thus, above the left summand (1) represents the zero constructor and the right summand (X) represents the suc constructor.

The zero constructor is represented by 1 (analogous to the unit type $\top$) because it lacks arguments (or isomorphically, it has a single trivial argument). The suc constructor is represented by the variable X, indicating that it takes an inductive argument. This is because μ is binding X (in the semantic expression $\mu X.\ 1 + X$) so that it may be used for inductive occurrences of ℕ.

The equation used above is actually a shorthand for explicitly defining a pattern functor $F : \mathbf{Set} \rightarrow \mathbf{Set}$ and obtaining its least fixed point by applying $\mu : (\mathbf{Set} \rightarrow \mathbf{Set}) \rightarrow \mathbf{Set}$.

$$F \triangleq \lambda X.\ 1 + X$$

$$\mathbb{N} \triangleq \mu\ F$$

Consider the notation using μ as a binder to be a shorthand for taking the fixpoint of an anonymous functor obtained by replacing the binding with a λ.

$$\mu X.\ 1 + X \triangleq \mu\ (\lambda X.\ 1 + X)$$

Binary Trees As another example, consider the type of binary trees (parameterized by A and B) containing A's in leaf positions and B's in branch positions (as presented in Section 4.2.2).

```
data Tree (A B : Set) : Set where
  leaf : A → Tree A B
  branch : Tree A B → B → Tree A B → Tree A B
```

The categorical model of the Tree type is the following fixpoint equation.

$$\mathbf{Tree} \triangleq \lambda A.\ \lambda B.\ \mu X.\ A + X \cdot B \cdot X$$

The leaf constructor takes a single argument of type A, so the constructor is represented by A (which is bound by a λ). The branch constructor has two inductive arguments and a non-inductive argument of type B. Thus, its inductive arguments are represented by X (bound by μ) and its non-inductive argument is represented by B (bound by another λ). The multiplication operator $(\cdot)$ represents multiple arguments of a constructor as a conjunction, and is analogous to the pair type $(\times)$. Hence, the multiplication operator is used to define the "products" part of a constructor with multiple arguments, in the sum-of-products representation of datatypes.

5.1.2 Formal Model

To take advantage of a categorical model of initial algebra semantics within type theory, we create a formal model by translating abstract definitions to concrete datatypes and functions. Recall that μ semantically defines a datatype by taking the fixpoint (using μ) of a pattern functor $F : \mathbf{Set} \to \mathbf{Set}$. It is called a *pattern* functor because its "pattern" must be restricted to using the polynomial set constructions covered in Section 5.1.1.

Informally (in the categorical model), we can check that a functor is defined under these restrictions, but in type theory (in the formal model) we must formally

capture these restrictions. We define the formal model by reifying:

1. The *pattern* fragment (enforcing the restriction to a polynomial language) of functors as a datatype (Desc below).
2. The actual pattern *functor* as a computational type family (⟦_⟧ below)
3. The *fixpoint* operator as a datatype (μ below).

Descriptions The first part of our formal model is the type of descriptions (Desc), a syntax for the *pattern* fragment of functors. A Desc is the syntactic reification of the polynomial expression language that must be used for a functor to qualify as a *pattern* functor (i.e., a Desc "describes" a valid, or pattern, functor). Rather than defining a pattern *functor* directly, we first *represent* it as a Desc such that any well-typed description can be *converted* into a functor meeting all pattern restrictions.

Below, the Desc constructors ‘1, ‘X, (‘+), and (‘•) respectively reify a *syntax* for the 1, X, $(+)$, and $(\cdot)$ polynomial set constructions. Of special note is the ‘κ *constant* constructor. The *constant* constructor reifies a syntax for injecting *non-inductive* constructor arguments (such as A in the leaf constructor of Tree). [2]

```
data Desc : Set₁ where
  ‘1 ‘X : Desc
  _‘+_ _‘•_ : Desc → Desc → Desc
  ‘κ : Set → Desc
```

For example, below is the description for the natural numbers datatype.

```
NatD : Desc
NatD = ‘1 ‘+ ‘X
```

Technically, ‘1 is subsumed by ‘κ applied to the unit type (⊤), but we keep

[2] As is often the case with injections, its syntax is implicit (i.e., invisible) when defining pattern functors using polynomial set constructions. For example, the categorical model of trees, using κ for explicit non-inductive arguments, would be $\lambda A.\ \lambda B.\ \mu X.\ \kappa\ A + X \cdot \kappa\ B \cdot X$.

`1 for legibility. For example, natural numbers can equivalently be described as follows.

$$\begin{array}{l} \mathsf{NatD} : \mathsf{Desc} \\ \mathsf{NatD} = \text{`}\kappa\ \top\ \text{`}{+}\ \text{`}\mathsf{X} \end{array}$$

Finally, note that we establish another convention of "quoting" description constructors with a backtick (e.g., `X for X). This emphasizes that they are a syntactic encoding of polynomial set constructions. As we will see, quoting Desc constructors is natural as they also act as codes of a universe (Section 8.2.1).

Pattern Functors The next part of our formal model is the reification of pattern functors ($F : \textbf{Set} \rightarrow \textbf{Set}$) as *type families* (Section 2.1.6) with Set as their domain (F : Set → Set). Rather than defining F directly, we define a *computational type family* (Section 2.1.11) to interpret ($[\![_]\!]$: Desc → Set → Set) the language of polynomial set constructions (Desc) as a pattern functor.

$$\begin{array}{l} [\![_]\!] : \mathsf{Desc} \rightarrow \mathsf{Set} \rightarrow \mathsf{Set} \\ [\![\ \text{`}1\ ]\!]\ X = \top \\ [\![\ \text{`}\mathsf{X}\ ]\!]\ X = X \\ [\![\ A\ \text{`}{+}\ B\ ]\!]\ X = [\![\ A\ ]\!]\ X \uplus [\![\ B\ ]\!]\ X \\ [\![\ A\ \text{`}{\bullet}\ B\ ]\!]\ X = [\![\ A\ ]\!]\ X \times [\![\ B\ ]\!]\ X \\ [\![\ \text{`}\kappa\ A\ ]\!]\ X = A \end{array}$$

By partially applying the interpretation function to a description, we get a model of a *pattern* functor F (rather than an arbitrary non-pattern functor).

$$\forall D.\ [\![\ D\ ]\!] \triangleq \mathsf{F}$$

For example, below we instantiate D to be the description of natural numbers (NatD, defined as `1 `+ `X), and demonstrate the functor produced by partially applying the interpretation function to NatD.

$$[\![\ \mathsf{NatD}\ ]\!] \equiv (\lambda\ X \rightarrow \top \uplus X)$$

Recall that the input to the pattern functor (F : Set → Set) represents the

inductive occurrences of the datatype being modeled. A sound model must rule out pattern functors representing datatypes that are not consistent in type theory, such as *negative* datatypes like Neg of Section 2.1.8.

In Agda, we could *directly* define the functor for Neg to be ($\mathsf{F}\ X = X \to X$), modeling the negative inductive occurrence of Neg in the argument to neg by using X in the domain of the function type. However, defining a fixpoint datatype for such a negative functor would be rejected by Agda's positivity checker, as it would make the language unsound.

Instead, we choose to define functors *indirectly* by partially applying a description to the interpretation function (rather than defining functors *directly* like the one for Neg above). In other words, the output Set of F is only composed of type theory equivalents of polynomial set constructions. For example, the output Set may use disjoint unions ($\uplus$), modeling ($+$), by interpreting the (`$+$) description. It may not use other arbitrary types lacking a polynomial set construction equivalent (because their is no Desc for them), like functions ($\to$) with negative occurrences of X.

Finally, note that it may appear that `κ could be used to inject many non-polynomial types. While this is true, it is not problematic because the type ($A$) that `κ injects must be *non-inductive*. The non-inductivity of A is enforced because A must be a type defined independently of X. In other words, the interpretation of `κ (i.e., $[\![$ `κ A $]\!]\ X = A$) does not pass X to A.

Fixpoints The final part of our formal model is the reification of the least fixed point operator ($\mu : (\mathbf{Set} \to \mathbf{Set}) \to \mathbf{Set}$) for pattern functors. We reify the least fixed point operator (μ : Desc $\to$ Set) as a datatype parameterized by a *description*, rather than a pattern functor ($\mathbf{Set} \to \mathbf{Set}$).

While the categorical model applies the least fixpoint operator directly to a pattern functor ($\mu\ F$), our model instead applies it to a description ($\mu\ D$). The

pattern functor ($\mathbf{Set} \to \mathbf{Set}$) argument of μ can be derived by the model μ by partially applying the interpretation function to the description argument ($[\![D]\!]$: Set $\to$ Set). Below is the datatype declaration for the fixpoint operator (μ), and its constructor (init) is declared shortly thereafter.

```
data μ (D : Desc) : Set where
```

In the categorical model the initial algebra (α_{init}) is used to construct values of the fixpoint of a functor F.

$$\alpha_{\text{init}} : F\ (\mu\ F) \to \mu\ F$$

Applying F to its least fixed point ($\mu\ F$) results in a type isomorphic to its fixpoint. In other words, the **Set** (or Set in the formal model) resulting from $F\ (\mu\ F)$ represents the types of constructors (and the types of their arguments) of $\mu\ F$. Therefore, the formal model of the fixpoint operator (μ) has a single constructor named init (for *initial algebra*), corresponding to α_{init} in the categorical model.

```
init : ⟦ D ⟧ (μ D) → μ D
```

Recall that we model the pattern functor (F) by partially applying ($[\![D]\!]$) the interpretation function to the description of the pattern functor. Additionally, our model of the fixpoint operator applies it to a description ($\mu\ D$), rather than a pattern functor directly. Therefore, the type of the argument to init represents the types of the constructors (and the types of their arguments) for $\mu\ D$.

For example, we can define the type of natural numbers (described by NatD as ‘1 ‘+ ‘X) as follows.

```
ℕ : Set
ℕ = μ NatD
```

Natural numbers are constructed by applying init to values of the following

type.

$$[\![\mathsf{NatD}]\!]\ \mathbb{N} \equiv (\top \uplus \mathbb{N})$$

Finally, notice that descriptions and fixpoints can also be interpreted as a universe (i.e., the universe of open algebraic types) by considering them to be codes ($\mathsf{Desc} : \mathsf{Set}$) and a meaning function ($\mu : \mathsf{Desc} \to \mathsf{Set}$) respectfully.

5.1.3 Examples

Having formally modeled *initial algebra semantics* by reifying parts of the categorical model as datatypes of type theory, now we provide examples of modeling specific type formers and values (using the formal model).

Natural Numbers We have already seen how to encode the type of natural numbers as the disjunction of the unit type and an inductive occurrence.

```
NatD : Desc
NatD = `1 `+ `X

ℕ : Set
ℕ = μ NatD
```

Recall that the type of the argument to the init constructor represents a choice of which constructor to use, and the types of the arguments for the chosen constructor. For the natural numbers, this type specializes as follows.

$$[\![\mathsf{NatD}]\!]\ \mathbb{N} \equiv (\top \uplus \mathbb{N})$$

To model the zero constructor, we choose the left injection of the disjoint union type (defined in Section 2.1.4), and apply it to the trivial unit constructor.

```
zeroArgs : ⊤ ⊎ ℕ
zeroArgs = inj₁ tt
```

To construct a value of a fixpoint (e.g., μ NatD), rather than the meaning

function applied to its fixpoint, we must apply the initial algebra (init). We leave out describing this step explicitly in future exposition.

```
zero : ℕ
zero = init zeroArgs
```

To model the suc constructor, we apply the right injection of disjoint union to the previous natural number (n), given as a function argument.

```
suc : ℕ → ℕ
suc n = init (inj₂ n)
```

There is no need to provide examples of using natural numbers encoded using our formal model. Once we model the type former and constructors according to their standard interface (i.e., their standard type signatures), their usage is indistinguishable from using type formers and constructors defined using datatype declarations (rather than μ).

```
two : ℕ
two = suc (suc zero)
```

The example above expands to the encoded term below, but by using the standard interface of type formers and constructors we do not need to construct it manually.

```
two' : μ (‘1 ‘+ ‘X)
two' = init (inj₂ (init (inj₂ (init (inj₁ tt)))))
```

Similarly, any function defined by pattern matching can retain its standard appearance of pattern matching on declared constructors by using *pattern synonyms*. Pattern synonyms are a notational feature of Agda that expands the left hand syntax to the term on the right hand side. Pattern synonyms can be used in the pattern matching fragment of the language. Thus, by defining pattern synonyms for zero and suc to expand into their init encodings, we can write functions like

plus in a way that is oblivious to the underlying encoding.

```
pattern zero = init (inj₁ tt)
pattern suc n = init (inj₂ n)

plus : ℕ → ℕ → ℕ
plus zero m = m
plus (suc n) m = suc (plus n m)
```

The addition function above expands to the version below, defined by pattern matching on constructors of our encoding (init et al.). The encoding also expands in the body of the function, such as the successor case of the addition function.

```
plus' : μ (‘1 ‘+ ‘X) → μ (‘1 ‘+ ‘X) → μ (‘1 ‘+ ‘X)
plus' (init (inj₁ tt)) m = m
plus' (init (inj₂ n)) m = init (inj₂ (plus' n m))
```

In future examples, we omit examples of non-constructor values and functions defined over modeled types. As explained, once we have derived the type former and constructors of a type using the primitives of our formal model, using the types to define values and function definitions is indistinguishable from using declared types thanks to syntactic conveniences afforded by Agda. Hence, all functions defined over declared types in Section 2.1 can be reused as functions over our formally modeled algebraic types.

Binary Trees The type of binary trees (Section 5.1.1) is modeled by a function taking its parameters (A and B), and returning the description of the disjoint union of A (encoding the leaf constructor), and the triple (as 2 right-nested pairs) consisting of two inductive occurrences and B (encoding the branch constructor).

```
TreeD : Set → Set → Desc
TreeD A B = ‘κ A ‘+ (‘X ‘• (‘κ B ‘• ‘X))

Tree : Set → Set → Set
```

```
Tree A B = μ (TreeD A B)

⟦ TreeD A B ⟧ (Tree A B) ≡ (A ⊎ (Tree A B × (B × Tree A B)))
```

To model the leaf constructor, we apply the left disjoint union injection to the function argument of type A (i.e., the node value for the leaf).

```
leaf : {A B : Set} → A → Tree A B
leaf a = init (inj₁ a)
```

To model the branch constructor, we apply the right disjoint union injection to a triple (2 right-nested pairs). The triple consists of the first inductive function argument (i.e., the left branch), the function argument of type B (i.e., the node value for the branch), and the second inductive function argument (i.e., the right branch).

```
branch : {A B : Set} → Tree A B → B → Tree A B → Tree A B
branch t₁ b t₂ = init (inj₂ (t₁ , (b , t₂)))
```

λ-Calculus Terms We introduce the type of *untyped λ-calculus terms* (Term) as a final and slightly more complex example (i.e., modeling a type with more than 2 constructors). Below we declare the Term type consisting of variable references (var), lambda abstractions (lam), and applications (app).

```
data Term : Set where
  var : (n : ℕ) → Term
  lam : (b : Term) → Term
  app : (f : Term) (a : Term) → Term
```

Our untyped lambda calculus terms use a deBruijn [15] encoding for variables. A deBruijn-encoded term references variables by a natural number index, where 0 refers to the variable bound by the most recent λ (and 1 refers to the next most recent, and so on). For example, below is a high-level syntax for the Church-encoded [8] numeral **one** and its deBruijn-encoded equivalent. In the example,

the term on the left names its variables, while the term on the right uses deBruijn variables, but both terms Church-encode the numeral **one**.

$$\textbf{one} \triangleq \lambda f.\ \lambda x.\ f\ x \triangleq \lambda\ (\lambda\ 1\ 0)$$

As a Term, we write the deBruijn-encoded numeral **one** as follows. Note the applications of the variable constructor (var) to natural numbers (ℕ) to refer to variables by their deBruijn index.[3]

```
one : Term
one = lam (lam (app (var 1) (var 0)))
```

To model Term, we describe the disjoint union of the natural numbers (encoding var) with the disjoint union of an inductive occurrence (encoding lam) and a pair of inductive occurrences (encoding app). This models three constructors using two right-nested disjoint unions.

```
TermD : Desc
TermD = `κ ℕ `+ (`X `+ (`X `• `X))

Term : Set
Term = μ TermD

⟦ TermD ⟧ Term ≡ (ℕ ⊎ (Term ⊎ (Term × Term)))
```

To model the var constructor, we apply the left disjoint union injection to the natural number argument.

```
var : ℕ → Term
var n = init (inj₁ n)
```

To model the lam constructor, we apply the left disjoint union injection to: the

[3] Our Term type is not scope safe in the sense that their could be natural numbers that are out of bounds with respect to the number of lam occurrences. We could index Term by the natural numbers to enforce scope safety, but this additional complexity only makes later examples (of the semantics for Term, already defined using indexed types) harder to read without introducing new concepts.

right disjoint union injection applied to the inductive argument.

```
lam : Term → Term
lam b = init (inj₂ (inj₁ b))
```

To model the app constructor, we apply the left disjoint union injection to: another left disjoint union injection but applied to a pair of inductive arguments.

```
app : Term → Term → Term
app f a = init (inj₂ (inj₂ (f , a)))
```

5.2 OPEN INFINITARY TYPES

In this section we review the initial algebra semantics for *infinitary* (Section 2.1.8) non-dependent types. We extend our previous categorical model, formal model, and examples, to support *infinitary* constructor arguments.

5.2.1 Categorical Model

The categorical model of *infinitary* inductive datatypes reuses the 1, $(+)$ and $(\cdot)$ polynomial set constructions. However, the inductive occurrences construction X is subsumed by the *infinitary* occurrences construction X^A. Functions are the type theoretic equivalent of exponential terms, where X raised to the power of A is equivalent to a function with domain A and codomain X.[4]

$$X^A \triangleq (A \to X)$$

Therefore, X^A is notation for an infinitary polynomial set construction whose domain is A and whose codomain is an inductive occurrence. Any non-infinitary inductive argument X can be isomorphically expressed as an infinitary argument by raising X to the power of 1 (or equivalently, a function whose domain is 1 and whose codomain is X).

[4] If A and X are finite sets, then the cardinality of X^A is equal to the cardinality of the graph of the function $A \to X$.

$$X \cong (X^1 \triangleq 1 \to X)$$

Natural Numbers For example, consider the infinitary (but isomorphic) declaration of the natural numbers below. The inductive argument to the suc constructor has been replaced with the infinitary argument f, using the unit type as its domain.

```
data ℕ : Set where
  zero : ℕ
  suc : (f : ⊤ → ℕ) → ℕ
```

The categorical model of the infinitary ℕ type is the fixpoint equation below.

$$\mathbb{N} \triangleq \mu X.\ 1 + X^1$$

The only difference between the non-infinitary and infinitary ℕ is that constructing it with suc must supply a function ignoring a ⊤ argument, and destructing suc requires applying f to the trivial value tt to access the inductive value in the body of f.

Binary Trees Below is a straightforward infinitary encoding of binary trees, replacing both inductive arguments of branch with infinitary ones by using the unit type as the domain.

```
data Tree (A B : Set) : Set where
  leaf : A → Tree A B
  branch : (f : ⊤ → Tree A B) (b : B) (g : ⊤ → Tree A B) → Tree A B
```

This translates to the categorical model of infinitary binary trees below, without any surprises.

$$\mathbf{Tree} \triangleq \lambda A.\ \lambda B.\ \mu X.\ A + X^1 \cdot B \cdot X^1$$

However, recall our series of isomorphic translations of the binary tree declaration used to model Tree via W types (Section 4.2.2). We can borrow two of those

isomorphisms to transform Tree into a less trivial instance of an infinitary type (i.e., one whose infinitary domains are types other than unit).

First, we reorder the b argument (of type B) to the front via symmetry (`A × B ≅ B × A`), swapping b and the inductive argument t_1 so that both inductive arguments (t_1 and t_2) to appear at the end of branch.

```
data Tree (A B : Set) : Set where
  leaf : A → Tree A B
  branch : (b : B) (t₁ : Tree A B) (t₂ : Tree A B) → Tree A B
```

Second, we appeal to the isomorphism that defines a non-dependent pair (the two arguments t_1 and t_2 above) as a dependent function (f below) from Bool to each component of the pair (`A × B ≅ Π Bool (λ b → if b then A else B)`).

```
data Tree (A B : Set) : Set where
  leaf : A → Tree A B
  branch : (b : B) (f : Bool → Tree A B) → Tree A B
```

This translates both inductive arguments into a *single* infinitary argument, where the domain is now Bool instead of ⊤. It makes sense for the domain (i.e., branching factor) to be Bool, as we are defining *binary* trees. Given that the cardinality of Bool is 2, we use initial algebra semantics to define a categorical model of infinitary binary trees by raising X to the power of 2 in the encoding of the branch constructor.

$$\mathbf{Tree} \triangleq \lambda A.\ \lambda B.\ \mu X.\ A + B \cdot X^2$$

5.2.2 Formal Model

To formally model *infinitary* types, we make minor changes to our previous non-infinitary formal model (Section 5.1.2). In all aspects of our formal model, we change from modeling merely inductive occurrences of types (X) to infinitary occurrences (X^A).

Descriptions Our formal model of descriptions stays the same, except that we replace the syntax for inductive occurrences (‘X) with a syntax for infinitary occurrences (‘X^). While inductive occurrences (‘X) have no arguments, infinitary occurrences (‘X^) have a Set argument representing the domain of the infinitary function type.

```
data Desc : Set₁ where
  ‘1 : Desc
  _‘+_ _‘•_ : Desc → Desc → Desc
  ‘κ ‘X^ : Set → Desc
```

For example, below we convert the suc constructor in the description of natural numbers to take an infinitary argument with a trivial domain.

```
NatD : Desc
NatD = ‘1 ‘+ ‘X^ ⊤
```

Finally, note that the “caret” in the syntax of infinitary occurrences (‘X^) connotes raising an inductive occurrence to some power (the power being the cardinality of the domain argument of type Set).

Pattern Functors Again, pattern functors (F : **Set** → **Set**) are not formally modeled directly. Instead, the formal model of a pattern functor (F : Set → Set) is the result of partially applying the interpretation function to a description (⟦_⟧ : Desc → Set → Set).

The interpretation of all patterns besides the infinitary pattern ‘X^ remains the same. The infinitary pattern ‘X^ A is interpreted as a function with domain A and codomain X. It is crucial that X (representing an inductive occurrence) appears in the codomain (rather than domain) of the function. Otherwise, our subsequent fixpoint construction (μ) would support negative datatypes (the Agda positivity checker prevents us from defining μ with X in the interpreted function domain

even if we tried).

```
⟦_⟧ : Desc → Set → Set
⟦ `1 ⟧ X = ⊤
⟦ A `+ B ⟧ X = ⟦ A ⟧ X ⊎ ⟦ B ⟧ X
⟦ A `• B ⟧ X = ⟦ A ⟧ X × ⟦ B ⟧ X
⟦ `κ A ⟧ X = A
⟦ `X^ A ⟧ X = A → X
```

Partially applying NatD (`1 `+ `X^ ⊤) to the interpretation function results in the following pattern functor for an infinitary encoding of natural numbers.

```
⟦ NatD ⟧ ≡ (λ X → ⊤ ⊎ (⊤ → X))
```

Notice how the argument of the suc constructor, which is the type to the right of the disjoint union, is an function from the unit type to the inductive X occurrence.

Fixpoints The initial algebra semantics for least fixed points (μ : (**Set** → **Set**) → **Set**) of *infinitary* types is formally modeled (μ : Desc → Set) the same way as the non-infinitary version. The init constructor of μ, modeling the initial algebra (α_{init}), is also unchanged.

```
data μ (D : Desc) : Set where
  init : ⟦ D ⟧ (μ D) → μ D
```

The natural numbers can be defined as a fixpoint of their description, as before.

```
ℕ : Set
ℕ = μ NatD
```

The type of the argument to the initial algebra of natural numbers is like the type of the natural number pattern functor, except with X replaced by the type of natural numbers. This makes the argument to the suc constructor an infinitary type, as the codomain ends with an inductive occurrence (the ℕ) type.

```
⟦ NatD ⟧ ℕ ≡ (⊤ ⊎ (⊤ → ℕ))
```

5.2.3 Examples

Now we repeat the examples of formal models of non-infinitary types (Section 5.1.2), converting models to their infinitary counterparts. A straightforward translation from the non-infinitary to the infinitary models *infinitary* versions of both the pattern functors and the exposed datatypes.

Alternatively, we can *expose* a model of *non-infinitary* datatypes that are defined in terms of *unexposed infinitary* pattern functors. In this scenario type formers do not require special treatment (i.e., their definitions can be equivalent to their non-infinitary counterparts). However, we must take special care when modeling constructors by exposing a non-infinitary type signature (i.e., interface) that is defined in terms of an infinitary (hidden, or unexposed) implementation.

Natural Numbers Let's begin with the straightforward model of infinitary natural numbers, defined with a model of an infinitary pattern functor. The infinitary (due to the f argument) definition of natural numbers is below.

```
data ℕ : Set where
  zero : ℕ
  suc : (f : ⊤ → ℕ) → ℕ
```

The infinitary pattern functor for this type is described by NatD. Its type former ℕ appears below, and is modeled the same way as its non-infinitary counterpart in Section 5.1.2.

```
NatD : Desc
NatD = `1 `+ `X^ ⊤

ℕ : Set
```

$\mathbb{N} = \mu\ \mathsf{NatD}$

$[\![\ \mathsf{NatD}\ ]\!]\ \mathbb{N} \equiv (\top \uplus (\top \to \mathbb{N}))$

The model of the zero constructor is also the same as its non-infinitary counterpart.

```
zero : ℕ
zero = init (inj₁ tt)
```

The model of the suc constructor is different, because it takes an infinitary argument (f).

```
suc : (⊤ → ℕ) → ℕ
suc f = init (inj₂ f)
```

But what if we wanted to model the non-infinitary definition of natural numbers below, even though we can only Describe infinitary pattern functors?

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

To expose a model of a non-infinitary type, with an unexposed infinitary pattern functor, we never need to change the type former (so our definition of ℕ above suffices). Because zero was never infinitary to begin with, its previous definition can also be reused.

However, we take special care to expose a model of a non-infinitary suc constructor in terms of its underlying (unexposed) infinitary pattern functor NatD. We expose the non-infinitary type signature of suc, acting as an interface. The implementation of the *infinitary* pattern functor of the formal model is hidden by this interface.

```
suc : ℕ → ℕ
```

```
suc n = init (inj₂ (λ u → n))
```

The implementation ignores the trivial argument u when constructing the predecessor as an infinitary function using the inductive input n.

Binary Trees Our pattern functor for binary trees models the infinitary definition of binary trees below.

```
data Tree (A B : Set) : Set where
  leaf : A → Tree A B
  branch : (b : B) (f : Bool → Tree A B) → Tree A B
```

The description of the binary tree pattern functor, and its type former, are given below.

```
TreeD : Set → Set → Desc
TreeD A B = `κ A `+ (`κ B `• `X^ Bool)

Tree : Set → Set → Set
Tree A B = μ (TreeD A B)

⟦ TreeD A B ⟧ (Tree A B) ≡ (A ⊎ (B × (Bool → Tree A B)))
```

The model of the leaf constructor is straightforward, as it is not infinitary.

```
leaf : {A B : Set} → A → Tree A B
leaf a = init (inj₁ a)
```

However, we model a non-infinitary branch constructor in terms of its underlying infinitary pattern functor. Below the model of the branch constructor is non-infinitary because its type signature does not contain any infinitary arguments (despite the fact that its implementation supplies infinitary values to the initial algebra, defined in terms of an infinitary pattern functor).

```
branch : {A B : Set} → Tree A B → B → Tree A B → Tree A B
```

$$\mathsf{branch}\ t_1\ b\ t_2 = \mathsf{init}\ (\mathsf{inj}_2\ (b\ ,\ (\lambda\ x \to \mathsf{if}\ x\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2)))$$

The second component of the pair (in the right disjoint union injection) is an infinitary function from Bool to Tree A B. Therefore, we simulate a non-infinitary branch by applying a conditional to the boolean argument of the function, returning the inductive t_1 argument in the true case and the inductive t_2 argument in the false case.

Above, we use the infinitary domain of Bool (which is isomorphic to $\top \uplus \top$) to model 2 inductive arguments. In general, the number of inductive arguments can be modeled with an appropriate type according to the pattern below.

0	$\bot$
1	$\top$
2	$\top \uplus \top$
3	$\top \uplus \top \uplus \top$
n	...

5.3 OPEN DEPENDENT TYPES

In this section, we review the initial algebra semantics for *dependent* types. We extend our previous *infinitary* and *non-dependent* categorical model (Section 5.2.1), and formal model (Section 5.2.1), to support constructor argument types that depend on previous constructor arguments.

5.3.1 Categorical Model

Compared to *non-dependent* types, the categorical model's "type signatures" for *pattern functors* ($F : \mathbf{Set} \to \mathbf{Set}$) and *least fixed points* ($\mu : (\mathbf{Set} \to \mathbf{Set}) \to \mathbf{Set}$) remain unchanged in the setting of *dependent* initial algebra semantics. However, we change the language of *polynomial set constructions* to be able to describe pattern functors of types involving dependencies.

We mostly keep the syntax of the non-dependent polynomial set constructions 1, $(+)$, $(\cdot)$, and X. However, the meaning of the product of two sets $(\cdot)$ is actually the *dependent* product (or dependent pair). The syntax of a dependent product uses type ascription (e.g., $(x : A) \cdot B\ x$), allowing the type (B) of the second component of the pair to depend on the value (x) of the first. In contrast, the syntax of a non-dependent product (e.g., $A \cdot B$) does not name the type of the first component of the pair. For example, dependent product can be used to express the set of pairs of natural numbers and finite sets (whose size depends on the natural number first component of the pair).

$$(n : \mathbb{N}) \cdot \mathbf{Fin}\ n$$

While we continue to use the sum of two sets operator $(+)$, it can now be derived using *dependent* $(\cdot)$ rather than be a primitive polynomial set construction. The definition of $(+)$ is derived as the dependent product of a boolean (the 2-element set) and a choice of either subterm.

$$(+) \triangleq \lambda A.\ \lambda B.\ (b : 2) \cdot \mathbf{if}\ b\ \mathbf{then}\ A\ \mathbf{else}\ B$$

We impose an additional restrictions on pattern functors (which are already restricted to contain only positive inductive occurrences) to always end in the unit set 1. That is, pattern functors must take the form of a (possibly empty) sequence of products (of either non-dependent or dependent arguments), ending in 1.[5] For example, below is the product of a dependent natural number, a non-dependent infinitary occurrence, and 1.

$$F \triangleq \lambda X.\ (n : \mathbb{N}) \cdot X^{\mathbf{Fin}\ n} \cdot 1$$

In general, the pattern functor is a (possibly dependent) product of n (possibly 0) sets, ending in a multiplication by the unit set 1. Each of the n sets (i.e.,

[5] Any set A is isomorphic to $A \cdot 1$. This is analogous to any type A being isomorphic to the pair type $A \times \top$, as the unit type only adds trivial (tt) information.

each A_i below) may dependent on the values of previous sets (i.e., each x_i below). Additionally, each A_i may be non-inductive (not using X) or infinitary (using X).

$$F \triangleq \lambda X.\ (x_0 : A_0) \cdot (x_1 : A_1\ x_0) \cdot (x_2 : A_2\ x_1\ x_2) \cdot ... \cdot (x_n : A_n\ x_0\ ...\ x_{n-1}) \cdot 1$$

The purpose of these additional constraints may not be readily apparent now. However, they allow us to seamlessly extend the categorical model of dependent types to include induction-recursion (in Section 5.4).

Finally, note that any use of sums (+) obeys our constraint as long as the left and right subterms obey the constraint. This is because the derived definition of (+) expands to a product.

$$F \triangleq (\lambda X.\ 1 + 1) \triangleq (b : 2) \cdot \textbf{if}\ b\ \textbf{then}\ 1\ \textbf{else}\ 1$$

Natural Numbers We reuse the infinitary definition of the natural numbers from Section 5.2.1.

```
data ℕ : Set where
  zero : ℕ
  suc : (f : ⊤ → ℕ) → ℕ
```

Compared to the infinitary and non-dependent (Section 5.2.1) natural numbers fixpoint, the only difference in our dependent setting is that the suc constructor ends by multiplying by 1 (obeying our constraint).

$$\mathbb{N} \triangleq \mu X.\ 1 + X^1 \cdot 1$$

Technically, the (+) is just notation so the true fixpoint is the expanded definition below.

$$\mathbb{N} \triangleq \mu X.\ (b : 2) \cdot \textbf{if}\ b\ \textbf{then}\ 1\ \textbf{else}\ X^1 \cdot 1$$

Rose Trees We use the infinitary definition of finitely branching rose trees from Section 2.1.8. In this definition of Rose, the list-of-branches argument is isomorphically expressed as a natural number and an infinitary argument with a finite set (whose size is equal to the natural number) as its domain.

```
data Rose (A : Set) : Set where
  rose : A → (n : ℕ) (f : Fin n → Rose A) → Rose A
```

The categorical model of infinitary rose trees must be defined in terms of *dependent* product, as the finite set (Fin n) infinitary domain is dependent on the natural number (n) argument.

$$\textbf{Rose} \triangleq \lambda A.\ \mu X.\ A \cdot (n : \mathbb{N}) \cdot X^{\textbf{Fin}\ n} \cdot 1$$

5.3.2 Formal Model

Our formal model of least fixed points is similar to previous versions. However, formally modeling *dependencies* in pattern functors requires significant changes, especially changes to the structure of pattern functor descriptions.

Descriptions Recall from Section 5.3.1 that we constrained dependent pattern functors to be a sequence of products ending in 1. Recall also that descriptions are the reification (or formal model) of the language used to create legal pattern functors. Hence, we change the type of descriptions to enforce that pattern functors (representing definitions of datatypes) are sequences of dependent pairs (Σ) ending in the unit type (⊤). Now we explain the definition of Desc for dependent algebraic types, and subsequently compare it to the Desc for non-dependent types from Section 5.2.2.

Below (in the definition of Desc), the `ι constructor models the pattern of ending a functor with the unit type. For now, this is simply a renaming of the former

‘1 constructor.[6] The ‘σ constructor models a *dependent* (but non-infinitary, thus also non-inductive) argument. The ‘δ constructor models an *infinitary* (but non-dependent) argument.[7] Thus, while the pattern functor of the categorical model uses a single product ($\cdot$) for any argument, our new description syntax distinguishes between dependent (‘σ) and infinitary non-dependent (‘δ) arguments.

```
data Desc : Set₁ where
  ‘ι : Desc
  ‘σ : (A : Set) (D : A → Desc) → Desc
  ‘δ : (A : Set) (D : Desc) → Desc
```

Compare this with the non-dependent description datatype (Section 5.2.2). The non-dependent pair (‘•) there is replaced by the (no longer infix) dependent pair ‘σ and infinitary non-dependent pair ‘δ. For example, RoseD, defined below, is the description of Rose trees. RoseD uses ‘σ to request a dependent A argument (although the dependency a is unused), then uses ‘σ to request a dependent natural number argument (n), then uses ‘δ to request a non-dependent but infinitary argument (whose domain is Fin n), and finally ends with ‘ι.

```
RoseD : Set → Desc
RoseD A = ‘σ A (λ a → ‘σ ℕ (λ n → ‘δ (Fin n) ‘ι))
```

When ‘σ is used to request an argument of type A, the rest of the description D may depend on a value of A. This is formally modeled by the infinitary type of D, namely A → Desc. Notice that the first argument of the non-dependent pair (‘•) from Section 5.2.2 is a description (Desc), but the first argument of the dependent pair ‘σ is a type (Set). Imagine that A was a description, and that

---

[6] However, in our subsequent extension supporting inductive-recursive types (Section 5.4)), ‘ι gains additional arguments.

[7] At this point it does not make sense for an infinitary argument (‘δ) to be dependent. At the time a datatype is defined, no functions exist that could operate over it. Hence, inductive occurrences need not be dependent arguments because there is no way to use the type being defined yet. However, once we extend descriptions to model inductive-recursive types (Section 5.4) we will need to add a notion of dependency to ‘δ.

D could depend on a value of the inductive type being defined (as the argument to the infinitary domain of D). Then, our type of descriptions (Desc) would be *negative* (and we could subsequently use it to model pattern functors of negative types). Hence, the first component of a dependent pair (A) must be restricted to a Set (guaranteed to be non-inductive) so that the infinitary type D (representing subsequent arguments in the description) remains *positive*.

The infinitary pair constructor `δ is like a specialized combination of the former infinitary constructor `X^ and the non-dependent pair constructor (`•). The $A$ argument represents the domain of the infinitary function (like the argument to `X^), and the non-dependent D argument represents the rest of the description (which cannot depend on the inductive occurrence because the inductive type has not been defined yet).

We can use `σ to derive (`+) as a dependent pair of a boolean and a choice of branches, similar to how we derived sums (+) from dependent products (·) for pattern functors (Section 5.3.1).

```
_`+_ : Desc → Desc → Desc
D `+ E = `σ Bool (λ b → if b then D else E)
```

Additionally, we can derive `κ and `X^ using `σ and `δ respectfully, then immediately ending with `ι (as these derived constructors do not require additional arguments).

```
`κ : Set → Desc
`κ A  = `σ A (λ a → `ι)

`X^ : Set → Desc
`X^ A  = `δ A `ι
```

Finally, we emphasize that (`•) *cannot* be derived from `σ and `δ. It is not clear whether the first argument (a Desc) to (`•) contains an infinitary (hence inductive)

occurrence, so we cannot decide whether to proceed by using `σ (disallowing inductiveness) or `δ (allowing inductiveness). Additionally, we would somehow need to convert the first argument of (`•), a Desc, to the first argument of `σ or `δ, a Set.

Pattern Functors Now we define the interpretation function (⟦_⟧ : Desc → Set → Set) that can be partially applied to descriptions of dependent types to produce formal models (F : Set → Set) of pattern functors (F : **Set** → **Set**) for dependent types. The type signatures of these constructions (⟦_⟧ and F) remains the same when adding dependent arguments, but the implementations change (because the constructors of Desc changed).

```
⟦_⟧ : Desc → Set → Set
⟦ `ι ⟧ X = ⊤
⟦ `σ A D ⟧ X = Σ A (λ a → ⟦ D a ⟧ X)
⟦ `δ A D ⟧ X = (A → X) × ⟦ D ⟧ X
```

We interpret the `ι constructor as the unit type (⊤). We interpret the `σ constructor as a dependent pair (Σ) whose first component is an A, and whose second component is the interpretation of the rest of the description (which *may depend* on the first component). We interpret the `δ constructor as a non-dependent pair (×) whose first component is an infinitary function from A to X (representing an inductive occurrence), and whose second component is the interpretation of the rest of the description (which *may not depend* on the first component).

Partially applying RoseD (along with its parameter A) to the interpretation function results in the following pattern functor for rose trees.

```
RoseD : Set → Desc
```

```
RoseD A = ‘σ A (λ a → ‘σ ℕ (λ n → ‘δ (Fin n) ‘ι))

⟦ RoseD A ⟧ ≡ (λ X → Σ A (λ a → Σ ℕ (λ n → (Fin n → X) × ⊤)))
```

Notice how the A and natural number arguments are interpreted using dependent pairs (Σ), and how the infinitary argument is interpreted using a non-dependent pair ($\times$).

Fixpoints The formal model (μ : Desc → Set) of least fixed points (μ : (**Set** → **Set**) → **Set**) of *dependent* types is unchanged, as is the formal model (init) of the initial algebra (α_{init}).

```
data μ (D : Desc) : Set where
  init : ⟦ D ⟧ (μ D) → μ D
```

As an example, below is the datatype of rose trees defined as a fixpoint.

```
Rose : Set → Set
Rose A = μ (RoseD A)

⟦ RoseD A ⟧ (Rose A) ≡ Σ A (λ a → Σ ℕ (λ n → (Fin n → Rose A) × ⊤))
```

5.3.3 Examples

Now we model the type formers and constructors of (possibly) dependent datatypes. The descriptions of these datatypes are interpreted as models of pattern functors constrained to be sequences of dependent and non-dependent infinitary pairs, ending in the unit type.

Rose Trees We begin by modeling rose trees, because they demonstrate dependencies between argument types while also being simple because they only have a single constructor. First, we repeat the definition of the rose tree description, its

pattern functor, and its type former as a fixpoint.

```
RoseD : Set → Desc
RoseD A = `σ A (λ a → `σ ℕ (λ n → `δ (Fin n) `ι))

Rose : Set → Set
Rose A = μ (RoseD A)

⟦ RoseD A ⟧ (Rose A) ≡ Σ A (λ a → Σ ℕ (λ n → (Fin n → Rose A) × ⊤))
```

Now we model the single constructor (rose) of Rose trees. Note that we are modeling the infinitary rose constructor (Section 2.1.8), rather than its List of roses variant, as indicated by the type signature of our derived rose constructor.

```
rose : {A : Set} (a : A) (n : ℕ) (f : Fin n → Rose A) → Rose A
rose a n f = init (a , (n , (f , tt)))
```

Because our dependent types are modeled as least fixed points of functors constrained to be sequences of pair types, values (e.g., like the rose constructor) are simply the initial algebra of a tuple encoded as a sequence of right-nested pairs (ending in the trivial unit value tt).

Natural Numbers Let's encode a model of natural numbers using descriptions for dependent types. We begin with the pattern functor for a dependent and infinitary encoding of the natural numbers. The zero constructor immediately ends with `ι. The suc constructor uses `δ to demand a trivial (i.e., where the domain is the unit type) infinitary argument (similar to Section 5.2.3), then ends with `ι.

```
NatD : Desc
NatD = `ι `+ `δ ⊤ `ι
```

Recall from Section 5.3.2 that a choice of constructors (`+) is derived as a dependent pair with a boolean domain and a choice between descriptions, so the

NatD above expands to the version below.

```
NatD : Desc
NatD = ‘σ Bool (λ b → if b then ‘ι else ‘δ ⊤ ‘ι)
```

For legibility (especially when describing types with more than 2 constructors), we often create a specialized enumeration type (NatT below) that takes the place of Bool. Then, we define the second argument to ‘σ as a separate function (NatDs below) mapping enumeration tags (representing constructors) to descriptions (representing constructor arguments). For example, we can encode the description of natural numbers by matching on “tags” of the enumeration type NatT.

```
data NatT : Set where
  zeroT sucT : NatT

NatDs : NatT → Desc
NatDs zeroT = ‘ι
NatDs sucT = ‘δ ⊤ ‘ι

NatD : Desc
NatD = ‘σ NatT NatDs
```

By convention, names of tags are suffixed with "T". Tags are merely enumerations and do not have arguments themselves. Rather, we match on tags in descriptions to declare the descriptions of arguments for each constructor (where each constructor is represented by a tag case).

Now let’s finish by modeling the type of natural numbers as a fixpoint, and its constructors as initial algebras of that fixpoint.

```
ℕ : Set
ℕ = μ NatD

zero : ℕ
zero = init (zeroT , tt)

suc : ℕ → ℕ
```

```
suc n = init (sucT , ((λ u → n) , tt))
```

Now we are encoding constructor choices as the initial algebra applied to a dependent pairs whose domain is an enumeration of tags and codomain is the description of arguments for each constructor tag. Hence, the first component (e.g., zeroT or sucT) in the tuple that the initial algebra is applied to is always the tag name.

Finally, note that the same pair constructor (,) is used for both dependent pair arguments (encoded by σ), and non-dependent infinitary pair arguments (encoded by δ). This is because our Agda definition of non-dependent pairs (×) is defined as a special case of dependent pairs (Σ) that ignores its first argument.

λ-Calculus Terms As a final example, we model the untyped λ-calculus terms introduced in Section 5.1.3 using descriptions of dependent types. We will first encode Term using nested booleans for constructor choices, and then repeat the example with named constructor enumeration tags.

Compared to the model of natural numbers, no new concepts are required to encode Terms. However, because Term has 3 constructors, we gain a greater appreciation of the legibility afforded by constructor tags compared to nested constructor choices encoded using booleans. Let's refamiliarize ourselves with the high-level declaration of Term.

```
data Term : Set where
  var : (n : ℕ) → Term
  lam : (b : Term) → Term
  app : (f : Term) (a : Term) → Term
```

First, let's describe the 3 constructors as a right-nested tuple of 3 choices using (‘+). The 1st choice describes var, the 2nd choice describes lam, and the 3rd choice describes app.

```
TermD : Desc
```

```
TermD = `σ ℕ (λ n → `ι) `+ (`δ ⊤ `ι `+ `δ Bool `ι)
```

Let's expand the definition of (`+) to see the nested choices.

```
TermD : Desc
TermD = `σ Bool λ b → if b
  then `σ ℕ (λ n → `ι)
  else `σ Bool λ b → if b
    then `δ ⊤ `ι
    else `δ Bool `ι
```

The var constructor is encoded in the true branch of the first choice, and the lam and app constructors are encoded in a nested choice within the false branch. Below we model the type former and constructors of Term.

```
Term : Set
Term = μ TermD

var : ℕ → Term
var n = init (true , n , tt)

lam : Term → Term
lam b = init (false , true , (λ u → b) , tt)

app : Term → Term → Term
app f a = init (false , false , (λ b → if b then f else a) , tt)
```

Notice how the 2nd and 3rd constructors (lam and app) are both defined as two nested choices, using false as the first pair component, and then another choice (true and false respectively) as their second component. Additionally, we expose an inductive (non-infinitary) model of app (having two non-infinitary Term arguments) using an if to branch on the infinitary Bool domain (as we did for Tree in Section 5.2.3).

Below we repeat the entire Term model, but using constructor tags instead of

nested boolean choices.

```
data TermT : Set where
  varT lamT appT : TermT

TermDs : TermT → Desc
TermDs varT = ‘σ ℕ (λ n → ‘ι)
TermDs lamT = ‘δ ⊤ ‘ι
TermDs appT = ‘δ Bool ‘ι

TermD : Desc
TermD = ‘σ TermT TermDs

Term : Set
Term = μ TermD

var : ℕ → Term
var n = init (varT , n , tt)

lam : Term → Term
lam b = init (lamT , (λ u → b) , tt)

app : Term → Term → Term
app f a = init (appT , (λ b → if b then f else a) , tt)
```

Note how, in the tagged construction, the first component of the pair is always a single tag, hence lam and app are not defined with nested choices.

5.4 OPEN INDUCTIVE-RECURSIVE TYPES

In this section, we extend the initial algebra semantics of *infinitary* and *dependent* types (Section 5.3) to *inductive-recursive* types (Section 2.1.9). An inductive-recursive type is mutually defined with a *decoding* function that may be used in the inductive definition of the type.

5.4.1 Categorical Model

In all of the previous categorical models we have worked with, the pattern functors were *endofunctors* between the category of sets. That is, each functor ($F : \mathbf{Set} \rightarrow \mathbf{Set}$) mapped each set to another set. Consequently, the fixpoint ($\mu : (\mathbf{Set} \rightarrow \mathbf{Set}) \rightarrow \mathbf{Set}$) of such a functor gave us back a set ($\mu\ F : \mathbf{Set}$). Hence, previously each type could be semantically modeled as a set ($\mathbf{Set}$).

To define a categorical model of *inductive-recursive* types, we need to model a type ($X : \mathbf{Set}$) along with its mutually defined *decoding* function ($d : X \rightarrow O$), mapping values of the type to values of some output type ($O : \mathbf{Set}$). For example, Section 2.1.9 presents the type of arithmetic expressions ($X \triangleq \mathsf{Arith}$) mutually defined with a decoding function ($d \triangleq \mathsf{eval} : \mathsf{Arith} \rightarrow \mathbb{N}$) that evaluates an expression to its natural number ($O \triangleq \mathbb{N}$) result. Thus, the categorical model of inductive-recursive sets involves the dependent product of a set and its decoding function. Such a dependent product is called a *slice*, notated as $\mathbf{Set}/O$ (where O is the output set).

$$\mathbf{Set}/O \triangleq (X : \mathbf{Set}) \cdot (X \rightarrow O)$$

Pattern functors for inductive-recursive types are endofunctors ($F : \mathbf{Set}/O \rightarrow \mathbf{Set}/O$) of the slice category $\mathbf{Set}/O$[8], and the fixpoint ($\mu : (\mathbf{Set}/O \rightarrow \mathbf{Set}/O) \rightarrow \mathbf{Set}/O$) of such a pattern functor returns a slice ($\mu F : \mathbf{Set}/O$). It is convenient to separate the definition of F into 2 parts, where we denote the part by a subscript (i.e., F_1 and F_2).

$$F_1 : \mathbf{Set}/O \rightarrow \mathbf{Set}$$

$$F_2 : (R : \mathbf{Set}/O) \rightarrow F_1\ R \rightarrow O$$

The first part (F_1) maps a slice to a set (modeling a *type*), similar to the

[8] Objects of the slice category $\mathbf{Set}/O$ are functions $f : X \rightarrow O$ (where X is some object-specific set and O is a set fixed for the category). Its morphisms are functions $h : X \rightarrow Y$ between objects $f : X \rightarrow O$ and $g : Y \rightarrow O$ such that $f = g \circ h$.

functors of previous subsections. The second part (F_2) maps a slice and a member of the set mapped by F_1, to a member of O (modeling a *decoding* function). By convention we use the letter R to refer to the *slice* argument to distinguish it from the contained set X and decoding function d. We can put these two components of the functor together as a dependent pair to form the actual endofunctor over slices.

$$F : \mathbf{Set}/O \to \mathbf{Set}/O \;\triangleq\; \lambda R.\ (F_1\ R,\ F_2\ R)$$

We can separate the definition of least fixed points to be defined similarly in terms of a fixed point operator (μ_1, returning a set), and its decoding function (μ_2, taking an $\mu_1\ F$ and returning an O).

$$\mu_1 : (\mathbf{Set}/O \to \mathbf{Set}/O) \to \mathbf{Set}$$

$$\mu_2 : (F : \mathbf{Set}/O \to \mathbf{Set}/O) \to \mu_1\ F \to O$$

$$\mu : (\mathbf{Set}/O \to \mathbf{Set}/O) \to \mathbf{Set}/O \;\triangleq\; \lambda F.\ (\mu_1\ F,\ \mu_2\ F)$$

Recall our restriction of pattern functors to a sequence of dependent products of non-inductive or infinitary arguments, terminating in 1.

$$F_1 \triangleq \lambda(X, d).\ (x_0 : A_0) \cdot (x_1 : A_1\ x_0) \cdot (x_2 : A_2\ x_1\ x_2) \cdot ... \cdot (x_n : A_n\ x_0\ ...\ x_{n-1}) \cdot 1$$

Before, it only made sense for non-inductive arguments to be dependent. For example, we could have a functor like the following (where $A : \mathbf{Set}$ and $B : A \to \mathbf{Set}$).

$$F_1 \triangleq \lambda(X, d).\ (x_1 : A) \cdot (x_2 : B\ a) \cdot 1$$

With the introduction of inductive-recursive types, it is now actually possible to use an inductive dependent argument by applying the decoding function (d). Below, we define functors like F in 2 parts, where F_1 defines the first (set) part and F_2 is defines the second (decoding function) part. For example, now we can have a functor like the following (where $A : \mathbf{Set}$ and $B : O \to \mathbf{Set}$).

$$F_1 \triangleq \lambda(X, d).\ (x_1 : X) \cdot (x_2 : B\ (d\ x_1)) \cdot 1$$

Any decoder (F_2) of F_1 has a tuple of arguments similar to the dependencies in the sequence of products defined in F_1 (the only difference is that the tuple ends in the unit argument $\bullet$, corresponding to the unit set 1 that terminates the product). For example, below the arguments x_1 and x_2 in F_2 correspond to the dependencies x_1 and x_2 in F_1 (where $f : (x : X) \to B\ (d\ x) \to O$).

$$F_2 \triangleq \lambda(X, d).\ \lambda(x_1, x_2, \bullet).\ f\ x_1\ x_2$$

Now we finally introduce a new notation that takes advantage of our structure of pattern functors as a sequence of dependent products terminating in 1. The new notation gives us a succinct way to simultaneously define the F_1 and F_2 parts of the pattern functor F by exploiting the shared structure between the dependencies in F_1 and arguments in F_2. Now we define F by terminating the sequence of products with ι (replacing 1) applied to an element of O. Because ι appears at the end of the sequence, it can be defined with access to all of the dependencies of the product that came before it. For example, below we define F directly (where $f : (x : X) \to B\ (d\ x) \to O$).

$$F \triangleq \lambda(X, d).\ (x_1 : X) \cdot (x_2 : B\ (d\ x_1)) \cdot \iota\ (f\ x_1\ x_2)$$

Once again, this is merely notation for directly defining F as a dependent pair (a member of the slice $\mathbf{Set}/O$). Hence, ι is also just notation rather than being a primitive set construction. For example, the notation above expands to the F below (first in terms of F_1 and F_2, and second once the definitions of F_1 and F_2 have been expanded).

$$F \ \triangleq\ \lambda(X, d).\ (F_1\ (X, d),\ F_2\ (X, d))$$

$$F \ \triangleq\ \lambda(X, d).\ ((x_1 : X) \cdot (x_2 : B\ (d\ x_1) \cdot 1,\ \lambda(x_1, x_2, \bullet).\ f\ x_1\ x_2))$$

In general, our new notation for inductive-recursive pattern functors is a sequence of dependent products of non-inductive or infinitary arguments, terminating in ι applied to an element of O, with dependencies x_0 through x_n in scope

(where n is the number of products).

$$F \triangleq \lambda(X, d).\ (x_0 : A_0) \cdot (x_1 : A_1\ x_0) \cdot \ldots \cdot (x_n : A_n\ x_0\ \ldots\ x_{n-1}) \cdot \iota\ (f\ x_0\ \ldots\ x_n)$$

Natural Numbers Any ordinary inductive type can instead be modeled as a trivial inductive-recursive type by combining the inductive type with a trivial decoding function from its values to unit. The inductive type can thus be defined normally, without referring to its trivial function. For example, below we define the type of natural numbers along with the trivial function (point) from natural numbers to unit.[9]

```
data ℕ : Set where
  zero : ℕ
  suc : (⊤ → ℕ) → ℕ

point : ℕ → ⊤
point _ = tt
```

Borrowing from our previous subscript notation for functors and fixpoints, we can rename the inductive definition of ℕ to $\mathbb{N}_1$ and its trivial decoding function point to $\mathbb{N}_2$. Then we can isomorphically model the natural numbers as an inductive-recursive type by combining the type and its decoding function using a pair.

```
data ℕ₁ : Set where
  zero : ℕ₁
  suc : (⊤ → ℕ₁) → ℕ₁

ℕ₂ : ℕ₁ → ⊤
ℕ₂ n = tt
```

[9] The intuition behind the name of the decoding function, point, is that any inhabitant of the function is forced to eventually return tt, the sole inhabitant of the unit type (⊤). Hence, all point functions are extensionally equivalent, as they all "point" to tt. Additionally, the single inhabitant tt of ⊤ can be considered a "point".

```
ℕ : Σ Set (λ A → A → ⊤)
ℕ = ℕ₁ , ℕ₂
```

First we define the categorical model for this trivially inductive-recursive type using the componentized definition of μ in terms of its set (μ_1) and decoding function (μ_2). Below, 1 (similar to ⊤) is the name of the unit set and $\bullet$ (similar to tt) is the name of its single inhabitant.

$$\mathbb{N}_1 \triangleq \mu_1(X, d).\ 1 + X^1 \cdot 1$$

$$\mathbb{N}_2 \triangleq \mu_2(X, d).\ \lambda n.\ \bullet$$

$$\mathbb{N} \triangleq \mu R.\ (\mu_1\ R,\ \ \mu_2\ R)$$

Alternatively, we can define $\mathbb{N}$ directly as a dependent pair where we inline the definition of $\mathbb{N}_1$ into the first component, and inline the definition of $\mathbb{N}_2$ into the second component.

$$\mathbb{N} \triangleq \mu(X, d).\ ((1 + X^1 \cdot 1), (\lambda n.\ \bullet))$$

Finally, we can define it most succinctly with our ι notation as follows.

$$\mathbb{N} \triangleq \mu(X, d).\ \iota \bullet + X^1 \cdot \iota \bullet$$

Because $\iota \bullet$ appears twice, once on either side of $(+)$, the ι-based $\mathbb{N}$ technically models the decoding function ℕ₂ below, which matches against zero and suc but returns tt in either case.

```
ℕ₂ : ℕ₁ → ⊤
ℕ₂ zero = tt
ℕ₂ (suc f) = tt
```

As a final example, consider a pattern functor of the natural numbers that takes advantage of the decoding function (d below) and dependency on an infinitary argument (f below).

$$\mathbb{N} \triangleq \mu(X, d).\ \iota \bullet + (f : X^1) \cdot \iota\ (d\ (f\ \bullet))$$

Above the result of applying the decoding function to a successor of a natural number is specified to be a *recursive call* of the decoding function d applied to: the infinitary predecessor f applied to the unit value $\bullet$. Hence, the pattern above is the categorical model of the decoding function below (notice the recursive call of decoding function $\mathbb{N}_2$ in the suc case).

$\mathbb{N}_2 : \mathbb{N}_1 \to \top$
$\mathbb{N}_2$ zero = tt
$\mathbb{N}_2$ (suc f) = $\mathbb{N}_2$ (f tt)

Now we understand the essence of *induction-recursion*: While the X parameter of the fixpoint operator μ allows us to construct *inductive* arguments, the d parameter allows us to perform *recursive* calls of the decoding function.

5.4.2 Formal Model

In this section we extend the formal model of dependent types (Section 5.3.2) to support inductive-recursive types. The previous description type (Desc), interpretation function ($⟦_⟧$) and least fixed point operator μ are all modified to be parameterized over an output type (O : Set), the codomain of the decoding function.

Descriptions Descriptions (of Section 5.3.2) must be modified to be parameterized over an output type O. Recall that descriptions are the syntactic reification of legal pattern functors. In Section 5.4.1 we presented 3 different ways to define pattern functors for inductive-recursive types.

1. Single pattern functors (F) as a dependent pair.
2. Two-part pattern functors (F_1 and F_2).
3. Single pattern functors (F) using ι.

Our descriptions formally model the syntax of the 3rd (ι) version of legal pattern functors. Recall that ι is applied to an O, hence we had an argument o of type

O to the ‘ι constructor. However, we also change ‘δ in a more subtle way (from Section 5.3.2).

```
data Desc (O : Set) : Set₁ where
  ‘ι : (o : O) → Desc O
  ‘σ : (A : Set) (D : A → Desc O) → Desc O
  ‘δ : (A : Set) (D : (A → O) → Desc O) → Desc O
```

Recall that ‘σ denotes a dependent *non-inductive* argument (of type A) that subsequent arguments, encoded by D, may depend on in. With induction-recursion, ‘δ denotes an infinitary (hence *inductive*) argument (whose domain is A) that subsequent arguments (D) may depend on. However, subsequent arguments in D do not depend directly on an infinitary argument (i.e., $A \to X$). Instead, D depends on a function (i.e., $A \to O$) that is an implicit *composition* of the decoding function and the infinitary function. This implicit composition hides the underlying infinitary argument, preventing an inductive argument (X) from appearing negatively in the domain of the infinitary argument D (instead, O appears). Below is an example of the natural numbers encoded as a trivially (i.e., where the codomain of the decoding function O is the unit type ⊤) inductive-recursive description.[10]

```
NatD : Desc ⊤
NatD = ‘σ Bool (λ b → if b then ‘ι tt else ‘δ ⊤ (λ f → ‘ι (f tt)))
```

In the example above ‘ι tt is returned in the zero branch. The suc branch returns the result of applying the composition (f) of the decoding function and the infinitary function to tt. This describes the definition of natural numbers below.

```
data ℕ₁ : Set where
  zero : ℕ₁
  suc : (⊤ → ℕ₁) → ℕ₁

ℕ₂ : ℕ₁ → ⊤
```

[10] It also happens to be a trivially infinitary type, because ‘δ is applied to ⊤, encoding a trivial infinitary domain.

$$\mathbb{N}_2\ \mathsf{zero} = \mathsf{tt}$$
$$\mathbb{N}_2\ (\mathsf{suc}\ n) = \mathbb{N}_2\ (n\ \mathsf{tt})$$

$$\mathbb{N} : \Sigma\ \mathsf{Set}\ (\lambda\ A \to A \to \top)$$
$$\mathbb{N} = \mathbb{N}_1\ ,\ \mathbb{N}_2$$

To understand where the implicit composition of the decoding function and the infinitary function is happening, recognize that in the successor case of the definitions of NatD and $\mathbb{N}_2$ above, $f = \mathbb{N}_2 \circ n$.

Pattern Functors Now we turn to the task of formally modeling pattern functors ($F : \mathbf{Set}/O \to \mathbf{Set}/O$) of inductive-recursive types. Before we can even consider doing so, we must formally model the concept of a slice $\mathbf{Set}/O$. A slice is formally modeled as a dependent pair type (Σ) parameterized by an output type (O). The first component of the pair is a type and the second component is its decoding function.

$$\mathsf{Set/} : \mathsf{Set} \to \mathsf{Set}_1$$
$$\mathsf{Set/}\ O = \Sigma\ \mathsf{Set}\ (\lambda\ A \to (A \to O))$$

We formally model pattern functors ($F : \mathbf{Set}/O \to \mathbf{Set}/O$) as the functor ($\mathsf{F} : \mathsf{Set/}\ O \to \mathsf{Set/}\ O$) resulting from the partial application of a description to the interpretation function ($[\![_]\!] : \{O : \mathsf{Set}\} \to \mathsf{Desc}\ O \to \mathsf{Set/}\ O \to \mathsf{Set/}\ O$). In Section 5.4.1 we showed the categorical model of F in terms of a component mapping slices to sets (F_1) and a component mapping slices to a decoding function (F_2). Our formal model similarly defines the interpretation function ($[\![_]\!]$) in terms of a type component ($[\![_]\!]_1$) and a decoding function component ($[\![_]\!]_2$), which also result in the pattern functor components (F_1 and F_2) when partially applied to a description.

$$[\![_]\!] : \{O : \mathsf{Set}\} \to \mathsf{Desc}\ O \to \mathsf{Set/}\ O \to \mathsf{Set/}\ O$$

```
⟦ D ⟧ R = ⟦ D ⟧₁ R , ⟦ D ⟧₂ R
```

First, consider the interpretation function component (⟦_⟧₁) mapping slices to types. The ‘ι and ‘σ cases are much like they were for the interpretation function of dependent types in Section 5.3.2.

```
⟦_⟧₁ : {O : Set} → Desc O → Set/ O → Set
⟦ ‘ι o ⟧₁ R = ⊤
⟦ ‘σ A D ⟧₁ R = Σ A (λ a → ⟦ D a ⟧₁ R)
⟦ ‘δ A D ⟧₁ R@(X , d) = Σ (A → X) λ f → ⟦ D (d ∘ f) ⟧₁ R
```

The infinitary ‘δ case now needs to be interpreted as a *dependent* pair type. The left component of the pair is the infinitary argument (f : $A \rightarrow X$). The right component is the interpretation of the description D applied to the *composition* of the decoding function (d) and the *dependent* infinitary argument (f). Thus the subsequent argument types contained in D can depend on the composed function (returning an O), but cannot directly depend on the infinitary function (returning an inductive X).

Before providing an example, we redefine the description of natural numbers by extracting the “if-statement” component into a separate definition. This separate definition (NatDs) returns the description of a particular constructor when applied to the appropriate boolean branch.

```
NatDs : Bool → Desc ⊤
NatDs b = if b then ‘ι tt else ‘δ ⊤ (λ f → ‘ι (f tt))

NatD : Desc ⊤
NatD = ‘σ Bool NatDs
```

To keep the example simple, we look at the result of applying the type component of the interpretation function to the description of the successor constructor

(rather than the entire natural numbers description).

```
⟦ NatDs false ⟧₁ ≡ λ { (X , d) → Σ (⊤ → X) (λ f → ⊤) }
```

The left component of the pair type is the infinitary argument of suc. The right component is just the unit type that terminates every sequence of dependent arguments, ignoring f (the composition of the decoding function and infinitary argument).

Second, consider the interpretation function component ($⟦_⟧_2$) mapping slices to decoding functions. The decoding function works by consuming the arguments (of type $⟦ D ⟧_1\ R$) while recursing down to the `ι base case and returning the o it contains.

```
⟦_⟧₂ : {O : Set} (D : Desc O) (R : Set/ O) → ⟦ D ⟧₁ R → O
⟦ `ι o ⟧₂ R tt = o
⟦ `σ A D ⟧₂ R (a , xs) = ⟦ D a ⟧₂ R xs
⟦ `δ A D ⟧₂ R@(X , d) (f , xs) = ⟦ D (d ∘ f) ⟧₂ R xs
```

The arguments are consumed by applying dependent descriptions (D) to the head argument (a non-inductive a or infinitary f), and recursively consuming the tail (xs). The `σ case recursively searches the subsequent arguments $xs$, which are described by the dependent description ($D$) applied to the non-inductive first component ($a$). The `δ case also searches the subsequent arguments (xs), but they are described by the dependent description (D) applied to the *composition* of the decoding function (d) and the infinitary argument f.

Fixpoints The fixpoint operator ($\mu : (\mathbf{Set}/O \rightarrow \mathbf{Set}/O) \rightarrow \mathbf{Set}/O$) of inductive-recursive types is reified as a *derived* function (μ : {O : Set} → Desc O → Set/ O), parameterized by the output type O and producing slices from descriptions. The pattern functor argument ($\mathbf{Set}/O \rightarrow \mathbf{Set}/O$) of μ can be derived by the formal model of μ by partially applying the interpretation function to the description argument ($⟦ D ⟧$: Set/ O → Set/ O).

In Section 5.4.1 we showed the categorical model of the fixpoint operator μ, defining it in terms of a set component (μ_1) and a decoding function component (μ_2). Our formal model similarly *derives* the fixpoint (μ) as a dependent pair consisting of a type component (μ_1) and a decoding function component (μ_2). We define these 3 constructions (a type synonym μ, a datatype μ_1, and a function μ_2) mutually below.[11]

```
mutual
  μ : {O : Set} → Desc O → Set/ O
  μ D = μ₁ D , μ₂ D

  data μ₁ {O : Set} (D : Desc O) : Set where
    init : ⟦ D ⟧₁ (μ D) → μ₁ D

  μ₂ : {O : Set} (D : Desc O) → μ₁ D → O
  μ₂ D (init xs) = ⟦ D ⟧₂ (μ D) xs
```

The argument to the initial algebra needs to be a type representing constructors (of μ_1, and their arguments). This type is computed by applying the first component ($[\![_]\!]_1$) of the interpretation function to the description (D) and its fixpoint ($\mu\ D$). The output of the decoding function (μ_2) is computed by applying the description (D), its fixpoint ($\mu\ D$), and the argument of the initial algebra (xs) to the second component ($[\![_]\!]_2$) of the interpretation function.

5.4.3 Examples

Now we formally model the type formers and constructors of *inductive-recursive* datatypes. Typically inductive-recursive datatypes are defined mutually in terms of a type and its decoding function. In our formal model, a single description captures definition of *both* the type and its decoding function.

[11] The type μ_1 and the function μ_2 in this section can only be defined by disabling Agda's positivity and termination checkers. In Section 5.4.4, we present an alternative model that need not disable any Agda checkers.

Natural Numbers Let's refamiliarize ourselves with the definition of natural numbers as a trivially inductive-recursive datatype. We use the variant of the inductive-recursive natural numbers where the suc case of decoding function (point) is defined recursively (rather than constantly returning tt).

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

point : ℕ → ⊤
point zero = tt
point (suc n) = point n
```

We expose the formal model of the *non-infinitary* natural numbers presented above. As in Section 5.3.3, this means our type former and constructors will have the *names* and *types* corresponding to the ones above. However, our underlying pattern functor formally models the *infinitary* and *slice-based* definition of natural numbers below.

The non-infinitary type ℕ above corresponds to infinitary type $\mathbb{N}_1$ below. The decoding function point above corresponds to $\mathbb{N}_2$ below. Finally, the slice ℕ below does not correspond to anything above. While slices are commonly used to describe the semantics of inductive-recursive types, they are rarely used in conventional programming with inductive-recursive types.

```
data ℕ₁ : Set where
  zero : ℕ₁
  suc : (⊤ → ℕ₁) → ℕ₁

ℕ₂ : ℕ₁ → ⊤
ℕ₂ zero = tt
ℕ₂ (suc n) = ℕ₂ (n tt)

ℕ : Set/ ⊤
```

```
ℕ = ℕ₁ , ℕ₂
```

Now we specify the pattern functor of the datatype as an inductive-recursive description. We use a datatype of tags (NatT), representing each constructor (as in Section 5.3.2). We also explicitly define the function (NatDs) taking tags to the description of arguments for the constructor that each tag represents.

```
data NatT : Set where
  zeroT sucT : NatT

NatDs : NatT → Desc ⊤
NatDs zeroT = ‘ι tt
NatDs sucT = ‘δ ⊤ (λ f → ‘ι (f tt))

NatD : Desc ⊤
NatD = ‘σ NatT NatDs
```

We model the type (ℕ) and decoding function (point) by applying the type component (μ_1) and decoding function component (μ_2) of the fixpoint operator to the description (NatD). Once again, we are modeling the *non-infinitary* and *slice-less* type of natural numbers in terms of its underlying *infinitary* and *slice-based* pattern functor.

```
ℕ : Set
ℕ = μ₁ NatD

point : ℕ → ⊤
point = μ₂ NatD
```

Finally, we model the constructors. As done previously, the suc constructor creates an infinitary argument as a function ignoring the infinitary domain value (u), and constantly returning the non-infinitary predecessor (n).

```
zero : ℕ
zero = init (zeroT , tt)
```

```
suc : ℕ → ℕ
suc n = init (sucT , (λ u → n) , tt)
```

Arithmetic Expressions Now we model a non-trivially inductive-recursive and non-trivially infinitary type, namely the type of arithmetic expressions (Arith). You may wish to revisit Section 2.1.9 for examples of what arithmetic expressions represent. An Arith can be evaluated to the natural number that the arithmetic expression represents, using the eval decoding function.

```
mutual
  data Arith : Set where
    Num : ℕ → Arith
    Prod : (a : Arith) (f : Fin (eval a) → Arith) → Arith

  eval : Arith → ℕ
  eval (Num n) = n
  eval (Prod a f) = prod (eval a) (λ i → eval (f i))
```

Our pattern functor models the *slice-based* and *infinitary* version of the arithmetic expressions below.

```
mutual
  data Arith₁ : Set where
    Num : ℕ → Arith₁
    Prod : (a : ⊤ → Arith₁) (f : Fin (Arith₂ (a tt)) → Arith₁) → Arith₁

  Arith₂ : Arith₁ → ℕ
  Arith₂ (Num n) = n
  Arith₂ (Prod a f) = prod (Arith₂ (a tt)) (λ i → Arith₂ (f i))

Arith : Set/ ℕ
Arith = Arith₁ , Arith₂
```

The description of the *slice-based* pattern functor is defined in terms of a function (ArithDs) taking arithmetic expression constructor tags (ArithT) to descriptions of the arguments for the constructor that each tag represents.

Compare the index that Fin is applied to in the type Arith_1 above and description ArithDs below. Notice that the dependent infinitary a in the description below represents the composition of the decoding function Arith_2 and the infinitary a above.

```
data ArithT : Set where
  NumT ProdT : ArithT

ArithDs : ArithT → Desc ℕ
ArithDs NumT = `σ ℕ λ n → `ι n
ArithDs ProdT =
  `δ ⊤ λ a →
  `δ (Fin (a tt)) λ f →
  `ι (prod (a tt) f)

ArithD : Desc ℕ
ArithD = `σ ArithT ArithDs
```

Also notice how each description, in the NumT and ProdT cases of ArithDs, ends in `ι. The description prior to `ι represents the type Arith_1 above, and the natural number contained in `ι represents the output of the decoding function $\mathsf{Arith}_2$ above. Finally, the arguments of the decoding function cases are represented by the non-inductive (`σ) and infinitary (`δ) dependencies of the description prior to `ι.

We model the type (Arith) and decoding function (eval) by applying the type component (μ_1) and decoding function component (μ_2) of the fixpoint operator to the description (ArithD).

```
Arith : Set
Arith = μ₁ ArithD

eval : Arith → ℕ
```

```
eval = μ₂ ArithD
```

The same techniques used to model the *non-infinitary* and *slice-less* constructors of the ℕ type are used to model the constructors of the Arith type.

```
Num : ℕ → Arith
Num n = init (NumT , n , tt)

Prod : (a : Arith) (f : Fin (eval a) → Arith) → Arith
Prod a f = init (ProdT , (λ u → a) , f , tt)
```

Vectors Now we show how to *derive* an *indexed* type, like vectors, from a non-trivially *inductive-recursive* type. But first, let's refamiliarize ourselves with the high-level indexed vector definition we wish to derive.

```
data Vec (A : Set) : ℕ → Set where
  nil : Vec A zero
  cons : (n : ℕ) (a : A) (xs : Vec A n) → Vec A (suc n)
```

Before describing the transformation [31] to turn this indexed type into an isomorphic type using induction-recursion, we describe the intuition behind the transformation. A well-known derived (isomorphic) representation of vectors is the dependent pair (Σ) of a List and a constraint on its length, using the equality type (≡).

```
data List (A : Set) : Set where
  nil : List A
  cons : (a : A) (xs : List A) → List A

length : {A : Set} → List A → ℕ
length nil = zero
length (cons a xs) = suc (length xs)

Vec : Set → ℕ → Set
```

```
Vec A n = Σ (List A) (λ xs → length xs ≡ n)
```

While this is a nice and simple translation, it doesn't capture the notion of a vector as intensionally as we would like. Specifically, the cons constructor of List does not a contain the non-inductive natural number argument (n). Additionally, while the outermost derived Vec contains the index constraint ($\equiv$), the inductive List argument (xs) of cons does not.

Instead of deriving Vec from List and length like above, we can use induction-recursion to *mutually* define these 3 components. Induction-recursion allows us to derive an inductive datatype (Vec_1, analogous to List) with the same collection of non-inductive constructor arguments as our high-level indexed Vec, and adds index constraints to go along with every inductive-argument.

```
mutual
  data Vec₁ (A : Set) : Set where
    nil : Vec₁ A
    cons : (n : ℕ) (a : A) (xsq : Vec A n) → Vec₁ A

  Vec₂ : {A : Set} → Vec₁ A → ℕ
  Vec₂ nil = zero
  Vec₂ (cons n x xsq) = suc n

  Vec : Set → ℕ → Set
  Vec A n = Σ (Vec₁ A) (λ xs → Vec₂ xs ≡ n)
```

We transform (as above) a high-level indexed type (like Vec) into a derived version (like Vec), using induction-recursion, by changing 3 things:

1. The original indexed type (Vec) becomes an inductive-recursive type (Vec_1), with a decoding function (Vec_2). The inductive-recursive type (Vec_1) still contains all non-inductive arguments (like n of cons).
2. Original inductive arguments (xs) of the indexed type are replaced by a value (xsq) of a derived dependent pair type (Vec). The first component

of the dependent pair is the inductive-recursive type Vec_1, and the second component constrains the index of the original inductive argument (n) to equal what the decoding function (Vec_2) returns.

3. The decoding function (Vec_2) is defined by matching on the constructors of the inductive-recursive type (Vec_1), and returning what the original high-level indexed type (Vec) had in the index position of the codomain for the corresponding constructor.

Finally, we make one last change, allowing us to formally model the indexed type of vectors using our initial algebra semantics of inductive-recursive types. The inductive-recursive type Vec_1 curries inductive occurrences of the derived dependent pair (Vec) as 2 separate arguments. Below, xsq is replaced by xs (the inductive argument of Vec_1) and q (the constraint). By consequence, the dependent pair Vec no longer needs to be defined mutually.

```
mutual
  data Vec₁ (A : Set) : Set where
    nil : Vec₁ A
    cons : (n : ℕ) (a : A) (xs : Vec₁ A) (q : Vec₂ xs ≡ n) → Vec₁ A

  Vec₂ : {A : Set} → Vec₁ A → ℕ
  Vec₂ nil = zero
  Vec₂ (cons n x xs q) = suc n

Vec : Set → ℕ → Set
Vec A n = Σ (Vec₁ A) (λ xs → Vec₂ xs ≡ n)
```

Now we formally model the *slice-based* pattern functor of the inductive-recursive Vec_1 type.

```
data VecT : Set where
  nilT consT : VecT

VecDs : Set → VecT → Desc ℕ
```

```
VecDs A nilT = `ι zero
VecDs A consT =
  `σ ℕ λ n →
  `σ A λ a →
  `δ ⊤ λ xs →
  `σ (xs tt ≡ n) λ q →
  `ι (suc n)

VecD : Set → Desc ℕ
VecD A = `σ VecT (VecDs A)
```

We model the inductive-recursive type (Vec_1) and decoding function (Vec_2) by applying the type component (μ_1) and decoding function component (μ_2) of the fixpoint operator to the description (VecD).

```
Vec₁ : Set → Set
Vec₁ A = μ₁ (VecD A)

Vec₂ : (A : Set) → Vec₁ A → ℕ
Vec₂ A = μ₂ (VecD A)
```

Finally, we model the indexed type (Vec) as a dependent pair, derived in terms of the inductive-recursive type (Vec_1) and an index constraint using the decoding function (Vec_2).

```
Vec : Set → ℕ → Set
Vec A n = Σ (Vec₁ A) (λ xs → Vec₂ A xs ≡ n)
```

The main thing to notice about the way we model the constructors is that our model of indexed vectors (Vec) is in terms of a dependent pair.

```
nil : {A : Set} → Vec A zero
nil = init (nilT , tt) , refl

cons : {A : Set} → (n : ℕ) (a : A) (xs : Vec A n) → Vec A (suc n)
cons n a (xs , q) = init (consT , n , a , (λ u → xs) , q , tt) , refl
```

Both nil and cons return an inductive-recursive Vec_1 in the first component of

the pair, and an index constraint proof (in terms of Vec_2) in the second component of the pair. Additionally, cons destructs its "inductive" Vec arguments in terms of the underlying pair components xs and q.

5.4.4 Agda Model

In previous sections on non-dependent types (Section 5.1.2), infinitary types (Section 5.2.2), and dependent types (Section 5.3.2), the formal model (i.e., a model in type theory) corresponded to the Agda model (i.e., a model in an implementation of type theory). Unfortunately, this is not the case for the formal model presented for inductive-recursive types in Section 5.4.2.

Although we used Agda syntax in the formal model of Section 5.4.2, we had to turn off the positivity and termination checkers when inductive-recursively defining the fixpoint datatype (μ_1) and its decoding function (μ_2). Even though Agda (the implementation of type theory that we are using) cannot confirm that this definition preserves consistency, Dybjer and Setzer have proven the consistency of the construction in a model of set theory (extended by the Mahlo cardinal) [23].

To pass Agda's positivity and termination checkers, we define the following Agda model as an alternative to the formal model in Section 5.4.2. Our Agda model mutually defines the pattern functor interpretation functions ($[\![_]\!]_1$ for the interpretation of types, and $[\![_]\!]_2$ for the interpretation of decoding functions), along with the inductive-recursive fixpoint type μ_1 and fixpoint decoding function (μ_2).

```
mutual
  ⟦_⟧₁ : {O : Set} (D R : Desc O) → Set
  ⟦ `ι o ⟧₁ R = ⊤
  ⟦ `σ A D ⟧₁ R = Σ A (λ a → ⟦ D a ⟧₁ R)
  ⟦ `δ A D ⟧₁ R = Σ (A → μ₁ R) λ f → ⟦ D (λ a → μ₂ R (f a)) ⟧₁ R

  ⟦_⟧₂ : {O : Set} (D R : Desc O) → ⟦ D ⟧₁ R → O
```

```
⟦ `ι o ⟧₂ R tt = o
⟦ `σ A D ⟧₂ R (a , xs) = ⟦ D a ⟧₂ R xs
⟦ `δ A D ⟧₂ R (f , xs) = ⟦ D (λ a → μ₂ R (f a)) ⟧₂ R xs

data μ₁ {O : Set} (D : Desc O) : Set where
  init : ⟦ D ⟧₁ D → μ₁ D

μ₂ : {O : Set} (D : Desc O) → μ₁ D → O
μ₂ D (init xs) = ⟦ D ⟧₂ D xs
```

Notice that the types of the pattern functor interpretation functions ($⟦_⟧_1$ and $⟦_⟧_2$) have changed. In the type of the interpretation functions, the R argument is now a description (Desc O), instead of a slice (Set/ O). Because R is now a description (rather than a slice), partially applying a description D to the interpretation function ($⟦_⟧$, defined as the dependent pair of $⟦_⟧_1$ and $⟦_⟧_2$ in Section 5.4.2) no longer results in a pattern endofunctor on slices. While we lose some of the beautiful correspondence with our categorical model, we have effectively inlined a specialized version of the interpretation functions that allows Agda to confirm that the type fixpoint component (μ_1) is positive and that the decoding function fixpoint component (μ_2) terminates.

All of our earlier examples of inductive-recursive type encodings (Section 5.4.3) still work. This is because our examples of type formers and constructors only rely on the interfaces exposed by μ_1 and μ_2, so changing their implementations to mutually be defined in terms of $⟦_⟧_1$ and $⟦_⟧_2$ does not break anything.

Finally, this construction of open algebraic types can also be found in Appendix B. In the Appendix, we remove backticks from the Desc constructor names, so that we may distinguish open descriptions from closed descriptions in Chapter 6. We also change the O parameter of μ_1 to be an explicit argument. The primitive types assumed in the construction of Appendix B are defined in Appendix A.

Part III

Closed Type Theory

Chapter 6

CLOSED ALGEBRAIC UNIVERSE

In this chapter[1] we formally model a closed type theory, or dependently typed language, supporting declared datatypes and fully generic programming. The high-level idea is to define a closed type theory, similar to the *Closed Well-Order Types* universe of Section 4.2.1, but replacing W types (Section 4.2.2) with fixpoints (μ_1) of descriptions (formally modeling initial algebra semantics, as in Section 5.4.2). Initial algebra semantics, unlike well-orderings, adequately models declared datatypes in intensional (as opposed to extensional) type theory.

We begin with a naive, failing attempt at defining a closed type theory using fixpoints (Section 6.1). After explaining why the simple but naive attempt actually defines an open (rather than closed) type theory, we explain how to properly close the theory (Section 6.2). Then, we define a procedure to close any type theory (Section 6.3), rather than just the universe we chose for generic programming in this dissertation. Finally, we conclude by comparing and contrasting types and kinds (Section 6.4).

Major Ideas The purpose of this chapter is to define a closed universe that models a dependently typed language supporting user-declared types, so that we may perform fully generic programming over it in Chapter 7. The key to defining the universe is to define a closed universe of built-in types, which includes the type of fixpoints (μ_1) from Section 5.4 as a built-in type. Essentially, we are replacing the W type in the closed universe of Section 4.2 with the fixpoint type μ_1.

[1] This chapter is adapted from work by myself and Sheard [18], as explained in Section 9.4.

Crucially, this requires us to mutually define the universe of closed built-in types (`Set in Section 6.2) with a *closed* equivalent (`Desc in Section 6.2) of the *open* descriptions (Desc) from Section 5.4. This way, the code of closed fixpoints (`$\mu_1$) can take a closed description (`Desc) as its argument, and the closed functor description codes for non-inductive arguments (`σ`) and infinitary arguments (`δ) can take a closed type (`Set) as an argument (for the non-inductive argument type and the non-inductive infinitary domain, respectively).

The closed codes of built-in types (`Set) and the closed codes of functor descriptions (`Desc) both have meaning functions that map the closed codes to their open equivalents. Specifically, the type meaning function (⟦_⟧) maps a closed type `Set to an open type Set, and the description meaning function («_») maps a closed description `Desc to an open description Desc.

6.1 OPEN INDUCTIVE-RECURSIVE TYPES

In this section, we present a naive, failing attempt at creating a *closed* universe using fixpoints. It is a failing attempt because it actually defines an *open* universe. We will define a universe similar to the *Closed Well-Order Types* of Section 4.2.2, but replacing W with μ_1 (of Appendix B), and adding the identity (or equality) type Id. First, let's remind ourselves of the definitions of the identity type, and the type of fixpoints for inductive-recursive definitions.

```
data Id (A : Set) (x : A) : A → Set where
  refl : Id A x x

data μ₁ (O : Set) (D : Desc O) : Set where
  init : ⟦ D ⟧₁ D → μ₁ O D
```

The identity type allows us to state propositionally that two values (x and y) are equal. If they are indeed equal, the constructor refl serves as a proof of the proposition. In previous parts of this dissertation, we used an infix version of the

identity type ($\equiv$), in which the type of the compared values is implicit. Here, we use Id so we can explicitly refer to the type (A) of the compared values. Similarly, above we define a version of the fixpoint operator (μ_1) that explicitly takes the codomain (O) of the inductive-recursive decoding function. The fixpoint operator (μ_1) also takes an explicit description argument (D), as before, where the kind of inductive-recursive descriptions (Desc) is defined in Section 5.4.2.

6.1.1 Formal Model

In the vector example of Section 5.4.3 we saw that *indexed types* can be derived from *inductive-recursive types* and *equality constraints* (i.e., use of identity type). In our universe below, we want to encode indexed types in addition to inductive-recursive types, thus we replace ‘W with ‘μ_1, *and* add ‘Id.

```
mutual
  data ‘Set : Set₁ where
    ‘⊥ ‘⊤ ‘Bool : ‘Set
    ‘Σ ‘Π : (A : ‘Set) (B : ⟦ A ⟧ → ‘Set) → ‘Set
    ‘Id : (A : ‘Set) (x y : ⟦ A ⟧) → ‘Set
    ‘μ₁ : (O : ‘Set) (D : Desc ⟦ O ⟧) → ‘Set

  ⟦_⟧ : ‘Set → Set
  ⟦ ‘⊥ ⟧ = ⊥
  ⟦ ‘⊤ ⟧ = ⊤
  ⟦ ‘Bool ⟧ = Bool
  ⟦ ‘Σ A B ⟧ = Σ ⟦ A ⟧ (λ a → ⟦ B a ⟧)
  ⟦ ‘Π A B ⟧ = (a : ⟦ A ⟧) → ⟦ B a ⟧
  ⟦ ‘Id A x y ⟧ = Id ⟦ A ⟧ x y
  ⟦ ‘μ₁ O D ⟧ = μ₁ ⟦ O ⟧ D
```

Nothing immediately problematic stands out as our universe looks quite like the *Closed Well-Order Types* universe. Let’s take a closer look at why the addition of the identity type (‘Id) is not problematic, but the addition of fixpoints (‘μ_1) is, by constructing values of both. First, we construct the (uninhabited) boolean

proposition that true is equal to false, using the identity type.

```
‘Bottom : ‘Set
‘Bottom = ‘Id ‘Bool true false
```

Above, the proposition (‘Bottom) can be encoded in the universe (i.e., defined as a value of ‘Set) by using the encoded identity type (‘Id, rather than Id). Additionally, the type of the compared values in the proposition can also be encoded in the universe (as ‘Bool, rather than Bool).

Hence, the identity type (Id) can be encoded in the universe using its backtick equivalent (‘Id). Additionally, its *type* argument can be ‘Bool, the backtick universe encoding of type Bool. Next (in Section 6.1.2), we will see that, while the fixpoint type (μ_1) can be encoded in the universe using its backtick equivalent (‘μ_1), its *description* argument cannot be a backtick encoding of a Desc constructor, which is the source of openness of our universe.

6.1.2 Source of Openness

To discover why ‘Set actually defines an *open* universe, let’s try to define the type of natural numbers in the universe (i.e., as a member of ‘Set).

```
NatDs : Bool → Desc ⊤
NatDs true = ι tt
NatDs false = δ ⊤ (λ u → ι tt)

NatD : Desc ⊤
NatD = σ Bool NatDs

‘ℕ : ‘Set
‘ℕ = ‘μ₁ ‘⊤ NatD
```

Above, the type of natural numbers (‘ℕ) and the codomain of the decoding function can be defined *within* the universe (using ‘μ_1 and ‘⊤ respectively, rather than μ_1 and ⊤). However, the description (NatD) of the natural numbers is defined

outside of the universe. This is because σ and δ are respectively applied to the types Bool and ⊤, which are *not* members of the universe ‘Set. Instead, they are types (Set) of our *open* metatheory (Agda). The second argument (D) of encoded fixpoints (‘μ_1) has type Desc, which seems harmless. However, let's inspect the definition of descriptions.

```
data Desc (O : Set) : Set₁ where
  ι : (o : O) → Desc O
  σ : (A : Set) (D : A → Desc O) → Desc O
  δ : (A : Set) (D : (A → O) → Desc O) → Desc O
```

The root of the problem is that the A argument of σ and δ has Agda's type Set, rather than a code of our universe ‘Set. Hence, the universe ‘Set that we defined is actually *open* because ‘μ_1 has an argument D of type Desc, which is an open type because it has Set arguments. There are 2 major consequences resulting from ‘μ_1 having an open type argument (D):

1. Encodings of declared algebraic datatypes can include non-inductive arguments (and decoding codomains) whose types are *not* in the universe ‘Set. For example, a constructor could have a vector (Vec) argument, which is in Agda's open universe Set, rather than our universe ‘Set (that we intended to be closed).
2. We *cannot* write fully generic functions over the universe, which requires defining generic functions that work over any ‘μ_1 applied to any Desc. We would get suck on the σ and δ cases of such functions because we could *not* case-analyze (or recurse into) the A arguments of type Set.

Both of these consequences are a result of Desc being a valid model of algebraic datatype declarations in an *open* universe (where we can use any type, or Set, of the Agda metalanguage to construct a Desc), but not in a *closed* universe (where we need to restrict Desc to only be constructed from closed types, or ‘Sets). We

overcome these problems, by truly defining ‘Set as a *closed* universe (in terms of a closed equivalent of descriptions, named ‘Desc), in the next section.

6.2 CLOSED INDUCTIVE-RECURSIVE TYPES

The key to creating an adequate[2] closed universe of algebraic datatypes in intensional type theory, is paying attention not only to *types* (Set), but also *kinds* (Set_1). Previously, we created the *Closed Vector Types* universe (Section 4.1) and the *Closed Well-Order Types* universe (Section 4.2). In those universes, the kind (Set_1) of types (Set) is the only kind around. Now we create the *Closed Inductive-Recursive Types* universe, where we additionally account for the kind (Set_1) of descriptions (Desc of Appendix B).[3] The lesson to learn is that closing a universe is not only about closing over some collection of *types*, but more generally some collection of *kinds*.

6.2.1 Formal Model

We wish to formally model a closed type theory, supporting user-declared datatypes, within an open type theory (Agda). To do so, we define a type of *closed types* (‘Set), and a meaning function mapping each closed type (‘Set) to an open type (Set) of our model.

We saw in Section 6.1 that descriptions (Desc) are actually *open*. Therefore, to model closed type theory we must also close over descriptions! To do so, we define a type of *closed descriptions* (‘Desc), and a meaning function mapping each

[2] By *adequate* we mean that values of algebraic types have intensionally unique canonical forms. This property is violated when values of algebraic types are encoded using well-orderings, as explained in Section 4.2.3. In contrast, encoding values of algebraic types using descriptions and fixpoints (like the examples in Section 5.4.3) is adequate.

[3] The “type” of types is actually a *kind* because $\text{Set} : \text{Set}_1$. Similarly, the “type” of descriptions is actually a *kind* because $\text{Desc} : (O : \text{Set}) \to \text{Set}_1$. Distinctively, the type former of descriptions is a function. Even though the function domain (O) is a type (Set), descriptions are still kinds because the codomain of the functional type former is a kind (Set_1). In other words, the codomain of a type former determines whether it is a type or a kind, not its domain.

closed description (‘Desc) to an open description (Desc) of our model. Below, we *mutually* define closed types (‘Set) and closed descriptions (‘Desc), and their meaning functions (⟦_⟧ and «_» respectively).

```
mutual
  data ‘Set : Set where
    ‘⊥ ‘⊤ ‘Bool : ‘Set
    ‘Σ ‘Π : (A : ‘Set) (B : ⟦ A ⟧ → ‘Set) → ‘Set
    ‘Id : (A : ‘Set) (x y : ⟦ A ⟧) → ‘Set
    ‘μ₁ : (O : ‘Set) (D : ‘Desc O) → ‘Set

  ⟦_⟧ : ‘Set → Set
  ⟦ ‘⊥ ⟧ = ⊥
  ⟦ ‘⊤ ⟧ = ⊤
  ⟦ ‘Bool ⟧ = Bool
  ⟦ ‘Σ A B ⟧ = Σ ⟦ A ⟧ (λ a → ⟦ B a ⟧)
  ⟦ ‘Π A B ⟧ = (a : ⟦ A ⟧) → ⟦ B a ⟧
  ⟦ ‘Id A x y ⟧ = Id ⟦ A ⟧ x y
  ⟦ ‘μ₁ O D ⟧ = μ₁ ⟦ O ⟧ « D »

  data ‘Desc (O : ‘Set) : Set where
    ‘ι : (o : ⟦ O ⟧) → ‘Desc O
    ‘σ : (A : ‘Set) (D : ⟦ A ⟧ → ‘Desc O) → ‘Desc O
    ‘δ : (A : ‘Set) (D : (o : ⟦ A ⟧ → ⟦ O ⟧) → ‘Desc O)
      → ‘Desc O

  «_» : {O : ‘Set} → ‘Desc O → Desc ⟦ O ⟧
  « ‘ι o » = ι o
  « ‘σ A D » = σ ⟦ A ⟧ (λ a → « D a »)
  « ‘δ A D » = δ ⟦ A ⟧ (λ o → « D o »)
```

Closed fixpoints (‘μ_1) of closed types (‘Set) now take a closed description (‘Desc) as their D argument, compared to Section 6.1, where D was an open description (Desc). Correspondingly, closed non-inductive arguments (‘σ) and closed infinitary arguments (‘δ) of closed descriptions (‘Desc) now take a closed type (‘Set) as their A argument. In contrast, the A argument of σ and δ in the definition of open

descriptions (Desc) is an open type (Set).

Before, the meaning function (⟦_⟧) for closed types only recursed on closed types (‘Set), but now it must mutually recurse using the meaning function («_») for closed descriptions (‘Desc). For example, consider the case of defining the meaning of closed fixpoints (‘μ_1), where ⟦_⟧ is recursively applied to the closed type O, and «_» is recursively applied to the closed description (D).

Conversely, the ‘σ case of the meaning function («_») for closed descriptions (‘Desc) must mutually recurse using the meaning function (⟦_⟧) for closed types (‘Set). For example, consider the case of defining the meaning of non-inductive arguments (‘σ), where «_» is recursively applied to the closed description (D a), and ⟦_⟧ is recursively applied to the closed type A.

Notice that closed descriptions (‘Desc) are *parameterized* by closed types (O of type ‘Set). Take a look at the type of the meaning function («_») for descriptions, mapping a closed description (‘Desc O) to an open description (Desc ⟦ O ⟧). Because open descriptions expect an *open type* (Set) parameter, we must apply the meaning function of types (⟦_⟧) to the closed type O, to ensure that our parameter for open descriptions is well-typed (i.e., is a Set rather than a ‘Set).

Finally, recall that our naive attempt (in Section 6.1) at closing the universe failed because the resulting universe is actually open. In Section 6.1, ‘Set contains a D argument (in ‘μ_1) whose *kind* (Set_1) is Desc. Therefore, ‘Set of Section 6.1 must be a *kind*, to account for the size of its D argument. In contrast, the closed universe of types (‘Set) of this section, and the closed universe of descriptions (‘Desc), are merely *types* (Set). Moreover, a measure of success for closing a universe is the ability to fit it in the size of types (Set) rather kinds (Set_1). The closed universe of algebraic types presented in this section can also be found in Appendix C.

6.2.2 Examples

In Section 5.4.3 we demonstrated various examples of encoding types and constructors using the universe of *open* inductive-recursive types (Section 5.4.2). Now, we repeat these examples in our *closed* universe (Section 6.2).

Datatypes encoded with *open descriptions* (Appendix B) can use any *open type* (Set) for the O parameter of descriptions (Desc), and the A argument of σ and δ. In contrast, *closed descriptions* (‘Desc) may only use *closed types* (‘Set) for the O parameter and A argument.

Natural Numbers We will encode a closed version of the following trivially infinitary and trivially inductive-recursive definition of the natural numbers.

```
data ℕ : Set where
  zero : ℕ
  suc : (⊤ → ℕ) → ℕ

point : ℕ → ⊤
point zero = tt
point (suc f) = point (f tt)
```

Below, we encode the closed description of the natural numbers. Compared to the description in Section 5.4.3, the one below uses the closed type ‘⊤ in the codomain of NatDs and argument to ‘δ, and uses the closed type ‘Bool in the argument to ‘σ.

```
NatDs : Bool → ‘Desc ‘⊤
NatDs true = ‘ι tt
NatDs false = ‘δ ‘⊤ (λ f → ‘ι (f tt))

NatD : ‘Desc ‘⊤
NatD = ‘σ ‘Bool NatDs
```

Below, we define the type former for natural numbers in two parts. First, we define the *code* for the closed type of natural numbers, naming it ‘ℕ and having

type ‘Set. Second, we define the interpretation of the closed code for the natural number type into our open formal model, naming it ℕ and having kind Set. By convention, we prefix closed type formers with a backtick to distinguish them from their interpretation in our open formal model.

```
‘ℕ : ‘Set
‘ℕ = ‘μ₁ ‘⊤ NatD

ℕ : Set
ℕ = ⟦ ‘ℕ ⟧
```

Defining the decoding function point for closed natural numbers amounts to applying the decoding function component μ_2 (from our open model of algebraic types in Section 5.4.2) to an *open* description. Hence, we apply the interpretation function («_») to our closed description (‘Desc) of natural numbers (NatD), translating it to the open description (Desc) expected by μ_2.

```
point : ℕ → ⊤
point = μ₂ « NatD »
```

Defining the constructors for the natural numbers is no different from the open version in Section 5.4.3. While we encode the closed type of natural numbers as ‘ℕ, we also interpret it as the open type ℕ in our formal model. While we *encode types* in a closed way, we can *use values* of the underlying open formal model. That is why constructors (e.g., zero and suc, below) appear no differently than in Section 5.4.3.

```
zero : ℕ
zero = init (true , tt)

suc : ℕ → ℕ
suc n = init (false , (λ u → n) , tt)
```

It is worth pointing out that creating named constructor tags, like the NatT below, is no longer possible in our closed universe. Instead, a choice of constructors

is encoded by applying ‘σ to ‘Bool, in a derived-sum way. Creating named tags like NatT requires extending an *open* theory with the new enumeration type, which is not possible in a *closed* theory.

```
data NatT : Set where
  zeroT sucT : NatT
```

Vectors Next, we will encode a closed version of the trivially infinitary and non-trivially inductive-recursive vectors using the translation from indexed types to inductive-recursive types described in Section 5.4.3. We will encode a closed version of the vector type below.

```
mutual
  data Vec₁ (A : Set) : Set where
    nil : Vec₁ A
    cons : (n : ℕ) (a : A) (xs : Vec₁ A) (q : Id ℕ (Vec₂ xs) n) → Vec₁ A

  Vec₂ : {A : Set} → Vec₁ A → ℕ
  Vec₂ nil = zero
  Vec₂ (cons n x xs q) = suc n

Vec : Set → ℕ → Set
Vec A n = Σ (Vec₁ A) (λ xs → Id ℕ (Vec₂ xs) n)
```

We get the closed description of vectors from the open description in Section 5.4.3 by replacing every instance of an open type with a closed type. For example, the decoding function codomain is the natural numbers, as specified by applying the type of closed descriptions (‘Desc) to the type of closed natural numbers (i.e., ‘ℕ, which we just defined above). Every first argument to ‘σ and ‘δ is a closed type (i.e., one with a backtick). The A parameter of VecDs and Vec is also a closed type (‘Set).

```
VecDs : ‘Set → Bool → ‘Desc ‘ℕ
VecDs A true = ‘ι zero
```

```
VecDs A false =
  ‘σ ‘ℕ λ n →
  ‘σ A λ a →
  ‘δ ‘⊤ λ xs →
  ‘σ (‘Id ‘ℕ (xs tt) n) λ q →
  ‘ι (suc n)

VecD : ‘Set → ‘Desc ‘ℕ
VecD A = ‘σ ‘Bool (VecDs A)
```

Next, we define the *codes* for the closed inductive-recursive vector type (‘Vec_1), its decoding function (‘Vec_2), and the indexed vector type (‘Vec). Again, this mostly involves adding backticks to type arguments.

```
‘Vec₁ : ‘Set → ‘Set
‘Vec₁ A = ‘μ₁ ‘ℕ (VecD A)

‘Vec₂ : (A : ‘Set) → ⟦ ‘Vec₁ A ⟧ → ℕ
‘Vec₂ A = μ₂ « VecD A »

‘Vec : ‘Set → ℕ → ‘Set
‘Vec A n = ‘Σ (‘Vec₁ A) (λ xs → ‘Id ‘ℕ (‘Vec₂ A xs) n)
```

Above, the “length” decoding function (‘Vec_2), and the closed indexed vector type former (‘Vec), take interpreted closed codes as their second arguments. The former does this by applying the type interpretation function (⟦_⟧) to closed inductive-recursive vectors codes (‘Vec_1), while the latter takes closed natural numbers (ℕ), which were also previously defined by applying the type interpretation function (⟦_⟧) to closed natural number codes ($\mathbb{N}_1$).

Additionally, the body of the definition of the closed decoding function (‘Vec_2) must apply the open decoding function fixpoint component (μ_2) to an open description, which it obtains by applying the description interpretation function («_») to the closed description defined by VecD.

Finally, we can define the formal model of closed indexed vectors and their

constructors.

```
Vec : ‘Set → ℕ → Set
Vec A n = ⟦ ‘Vec A n ⟧

nil : {A : ‘Set} → Vec A zero
nil = init (true , tt) , refl

cons : {A : ‘Set} {n : ℕ} (a : ⟦ A ⟧) (xs : Vec A n) → Vec A (suc n)
cons {n = n} a (xs , refl) = init (false , n , a , (λ u → xs) , refl , tt) , refl
```

Above, the types of the vector type former and its constructors are visually similar to the type data declaration (of Vec) presented at the beginning. One key difference is that every open type (Set) is replaced by a closed type (‘Set). While the natural number argument (ℕ) is defined as the *interpretation* of the natural numbers, the type argument (‘Set) remains uninterpreted. Keeping the closed type (‘Set) argument uninterpreted is the key to writing fully generic functions (in Chapter 7) by pattern matching against closed type codes (i.e., the constructors of ‘Set).

Finite Sets Now we give the type of finite sets (Fin) as another example (in addition to Vec) of modeling an open indexed type as an open inductive-recursive type. First, review the high-level open indexed type of finite sets.

```
data Fin : ℕ → Set where
  here : (n : ℕ) → Fin (suc n)
  there : (n : ℕ) (i : Fin n) → Fin (suc n)
```

Using the same procedure to derive indexed vectors from inductive-recursive vectors (in Section 5.4.3), we derive indexed finite sets (Fin) from the inductive-recursive type of finite sets (Fin_1) and its decoding function (Fin_2). The decoding

function computes the index of the codomain of each constructor, from its arguments.

```
mutual
  data Fin₁ : Set where
    here : (n : ℕ) → Fin₁
    there : (n : ℕ) (i : Fin₁) (q : Id ℕ (Fin₂ i) n) → Fin₁

  Fin₂ : Fin₁ → ℕ
  Fin₂ (here n) = suc n
  Fin₂ (there n i q) = suc n

Fin : ℕ → Set
Fin n = Σ (Fin₁) (λ i → Id ℕ (Fin₂ i) n)
```

Converting the open type above to a closed description follows the same rules that we followed to convert open vectors to a closed description. The primary difference is that the description of closed finite sets is not parameterized by a closed type A, because the type of finite sets is not parameterized.

```
FinDs : Bool → `Desc `ℕ
FinDs true = `σ `ℕ λ n → `ι (suc n)
FinDs false =
  `σ `ℕ λ n →
  `δ `⊤ λ i →
  `σ (`Id `ℕ (i tt) n) λ q →
  `ι (suc n)

FinD : `Desc `ℕ
FinD = `σ `Bool FinDs
```

Finally, we define the closed type code components (`Fin₁, `Fin₂, and `Fin) of finite sets. We also define the type former (Fin) and its constructors (here and there) by interpreting closed codes in our open model.

```
`Fin₁ : `Set
`Fin₁ = `μ₁ `ℕ FinD
```

```
‘Fin₂ : ⟦ ‘Fin₁ ⟧ → ℕ
‘Fin₂ = μ₂ « FinD »

‘Fin : ℕ → ‘Set
‘Fin n = ‘Σ ‘Fin₁ (λ i → ‘Id ‘ℕ (‘Fin₂ i) n)

Fin : ℕ → Set
Fin n = ⟦ ‘Fin n ⟧

here : {n : ℕ} → Fin (suc n)
here {n} = init (true , n , tt) , refl

there : {n : ℕ} (i : Fin n) → Fin (suc n)
there {n} (i , refl) = init (false , n , (λ u → i) , refl , tt) , refl
```

Nothing new is required to understand the constructions above. One minor change, compared to the data declaration of indexed finite sets earlier, is that we expose an *implicit* natural number (n) argument in the here and there constructors.

We presented the closed type of finite sets for two reasons. First, as another example of an indexed (but not parameterized) type derived from an inductive-recursive type. Second, our next example is defining closed arithmetic expressions (Arith), which depends on closed finite sets as an argument.

Arithmetic Expressions Now we will close the type of arithmetic expressions (Arith), an example of a non-trivially infinitary and non-trivially inductive-recursive type. All previous examples were trivially infinitary (ℕ, Fin, and Vec). Additionally, arithmetic expressions are "naturally" inductive-recursive, whereas Vec and Fin are indexed types derived from inductive-recursive encodings. First, review the high-level declaration of arithmetic expressions.

```
mutual
  data Arith : Set where
    Num : ℕ → Arith
```

```
    Prod : (a : Arith) (f : Fin (eval a) → Arith) → Arith

  eval : Arith → ℕ
  eval (Num n) = n
  eval (Prod a f) = prod (eval a) (λ i → eval (f i))
```

Below, we define the closed description of arithmetic expressions, which is quite similar to its open description in Section 5.4.3. The interesting difference is that the second `δ encodes the infinitary domain of $f$ to be a *closed* finite set (`Fin). Hence, there is no issue defining closed inductive-recursive types that use closed indexed types derived from closed inductive-recursive types.

```
ArithDs : Bool → `Desc `ℕ
ArithDs true = `σ `ℕ λ n → `ι n
ArithDs false =
  `δ `⊤ λ a →
  `δ (`Fin (a tt)) λ f →
  `ι (prod (a tt) f)

ArithD : `Desc `ℕ
ArithD = `σ `Bool ArithDs
```

Now we can define the closed type of (codes for) arithmetic expressions (`Arith), and its decoding function (eval).

```
`Arith : `Set
`Arith = `μ₁ `ℕ ArithD

eval : ⟦ `Arith ⟧ → ℕ
eval = μ₂ « ArithD »
```

Finally, we define the type former (Arith) and its constructors (Num and Prod) by interpreting closed codes in our open model. In the definition of Prod, we expose a non-infinitary a argument (of type Arith), so its position in the init tuple of arguments is wrapped in a trivially infinitary function that ignores its u argument (of type unit). In contrast, the second argument f is naturally infinitary, hence no

such wrapping is necessary for f, within init.

```
Arith : Set
Arith = ⟦ `Arith ⟧

Num : ℕ → Arith
Num n = init (true , n , tt)

Prod : (a : Arith) (f : Fin (eval a) → Arith) → Arith
Prod a f = init (false , (λ u → a) , f , tt)
```

6.2.3 Kind-Generalized Universes

Because we are claiming that we are formally modeling a closed *universe* (Section 2.2), we must be able to inhabit the type of *codes* and its *meaning* function. A universe (Univ) can be *formally* modeled as a dependent record consisting of a Code type, and a Meaning function mapping codes to types (Set).[4]

```
record Univ : Set₁ where
  field
    Code : Set
    Meaning : Code → Set
```

As expected, we can model the *Closed Inductive-Recursive Types* universe as a member Univ, by using `Set for the codes and ⟦_⟧ for the meaning function.

```
`SetU : Univ
`SetU = record { Code = `Set ; Meaning = ⟦_⟧ }
```

Thus, `SetU is the evidence that *Closed Inductive-Recursive Types* defines a universe, where closed type codes `Set are formally modeled by the kind of open types Set via the meaning function ⟦_⟧. Now that we have defined a closed universe

[4] In Section 2.2, universes are modeled as a dependent pair (Σ) type, where the first component is the type of codes and the second is the meaning function. The Univ record is really just a dependent pair that we have named Univ, and whose components we have named Code and Meaning.

modeled in terms of the kind of open types (Set), can we similarly define a closed universe modeled in terms of our other kind, namely the kind of open descriptions (Desc)?

We can, and we call it the *Closed Inductive-Recursive Descriptions* universe. But first, we must generalize what it means to be a universe. Previously, we defined the Univ record with a Meaning function whose codomain is the kind Set. Now, we define a generalized version where the codomain of the Meaning function is an arbitrary kind (K : Set_1).

```
record Univ (K : Set₁) : Set₁ where
  field
    Code : Set
    Meaning : Code → K
```

We can still define *Closed Inductive-Recursive Types* as a kind-generalized universe by specializing K to the kind Set.

```
‘SetU : Univ Set
‘SetU = record { Code = ‘Set ; Meaning = ⟦_⟧ }
```

However, now we can also define *Closed Inductive-Recursive Descriptions* as a kind-generalized universe by specializing K to the kind Desc.

```
‘DescU : (O : ‘Set) → Univ (Desc ⟦ O ⟧)
‘DescU O = record { Code = ‘Desc O ; Meaning = «_» }
```

Thus, ‘DescU is the evidence that *Closed Inductive-Recursive Descriptions* is a *parameterized* universe (Section 2.2.9), where the parameter O represents the codomain of the decoding function of the closed inductive-recursive algebraic datatypes. If we modeled standard (i.e., not inductive-recursive) dependent algebraic datatypes (like in Section 5.3.2), then this parameter would disappear.

By creating a closed universe of types that includes closed user-declared datatypes modeled using initial algebra semantics, we learn that the standard notion of a universe in type theory can be generalized. A universe normally maps codes to *types*

(Set), but more generally the meaning function can map codes to any *kind*, such as *descriptions* (Desc). This generalization explains why we call ⟦_⟧ the *meaning* function for closed types (‘Set), but also call «_» the *meaning* function for closed descriptions (‘Desc).

6.3 HOW TO CLOSE A UNIVERSE

The closed universe of Section 6.2 is a fine result, because it supports user-declared datatypes, but also fully generic programming (demonstrated in Chapter 7). However, readers may be curious how we arrived at this universe. Perhaps, more importantly, what *procedure* turns an open universe into a closed version? You may want to support fully generic programming over a universe that represents algebraic datatypes with different properties, or uses a different encoding of descriptions, or uses an entirely different style of semantics. We describe a procedure to close a universe below.

6.3.1 Procedure

1. Select a kind K, then mutually:
   (a) Declare a *kind* ‘K, representing (what will be) closed codes of K.
   (b) For each formation rule of K, encode it as a constructor of ‘K.
   (c) Define a meaning function (⟦_⟧) mapping each encoded constructor of ‘K to the actual K formation rule it represents.
2. In the kind former and constructors of ‘K, and in the body of the meaning function (⟦_⟧), simultaneously:
   (a) Replace occurrences of the kind K with its closed encoding ‘K.
   (b) Replace references to A of kind K with the meaning function applied to the reference (⟦ A ⟧).

3. Recursively apply this procedure for another kind J.
   (a) Select J from the arguments of either the kind former, or formation rules, of K.
   (b) In the recursive Step 1, for any K that has already been closed over, implicitly replace K and its references with $`K$ and applications of its meaning function ($[\![_]\!]$).
4. Change $`K$ from a *kind* to a *type*, by replacing Set with $\mathsf{Set}_1$ in the codomain of the kind former of $`K$.

In the procedure above, all closed codes $`K$, and their meaning functions ($[\![_]\!]$), are *mutually* defined. Once the procedure terminates, all closed codes $`K$ will be *types* (Set), rather than *kinds* (Set_1), thanks to **Step 4**.

6.3.2 Example Procedure Run

Typically, we are interested in closing over a universe of types, so our initial K will be the kind of open types (Set), and its formation rules will be some finite collection of *type formers* (e.g., Bool, Id, Σ, μ_1, etc.) Subsequently, other kinds K (e.g., Desc) that we encounter have *constructors* (e.g., ι, σ, and δ) as their formation rules. For example, consider closing over the subset of the kind Set below.

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

data Vec (A : Set) : ℕ → Set where
  nil : Vec A zero
  cons : {n : ℕ} → A → Vec A n → Vec A (suc n)

data μ₁ (O : Set) (D : Desc O) : Set where
  init : ⟦ D ⟧₁ D → μ₁ O D
```

Step 1 We select K to be kind Set, and the *type formers* of the collection of types above are its *formation rules*. Once this step is complete, ‘K is ‘Set (representing what will be closed types), and its meaning function is ⟦_⟧. We present both below.

```
data ‘Set : Set₁ where
  ‘ℕ : ‘Set
  ‘Vec : (A : Set) (n : ℕ) →  ‘Set
  ‘μ₁ : (O : Set) (D : Desc O) → ‘Set

⟦_⟧ : ‘Set → Set
⟦ ‘ℕ ⟧ = ℕ
⟦ ‘Vec A n ⟧ = Vec A n
⟦ ‘μ₁ O D ⟧ = μ₁ O D
```

Step 2 Next, we replace occurrences of Set with ‘Set, and references A of kind Set with ⟦ A ⟧.

```
data ‘Set : Set₁ where
  ‘ℕ : ‘Set
  ‘Vec : (A : ‘Set) (n : ℕ) →  ‘Set
  ‘μ₁ : (O : ‘Set) (D : Desc ⟦ O ⟧) → ‘Set

⟦_⟧ : ‘Set → Set
⟦ ‘ℕ ⟧ = ℕ
⟦ ‘Vec A n ⟧ = Vec ⟦ A ⟧ n
⟦ ‘μ₁ O D ⟧ = μ₁ ⟦ O ⟧ D
```

At this point, our universe is quite like our failing attempt of a closed universe (Section 6.1), because Desc in argument D of ‘μ_1 is not closed yet.

Step 3 Next, we encounter the kind of descriptions (Desc) in the D argument of the ‘μ_1 constructor, so we must recursively apply the procedure by choosing J to be Desc. For the next part of this procedure run, we need to start over at **Step 1** when recursively closing over the kind Desc. However, we will instead call this

Step 3.1, where the **3** prefix indicates that the recursion was initiated by **Step 3** when closing over the kind Set. For reference, we present the kind of descriptions (Desc) below.

```
data Desc (O : Set) : Set₁ where
  ι : (o : O) → Desc O
  σ : (A : Set) (D : A → Desc O) → Desc O
  δ : (A : Set) (D : (A → O) → Desc O) → Desc O
```

Step 3.1 The *constructors* of Desc above are its *formation rules*. Once this step is complete, `J is `Desc (representing what will be closed descriptions), and its meaning function is «_». We present both below.

```
data `Desc (O : `Set) : Set₁ where
  `ι : (o : ⟦ O ⟧) → `Desc O
  `σ : (A : `Set) (D : ⟦ A ⟧ → Desc ⟦ O ⟧) → `Desc O
  `δ : (A : `Set) (D : (⟦ A ⟧ → ⟦ O ⟧) → Desc ⟦ O ⟧) → `Desc O

«_» : {O : `Set} → `Desc O → Desc ⟦ O ⟧
« `ι o » = ι o
« `σ A D » = σ ⟦ A ⟧ D
« `δ A D » = δ ⟦ A ⟧ D
```

Notice that the kind former argument O (of `Desc) *already* has kind `Set (rather than Set) because Set was previously encoded as `Set. Similarly, $A$ arguments of `Desc constructors, and the O argument in the type of the closed descriptions meaning function («_»), *already* have kind `Set. In all three of these places, *and* in the body of the closed descriptions meaning function («_»), references (e.g., $A$) to kinds `Set *already* have the meaning function of closed types applied to them (e.g., ⟦ A ⟧).

Step 3.2 Next, we replace occurrences of Desc with ‘Desc, and references D of kind Desc with « D ».

```
data ‘Desc (O : ‘Set) : Set₁ where
  ‘ι : (o : ⟦ O ⟧) → ‘Desc O
  ‘σ : (A : ‘Set) (D : ⟦ A ⟧ → ‘Desc O) → ‘Desc O
  ‘δ : (A : ‘Set) (D : (⟦ A ⟧ → ⟦ O ⟧) → ‘Desc O) → ‘Desc O

«_» : {O : ‘Set} → ‘Desc O → Desc ⟦ O ⟧
« ‘ι o » = ι o
« ‘σ A D » = σ ⟦ A ⟧ (λ a → « D a »)
« ‘δ A D » = δ ⟦ A ⟧ (λ o → « D o »)
```

Notice that ‘Desc (which replaced Desc in the D arguments of the ‘μ_1, ‘σ, and ‘δ constructors) is applied to O, without the closed types meaning function (⟦_⟧), because the ‘Desc kind former expects a closed type (‘Set).

Additionally, the ‘σ and ‘δ cases of the closed descriptions meaning function («_») now recursively apply «_» to the result of the infinitary function D.

Step 4 Because there are no kinds left to recursively apply the procedure to, **Step 4.1** and **Step 4.2** can be completed by changing closed types (‘Set) and closed descriptions (‘Desc) from *kinds* to *types*. Any kinds that were arguments of the original collection of type formers have been replaced by types, making the final universe closed. Below is the final result of the procedure.

```
mutual
  data ‘Set : Set where
    ‘ℕ : ‘Set
    ‘Vec : (A : ‘Set) (n : ℕ) →  ‘Set
    ‘μ₁ : (O : ‘Set) (D : ‘Desc O) → ‘Set

  ⟦_⟧ : ‘Set → Set
  ⟦ ‘ℕ ⟧ = ℕ
  ⟦ ‘Vec A n ⟧ = Vec ⟦ A ⟧ n
  ⟦ ‘μ₁ O D ⟧ = μ₁ ⟦ O ⟧ « D »
```

```
data 'Desc (O : 'Set) : Set where
  'ι : (o : ⟦ O ⟧) → 'Desc O
  'σ : (A : 'Set) (D : ⟦ A ⟧ → 'Desc O) → 'Desc O
  'δ : (A : 'Set) (D : (⟦ A ⟧ → ⟦ O ⟧) → 'Desc O) → 'Desc O

«_» : {O : 'Set} → 'Desc O → Desc ⟦ O ⟧
« 'ι o » = ι o
« 'σ A D » = σ ⟦ A ⟧ (λ a → « D a »)
« 'δ A D » = δ ⟦ A ⟧ (λ o → « D o »)
```

Reflecting upon how the procedure operates, we come to understand that only *kind* arguments of the original type formers are encoded, and the meaning function is only applied to members of kinds. This explains why the ℕ argument of the vector type former (encoded as 'Vec) did *not* get encoded. Hence, we *did not* create a code (i.e., 'ℕ) and meaning function for the *type* of natural numbers, but we *did* (i.e., 'Desc) for the *kind* of descriptions. Additionally, the type meaning function (⟦_⟧) does not recurse on n in the 'Vec case, nor does the description meaning function («_») recurse on o in the 'ι case, because both n and o are values of a *type* rather than members of a *kind*.

6.4 TYPES VERSUS KINDS

In Section 6.3 we explain how to close over a subset of types, mutually by closing over descriptions. In this section we examine the distinctions between kinds and types in more detail. In particular, we compare and contrast the kind of types (Set : Set_1) and the kind of descriptions (Desc : Set → Set_1), and where they show up (and do not show up) in the universe construction.

6.4.1 Open Types and Kinds

While both types (Set) and descriptions (Desc) are *open* kinds (Set_1), somehow Desc feels "more closed" than Set. We will precisely identify the properties that

cause that feeling, by comparing and contrasting the open *kinds* Set and Desc. First, let's revisit the idea of open *types* from Section 2.1.12.

Lists Consider the type of lists (List below), parameterized by some type A (representing the type of the elements of the list). List is an *open type*, because its collection of values is open. Hence, whenever a new type is declared (in open type theory), it can be used as the A parameter (e.g., by applying List to Bool, Tree, etc.). This is because the kind of the A parameter is Set, the canonical source of openness.

```
data List (A : Set) : Set where
  nil : List A
  cons : (a : A) (xs : List A) → List A
```

While the type of lists (List) is *open*, it is inductively defined by a *closed* collection of constructors. This may seem obvious, but we will return to this point when discussing the difference between the kinds Set and Desc.

Descriptions Now let's consider the kind of descriptions (Desc below), parameterized by some type O, representing the codomain of the inductive-recursive decoding function. Desc is an *open kind*, because its collection of elements is open. Henceforth, we refer to the inhabitants of kinds as *large values* (we could emphasize the distinction with inhabitants of types by calling them *small values*).

```
data Desc (O : Set) : Set₁ where
  ι : (o : O) → Desc O
  σ : (A : Set) (D : A → Desc O) → Desc O
  δ : (A : Set) (D : (A → O) → Desc O) → Desc O
```

Similar to why List is an open type, Desc is an open kind because its collection of large values grows as its O parameter is instantiated to different types, and is used as the type of the o argument in the ι constructor. Another reason why descriptions are an open kind is that the A argument of the σ and δ constructors

have kind Set. Because Desc constructors define elements of a kind, we sometimes call them *large constructors*.

While the kind of descriptions (Desc) is *open*, it is inductively defined by a *closed* collection of constructors. The List type and Desc kind are both *open*, but are defined by a *closed* collection of constructors, because they are both *inductively defined*.

Types Finally, consider the kind of types (Set). The kind of types is open, as it is the canonical source of openness. Unlike Desc, Set is *not* inductively defined. Every time a new type is declared, its type former becomes a new "constructor" of the kind of types (Set). For example, consider the datatype declarations below.

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

data Vec (A : Set) : ℕ → Set where
  nil : Vec A zero
  cons : {n : ℕ} → A → Vec A n → Vec A (suc n)

data μ₁ (O : Set) (D : Desc O) : Set where
  init : ⟦ D ⟧₁ D → μ₁ O D
```

We can split a datatype declaration into 2 parts.

1. The *signature*, containing the type former between data and where keywords.
2. The *body*, containing the constructors after the where keyword.

The formation rules of kind Set are defined by the *signature* part of datatype declarations, but the collection of formation rules is *open* to extension (i.e., whenever a new type is declared). In contrast, the formation rules of kind Desc is defined by the *body* part of its datatype declaration, using the *closed* collection of constructors in the body.

6.4.2 Gratuitous Kinds

An algebraic declaration introduces a *type* into our open theory if the codomain of the *signature* of the declaration is Set. In contrast, an algebraic declaration introduces a *kind* into our open theory if the codomain of the *signature* of the declaration is Set_1.

But what determines that a declaration *needs* to be a kind, as opposed to a type? If all non-inductive arguments of all constructors are classified by types (like $n : \mathbb{N}$, b : Bool, and $a : A$ where A : Set), then we can *choose* to algebraically declare a *type*. For example, lists (List, below) *can* be declared as a *type* because the only non-inductive constructor argument is a of *type* A (where A : Set, and is the parameter of the lists).

```
data List (A : Set) : Set where
  nil : List A
  cons : (a : A) (xs : List A) → List A
```

However, we may choose to declare lists *gratuitously* as a *kind* (List_1, below), even though it is consistent to declare them as a type.

```
data List₁ (A : Set) : Set₁ where
  nil : List₁ A
  cons : (a : A) (xs : List₁ A) → List₁ A
```

If at least one algebraic constructor has an argument that is classified by a *kind* (like A : Set, D : Desc O, etc.), then we *must* algebraically declare a *kind*. For example, heterogenous lists (HList, below) must be declared as a *kind*, because the cons constructor contains argument A of *kind* Set.

```
data HList : Set₁ where
  nil : HList
  cons : {A : Set} → A → HList → HList
```

Finally, we emphasize that both types and kinds can be closed or open, so the type versus kind distinction is *orthogonal* to the closed versus open distinction.

For example, the *type* of parameterized lists (List, above) is *open* (we explain how parameterization makes this possible in Section 6.4.4). On the other hand, below we gratuitously declare *closed* natural numbers ($\mathbb{N}_1$) as a *kind*.

```
data ℕ₁ : Set₁ where
  zero : ℕ₁
  suc : (n : ℕ₁) → ℕ₁
```

6.4.3 Types versus Descriptions

In an open type theory like Agda, Set is a unique kind because it is *not* inductively defined (i.e., it has an open collection of formation rules, extended by type formers in the signature of datatype declarations). Every other kind (like Desc) is defined by a closed collection of formation rules (i.e., the constructors in the body of the datatype declaration for the kind).

The open-versus-closed formation rules distinction between kinds Set and Desc, and the difference in the way the formation rules are defined by datatype declaration signatures or bodies, is what made coming up with an adequate definition of a closed universe difficult. In fact, we first defined an inadequate definition of a closed universe [18], where certain algebraic types (like natural numbers) were (adequately) encoded in the first universe, but others (like parameterized lists) needed to be (inadequately) lifted to the second universe.

The solution to defining a closed universe adequately (as in Section 6.2, following the procedure in Section 6.3) is to create codes for types (‘Set) *mutually* with codes for descriptions (‘Desc). At first, this may seem odd because descriptions (Desc) can already be viewed as codes (whose interpretation function is the fixpoint operator μ_1). Hence, ‘Desc can be viewed as a code for codes. However, this is necessary because Desc codes are *open descriptions*, while ‘Desc codes are *closed descriptions*.

Part of what led us to realizing that codes for closed types (‘Set) need to

be defined mutually with codes for closed descriptions (‘Desc), was viewing the “constructors” of kinds Set and Desc in a unifying way as kind formers. Rather than focusing on the syntactic difference that a type former (of Set) appears in the signature of a declaration, and a large constructor (of Desc) appears in the body of a declaration, we can simply focus on the fact they are both formation rules of some kind. For example, below we list the formation rules for the kind Set and the kind Desc in a unified way.

$\mathsf{Set} : \mathsf{Set}_1$	$\mathsf{Desc} : (O : \mathsf{Set}) \to \mathsf{Set}_1$
$\mathbb{N} : \mathsf{Set}$	$\iota : (o : O) \to \mathsf{Desc}\ O$
$\mathsf{Vec} : (A : \mathsf{Set})\ (n : \mathbb{N}) \to \mathsf{Set}$	$\sigma : (A : \mathsf{Set})$ $(D : A \to \mathsf{Desc}\ O) \to \mathsf{Desc}\ O$
$\mu_1 : (O : \mathsf{Set})\ (D : \mathsf{Desc}\ O) \to \mathsf{Set}$	$\delta : (A : \mathsf{Set})$ $(D : (A \to O) \to \mathsf{Desc}\ O) \to \mathsf{Desc}\ O$

The first row contains the *kind formers* Set and Desc. Subsequent rows in the first column contain *type formers*, serving a role analogous to large constructors, but for kind Set. Subsequent rows in the second column contain the large constructors of kind Desc. This unified way of presenting Set and Desc leads us to refer to large constructors of Desc as *description formers*.

6.4.4 Kind-Parameterized Types

To perform fully generic programming, our original goal was to create a closed universe of *types*. This universe corresponds to the first universe in a hierarchy of universes (we define the hierarchy in Chapter 8). For the first universe to be adequate, it should contain all possible *small values*. In other words, ‘Set should encode *types* like ‘Bool, ‘Σ, and ‘Vec, whose elements are small values. However, it should *not* encode *kinds* like ‘Set and ‘Desc, whose elements would be large values. Encoding large values in the first universe leads to inconsistency due to a *type in type* paradox [28, 34].

If ‘Desc should *not* be encoded in our first universe, then why do we need to close over it when defining our universe at all? The answer is that the *kind* Desc appears as an argument to the *type* former of μ_1. This is similar to how the *kind* Set appears as an argument to the *type* former of Vec. However, this leads us to the next question: why can a type like Vec have a kind-level type former argument (i.e., its parameter A of kind Set) while remaining a type itself (rather than being lifted to a kind)? The answer has to do with both Vec and μ_1 being defined as *kind-parameterized* types.

Vectors Consider the type of vectors, parameterized by elements of some type A, and indexed by the natural numbers.

```
data Vec (A : Set) : ℕ → Set where
  nil : Vec A zero
  cons : {n : ℕ} → A → Vec A n → Vec A (suc n)
```

Vectors are *types*, rather than *kinds*, because the codomain of their type former is Set (rather than Set_1). An algebraic datatype can consistently be classified as a *type* so long as its constructors do not contain a kind (e.g., Set) as a formal argument. Datatype *parameters* give us a way to refer to A (of kind Set) in the vector constructors, without actually taking A as a formal argument in the *declaration* of each constructor. Hence, the declarations of the nil and cons constructors do not have an A argument. However, if we consider the types of the constructors (rather than their declarations), we see that A appears as an informal argument to each constructor.

```
nil : {A : Set} → Vec A zero
cons : {A : Set} {n : ℕ} (a : A) → Vec A n → Vec A (suc n)
```

Notice that the cons constructor must take n as a formal argument so that it may determine the index to be suc n. We call A an informal argument because the underlying constructor declaration does not store the type A (even though cons

does store the value a of type A, because a is not a parameter of Vec). It is exactly this fact, that the declaration of the nil and cons constructors do not formally store A (of kind Set) as an argument, that allows Vec to be a type (Set) rather than a kind (Set₁). To see the difference, we define vectors to be indexed by A, rather than parameterized by A, below.

```
data Vec : (A : Set) → ℕ → Set₁ where
  nil : {A : Set} → Vec A zero
  cons : {A : Set} {n : ℕ} → A → Vec A n → Vec A (suc n)
```

The type former Vec declares A to be an index because it appears to the right of the colon in the datatype declaration signature (appearing to the left makes it a parameter). Now the nil and cons constructors must take A as a formal argument, because it is no longer available as a parameter. Because A is a *kind* (Set), and it appears as formal constructor arguments, the indexed vector type must be a kind (hence, the codomain of its former is Set_1).

Fixpoints Now let's reconsider the definition of the *type* of fixpoints, *parameterized* by the decoding codomain O and the description D. Below, we only present the definition of the interpretation function ($⟦_⟧_1$) and the fixpoint datatype (μ_1). For the full definition, including the decoding function, see Section 5.4.2.

```
⟦_⟧₁ : {O : Set} (D R : Desc O) → Set
⟦ ι o ⟧₁ R = ⊤
⟦ σ A D ⟧₁ R = Σ A (λ a → ⟦ D a ⟧₁ R)
⟦ δ A D ⟧₁ R = Σ (A → μ₁ _ R) λ f → ⟦ D (μ₂ R ∘ f) ⟧₁ R

data μ₁ (O : Set) (D : Desc O) : Set where
  init : ⟦ D ⟧₁ D → μ₁ O D
```

Both parameters (O and D) of the fixpoint datatype (μ_1) are kinds (Set and Desc, respectively). Hence, μ_1 can be a type (Set), rather than a kind (Set_1), because its constructor (init) does not contain any formal arguments that are classified

by kinds. While the type parameter (O) is used similarly to the type parameter (A) of vectors, the description parameter (D) is used in a significant way. The interpretation function ($[\![_]\!]_1$) is applied to the D parameter to compute the *type* of the argument to init. While $[\![_]\!]_1$ takes a *kind* (D) as an input, it returns a type as an output. Hence, init never actually stores a description (i.e., a kind) as a formal argument.

We discuss the significance of computing over a *large* (i.e., a *kind*) parameter in a constructor argument of a *type* in Chapter 10. The consequence is that fixpoints can be defined as a *type*, hence they model algebraic datatypes as types, whose inhabitants are (small) *values*. It would be inadequate to model algebraic datatypes (like natural numbers or vectors) at the level of kinds, because users expect to declare them as types. Significantly, by defining closed descriptions (\`Desc) mutually with closed types (\`Set), we preserve the adequate encoding of \`$\mu_1$ as a closed *type*, allowing our formal model of closed algebraic datatypes (like in \`$\mathbb{N}$ and \`Vec in Section 6.2.2) to adequately classify small values (like \`zero and \`nil).

Heterogenous Lists We have learned that certain datatypes can be declared as types, rather than kinds, by changing datatype indices to datatype parameters. However, if a datatype is not indexed, then this change is not applicable, and the type must be declared as a kind. For example, consider the *kind* of heterogenous lists (HList below).

```
data HList : Set₁ where
  nil : HList
  cons : {A : Set} → A → HList → HList
```

Because the cons constructor of heterogenous lists takes A of kind Set as a formal argument, there is no choice but to make HList a kind (Set_1). We could imagine indexing HList by the collection of types it contains, and then using our trick to turn the index into a parameter. However, this would not adequately define

heterogenous lists, because the types of elements would be statically determined. For similar reasons, the descriptions (Desc) must be a kind, rather than a type.

The first closed universe (of Section 6.2) cannot encode kinds like Set, Desc, and HList. However, Chapter 8 defines a closed hierarchy of universes, allowing kinds to be represented in the next (i.e., second) universe (i.e., the universe of closed kinds). Further levels of the universe correspond to closed superkinds (Set_2), and so on (Set_3, Set_4, ... , Set_ω).

Chapter 7

FULLY GENERIC FUNCTIONS

In this chapter[1] we formally model fully generic programming in a closed dependently typed language. We write fully generic functions in the universe of Section 6.2, supporting user-declared datatypes while remaining closed.

Thus far we have focused on defining concrete datatypes in our universe of (inductive-recursive) algebraic types. *Smart constructors* (defined as functions, first demonstrated in Section 5.1.3), for the type former and constructors of a concrete algebraic datatype, allow us to *construct* concrete types and their values while hiding their generic encoding in terms of initial algebra semantics. Similarly, *pattern synonyms* (demonstrated in Section 5.1.3), for constructors of concrete types encoded using initial algebra semantics, allow us to *deconstruct* generically encoded values by writing functions defined by pattern matching while hiding underlying algebraic encodings.

While smart constructors and pattern synonyms shelter users from generic encodings when they construct and deconstruct *concrete* datatypes, fully generic programming requires users to understand how to generically construct and deconstruct *encoded* datatypes, by applying and matching against the initial algebra constructor of μ_1. By definition, fully generic functions can be applied to (and may return) values of any user-declared type, thus understanding the underlying generic encoding (or something isomorphic to it) is necessary. In this chapter we define three fully generic functions:

[1] This chapter is adapted from work by myself and Sheard [18], as explained in Section 9.4.

1. count, in Section 7.1, counting the number of nodes in a generically encoded value.
2. lookup, in Section 7.2, looking up any subnode of a generically encoded value.
3. ast, in Section 7.3, marshalling any generically encoded value to an abstract syntax tree (AST), defined as a rose tree.

Major Ideas The purpose of this chapter is to demonstrate examples of fully generic programming over the universe defined in Section 6.2 (which also appears in Appendix C). Traditional generic programs (as explained in the introduction of Chapter 1) only recurse into inductive constructor arguments. We could write a traditional generic size function, like the one in Section 1.2.1, over the open universe of inductive-recursive types in Section 5.4.4. We could also write other traditional generic functions that only need to recurse into inductive constructor arguments, such as map and fold.

In contrast, this chapter focuses on writing fully generic programs, like the count function of Section 1.2.2. Fully generic programs can recurse into *both* the inductive and non-inductive arguments of constructors. In Section 7.1, we define a fully generic count function over the closed universe of Section 6.2, modeling a dependently typed language supporting user-declared types. Functions that marshal data into another format (such as binary, JSON, XML, etc.) are a prime example of fully generic programming. When marshaling data, it is not enough to marshal just the *inductive structure* of values, we also want to marshal all of the *non-inductive* values contained in the structure.

Section 7.3 features a fully generic function (ast) that marshals data into a common rose tree format, used to visualize values with Graphviz [25]. The primary thing to notice in this chapter is that the definitions of generic functions recurse into non-inductive arguments. This includes recursion into both components of the built-in type of pairs (in the `Σ case of closed built-in types `Set), and recursion into

non-inductive constructor arguments (in the ‘σ case of closed functor descriptions ‘Desc).

7.1 FULLY GENERIC COUNT

In this section, we develop a fully generic count function that counts the number of nodes that make up a generically encoded value. The count function is used in the type of the subsequently-defined generic lookup in Section 7.2. The count function is used as the maximum bound for the index argument of lookup.

7.1.1 Generic Types

Before covering the details of implementing count, we return to the introduction of our dissertation to clarify our intuition about the type signatures of fully generic functions. In Section 1.2.3, we hinted that any fully generic function can be defined by mutually defining a function over all types and another function over all descriptions (whose fixpoint is a special case of the function over all types).

$$(A : \mathsf{Type})\ (a : [\![\ A\ ]\!]) \to \ldots$$

$$(D : \mathsf{Desc})\ (x : \mu\ D) \to \ldots$$

Specializing this template to a generic count, and making some changes to work with our closed universe of Section 6.2 (discussed below), results in the following two mutually defined functions.

```
count : (A : ‘Set) → ⟦ A ⟧ → ℕ
countμ : {O : ‘Set} (D : ‘Desc O) → μ₁ ⟦ O ⟧ « D » → ℕ
```

The intuition (presented in Section 1.2.3 of the introduction) behind the closed count function is largely correct. The only difference is that we have renamed Type to ‘Set, to notationally emphasize that its interpretation as a Set is obtained by “removing the backtick”.

However, the intuition behind the closed countμ function is simplified in the introduction. A minor difference is that we must add an O parameter, to account for the codomain of the inductive-recursive decoding function.

The first major difference is that the intuition from the introduction leads to defining countμ over all *open* descriptions (Desc), but fully generic programming demands that we define it over all *closed* descriptions (‘Desc). Let's remind ourselves of the definition (from Section 5.4.4) of the type component of the fixpoint operator:

```
data μ₁ (O : Set) (D : Desc O) : Set where
  init : ⟦ D ⟧₁ D → μ₁ O D
```

Recall that μ_1 expects O to be the kind of open types (Set), and D to be the kind of open descriptions (Desc). When we write the type of a generic function, like countμ, we quantify over all closed types O (of type ‘Set), and all closed descriptions D (of type ‘Desc).

The third argument to countμ is the result of applying the type meaning function (⟦_⟧) to the closed type (‘μ_1 O D), which definitionally reduces to μ_1 applied the type meaning (⟦_⟧) of O and the description meaning («_») of D. This models values of closed types within our open metalanguage, Agda (using open types like μ_1).

The second major difference between the types we use for fully generic programming, and the types behind the intuition in the introduction, is that we cannot directly define a function like countμ over all closed descriptions. The problem is that the inductive hypothesis is not general enough in the infinitary (hence, also inductive) ‘δ case. If we tried to write countμ directly, we would not remember the original inductive description when we reach the ‘δ case, because countμ would be defined by recursively destructing the description argument.

Instead of mutually defining count with countμ (a function over all algebraic

types), we mutually define count with counts (a function over all arguments of algebraic types, isomorphic to countμ). The counts function has an extra description argument, R, that stays constant to remember the original description as the D description argument is recursively destructed.

$$\text{counts} : \{O : \text{‘Set}\}\ (D\ R : \text{‘Desc}\ O) \to [\![\ «\ D\ »\ ]\!]_1\ «\ R\ » \to \mathbb{N}$$

Recall that $[\![_]\!]_1$ (defined below, for reference) is the type component of the interpretation function for descriptions. It appears as the sole argument to the init ial algebra constructor of μ_1. Because $\mu_1\ O\ D$ is isomorphic to $[\![\ D\ ]\!]_1\ D$, defining counts is an acceptable alternative to defining countμ.

$$
\begin{aligned}
&[\![_]\!]_1 : \{O : \text{Set}\}\ (D\ R : \text{Desc}\ O) \to \text{Set} \\
&[\![\ \iota\ o\ ]\!]_1\ R = \top \\
&[\![\ \sigma\ A\ D\ ]\!]_1\ R = \Sigma\ A\ (\lambda\ a \to [\![\ D\ a\ ]\!]_1\ R) \\
&[\![\ \delta\ A\ D\ ]\!]_1\ R = \Sigma\ (A \to \mu_1\ _\ R)\ \lambda\ f \to [\![\ D\ (\lambda\ a \to \mu_2\ R\ (f\ a))\ ]\!]_1\ R
\end{aligned}
$$

The interpretation function ($[\![_]\!]_1$) recurses over the first argument (D) to determine the type of constructor arguments, while holding the second argument (R) constant. This allows $[\![_]\!]_1$ to remember the original *complete* description (R) of the algebraic type, even though it is destructing a copy of it (D) as it recurses.

By remembering the original description (R), the open δ case can request an infinitary (hence, also inductive) argument as the first argument to Σ. For analogous reasons, counts is generically defined over all descriptions (D), but also a copy (R) of the original *complete* description that it can use to count infinitary arguments in the closed ‘δ case.

In summary, we define how to generically count values of the closed universe in terms of 2 mutually defined functions, count and counts. The first is defined over all closed types (‘Set) and the second is defined over all closed descriptions (‘Desc).

7.1.2 Counting All Values

First, let's define count fully generically for all values of all types (of Appendix C). This involves calling counts in the `μ_1 case, defined mutually (in Section 7.1.3) over all arguments of the initial algebra. Below, we restate the type of count, and then define count by case analysis and recursion over all of its closed types.

```
count : (A : `Set) → ⟦ A ⟧ → ℕ
```

Recall that we wish to define count as the sum of all constructors and the recursive count of all constructor arguments. It may be helpful to review count for the *fixed* closed universe in the introduction (Section 1.2.2), to see how it compares to our new count, defined over an *extendable* closed universe (by user-declared datatypes).

Dependent Pair We count a dependent pair by summing the recursive count of both its components (a and b), plus 1 to also count the pair constructor (,).

```
count (`Σ A B) (a , b) = 1 + count A a + count (B a) b
```

Notice that the *dependent* type of the second component (b) is computed by applying the codomain of the dependent pair (B) to the first component (a).

Algebraic Fixpoint We count an algebraic fixpoint by recursively counting its arguments (xs) using counts, plus 1 to account for the init constructor.

```
count (`μ₁ O D) (init xs) = 1 + counts D D xs
```

When we initially call counts, D is used for both of its arguments. However, as counts recurses, the first description argument will be destructed while the second (original) description argument is held constant.

Remaining Values All constructors of the remaining types (such as Bool) do not have arguments, so we count them as 1 (for their constructor, plus 0 for their

arguments). Note that this includes functions (the ‘Π case), which we treat as a black box by counting the λ constructor as 1, without recursively counting its body.

$$\text{count } A\ a = 1$$

7.1.3 Counting Algebraic Arguments

Second, let’s define counts fully generically for all arguments of the initial algebra. This involves calling count in the ‘σ and ‘δ cases, defined mutually (in Section 7.1.2) over all values of all types. Below, we restate the type of counts, and then define counts by case analysis and recursion over all of its closed descriptions (for reference, the declaration of Desc appears in Appendix B).

$$\text{counts} : \{O : \text{‘Set}\}\ (D\ R : \text{‘Desc}\ O) \to [\![\ \text{«}\ D\ \text{»}\ ]\!]_1\ \text{«}\ R\ \text{»} \to \mathbb{N}$$

Recursion is performed over the first description argument (D), while the second argument (R) is kept constant, so we have access to the original description in the inductive ‘δ case.

Finally, our intention is to count an algebraic value init xs as 1 (for init) plus the recursive sum of all of its arguments (for xs). Even though xs is technically implemented as a sequence of dependent pairs (,), we will not add 1 for each pair constructor (,), which we choose to view as part of the encoding rather than something to be counted. Hence, counts treats its argument xs as a single n-tuple, rather than several nested pairs.[2]

[2] Although we are hiding the nested-pairs (of the initial algebra) aspect of the encoding, we are exposing the encoding when counting constructors. Constructors are encoded as a dependent pair, representing a disjoint union. Our count function counts the boolean and the pair constructor, rather than hiding that aspect of the encoding. We could create a separate universe of codes that explicitly represents constructors, along with a new meaning function mapping to the underlying Descriptions, so that our generic count could hide the encoding of constructors as derived disjoint unions. However, we chose not to do so to make the presentation easier to follow.

Non-Inductive Argument When we come across a non-inductive argument in a sequence of arguments, we sum the count of the non-inductive argument (a) with the counts of the remainder of the sequence of arguments (xs).

```
counts (`σ A D) R (a , xs) = count A a + counts (D a) R xs
```

Note that a is counted using our mutually defined count over all values, and xs is recursively counted (via counts) using the description resulting from applying the *dependent* description D (of `σ) to the value a. The original description R is passed to counts unchanged.

Inductive Argument When we come across an inductive argument, in a sequence of arguments, we sum the count of the inductive argument (x) with the counts of the remainder of the sequence of arguments (xs).

```
counts (`δ `⊤ D) R (f , xs) = count (`μ₁ _ R) (f tt) +
  counts (D (μ₂ « R » ∘ f)) R xs
```

Inductive arguments are a special case of infinitary arguments where the domain of the infinitary function is the unit type (⊤). The first argument to the *closed description* `δ is a *closed type*. Because the first argument is a closed type, we can pattern match against the closed unit type (`⊤). This allows us to distinguish how we count *inductive* arguments from how we count *infinitary* arguments, and is *only possible* because our universe is *closed* (i.e., if the argument had kind Set, it would be open and we could not pattern match against it)!

The inductive argument is obtained by applying the infinitary argument f to the trivial value tt. But what type should we use to count it? Because it is an *inductive* (hence, algebraic) value, the type should be the fixpoint (`μ_1) applied to some description. We kept the original description (R) to count inductive arguments for exactly this case.

The remaining sequence of arguments (xs) is recursively counted (via counts)

using the description resulting from applying the *dependent* description D (of ‘δ) to the *composition* of the decoding function fixpoint component (μ_2) and the infinitary value f. Recall (from Section 5.4.2) that the D argument of ‘δ is a description that depends on the *decoding function* for our inductive-recursive type. The *type* of the decoding function is the *implicit* composition of the decoding fixpoint component (μ_2) and the infinitary value f (the nature of the implicit composition is explained in Section 5.4.2). In our generic function above, we *explicitly* create the *value* of this decoding function to satisfy the implicit expectation in the type of its description.

Infinitary Argument When we come across an infinitary argument, in a sequence of arguments, we add 1 to the counts of the remainder of the sequence of arguments (xs). This counts the infinitary λ constructor as 1, treating it as a black box, analogous to how we count the ‘Π case as 1.

```
counts (‘δ A D) R (f , xs) = 1 + counts (D (μ₂ « R » ∘ f)) R xs
```

The remaining sequence of arguments (xs) is recursively counted just like the inductive (‘δ ‘⊤) case, where the dependent description D is applied to the composition of the fixpoint interpretation component (μ_2) and the infinitary argument (f).

Last Argument Every sequence of algebraic constructor arguments terminates in the trivial value tt of type unit (⊤), which we count as 1.

```
counts (‘ι o) R tt = 1
```

We could choose to count the trivial (tt) last argument as 0, hiding the aspect of the encoding that every sequence of arguments is terminated by tt. However, we choose to count tt as 1 because the subsequently defined generic function, lookup in Section 7.2, treats the result of count as an *index* into all values and subvalues

of a type.[3]

7.1.4 Examples

Now that we've defined count fully generically, let's run it on some examples to better understand how it works. The key lesson to take away is that count and counts use a *depth-first* traversal to count values and subvalues.

Natural Numbers First, we consider counting the closed encoding of the natural number 0. It may be helpful to review the closed definition of the natural numbers in Section 6.2.2. The natural number 0 is encoded (below) by zero as the initial algebra applied to a dependent pair (,) consisting of the boolean true (selecting the zero-constructor branch of the dependent description used to encode the natural numbers), and the unit value (tt).

```
zero : ⟦ ‘ℕ ⟧
zero = init (true , tt)
```

We generically count the closed zero by summing 1 for the initial algebra constructor, 1 for the true argument, and 1 for the terminating unit argument (tt),

[3] Given our generic encoding of *inductive-recursive* types, the ability to count or lookup the trivial value (tt) may not seem useful. Nevertheless, we include this functionality because it becomes useful when generalizing our universe to include *indexed* algebraic types. In the initial algebra semantics for indexed types, the final unit type (⊤) is replaced by the identity type (Id), used as a constraint on the index of the algebraic type. Being able to count and lookup the constraint is important in the indexed universe.

resulting in 3.[4]

```
count ‘ℕ zero ≡ 3
```

Next, let's define the closed natural number 1. We can define one by applying our closed successor (from Section 6.2.2) to our closed zero.

```
one : ⟦ ‘ℕ ⟧
one = suc zero
```

Expanding these definitions results in the closed encoding of 1 below.

```
one ≡ init (false , (λ _ → init (true , tt)) , tt)
```

The false value in the closed one definition selects the successor branch of the description, and the next argument contains the inlined definition of zero, wrapped in a function ignoring its trivial unit argument. Recall that *inductive* natural numbers are encoded as *trivially infinitary* types, using the unit type (⊤) as the domain of the infinitary function. The **Inductive** case of counts is able to recursively count the inductive body of the successor (i.e., zero) because it is able to pattern match against the closed type ‘⊤ to distinguish counting inductive (or trivially-infinitary) arguments from counting (truly) infinitary arguments.

```
count ‘ℕ one ≡ 6
```

Finally, we count closed one as 6, adding up 1 for each constructor appearing in the encoded definition (init, false, init, true, tt, and tt), from left to right. The reason behind that order is that count and counts recursively add 1 for each encoded constructor by doing a *depth-first* traversal. To help visualize the traversal, and

[4] Once again, this is an encoding-aware count, because it is used to *index* which nodes (in a generically encoded data structure) to lookup (in Section 7.2). It would also be possible to define an encoding-unaware count, that does not count true (encoding constructor choice) and tt (encoding the end of the sequence of constructors). Encoding-unaware count could be generically defined over a universe of constructor-aware descriptions, equipped with an interpretation function to translate constructor-aware descriptions into our constructor-unaware descriptions. Applying an encoding-unaware count to zero would result in 0, as it would not count constructor choice data (like true) or argument sequence data (like tt).

aid in the legibility of encoded values, refer to Figure 7.1. The edges of Figure 7.1 are labeled according to a depth-first traversal of nodes (where 0 is an implicit edge for the root node). Because count (and counts) traverses in a depth-first manner, each edge represents the aggregate count at the time count is called for the corresponding node. Note that the result of applying count to the root node is 1 plus the final edge (1 + 5, above).[5]

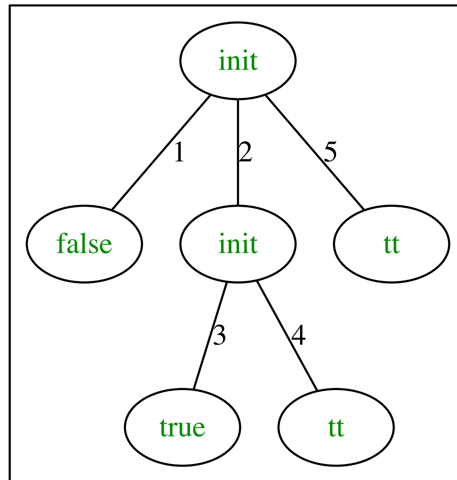

Figure 7.1: The natural number 1, as a closed algebraic type.

The depth-first labeling of edges pointing to nodes that count performs makes it an ideal function to index positions of arguments in generically encoded values. For example, in Figure 7.1 we can see value zero at edge 2, and value one at edge 0 (the root node). Finally, let's define closed two.

two : ⟦ ‘ℕ ⟧

[5] All algebraic types in figures hide the infinitary λ constructor at inductive argument positions, because count (whose depth-first traversal the Figure represents) implicitly applies trivially infinitary functions to tt in the **Inductive** (‘δ) case.

two = suc one

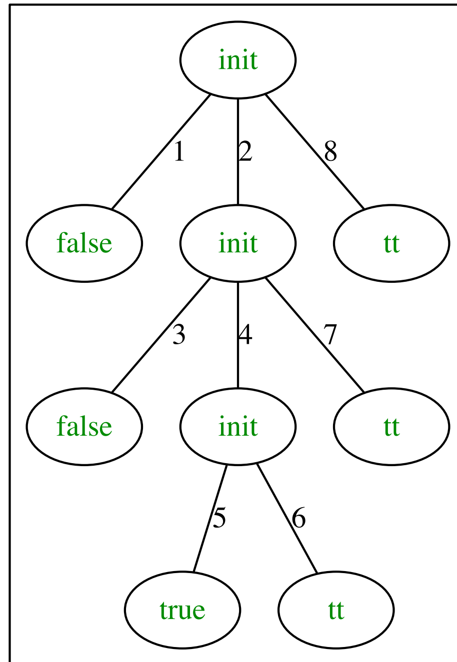

Figure 7.2: The natural number 2, as a closed algebraic type.

Counting closed two results in 9, as can be seen in Figure 7.2 (by adding 1 to the final edge 8). In Figure 7.2, two appears at edge 0, one appears at edge 2, and zero appears at edge 4.

count ‘ℕ two ≡ 9

Vectors Now let's encode the closed 0-length empty vector ([]). Again, it may be helpful to review the closed definition of vectors in Section 6.2.2. Recall that indexed vectors are encoded as a dependent pair whose first component is an inductive-recursive $‘\mathsf{Vec}_1$ (like a list, but with natural number arguments and index

constraints on inductive occurrences), and whose second component constrains (via the equality type Id) the decoding (via `Vec_2, or the length function) of the first component to be the appropriate index.

Below, the closed empty vector `[]` is encoded by nil as such a dependent pair, where the first component is an initial algebra value (of type `Vec}_1$), and the second component is a proof (using refl, the constructor of equality type Id) that $\text{`Vec_2 applied to the first component is indeed zero (the expected length of the empty vector, as specified by its type).

```
nil : {A : `Set} → ⟦ `Vec A zero ⟧
nil = init (true , tt) , refl
```

Our examples count vectors of pairs of strings. The generic count of the empty vector (i.e., nil) of pairs of strings is 5, the sum of 1 for init, true, tt, (,), and refl.

```
count (`Vec (`String `× `String) zero) nil ≡ 5
```

Next, let's define length-1 closed vector of pairs of strings `[("a", "x")]`. We can define vec1 by applying our closed cons constructor (from Section 6.2.2) to our closed zero constructor.

```
vec1 : ⟦ `Vec (`String `× `String) one ⟧
vec1 = cons ("a" , "x") nil
```

Expanding these definitions results in the closed encoding of `[("a", "x")]` below.

```
vec1 ≡ (init
  ( false
  , init (true , tt)
  , ("a" , "x")
  , (λ _ → init (true , tt))
  , refl
  , tt)
```

```
, refl)
```

To understand this, it is worth remembering that indexed vectors (‘Vec) are defined as a constraint paired with an inductive-recursive (but not indexed) version of the vector (‘Vec_1). Below, we directly define the indexed vector vec1 (of type ‘Vec), without using the smart constructors cons and nil, in terms of the auxiliary definition vec1_1 for the inductive-recursive (‘Vec_1) left component of the pair (of type ‘Vec_1).

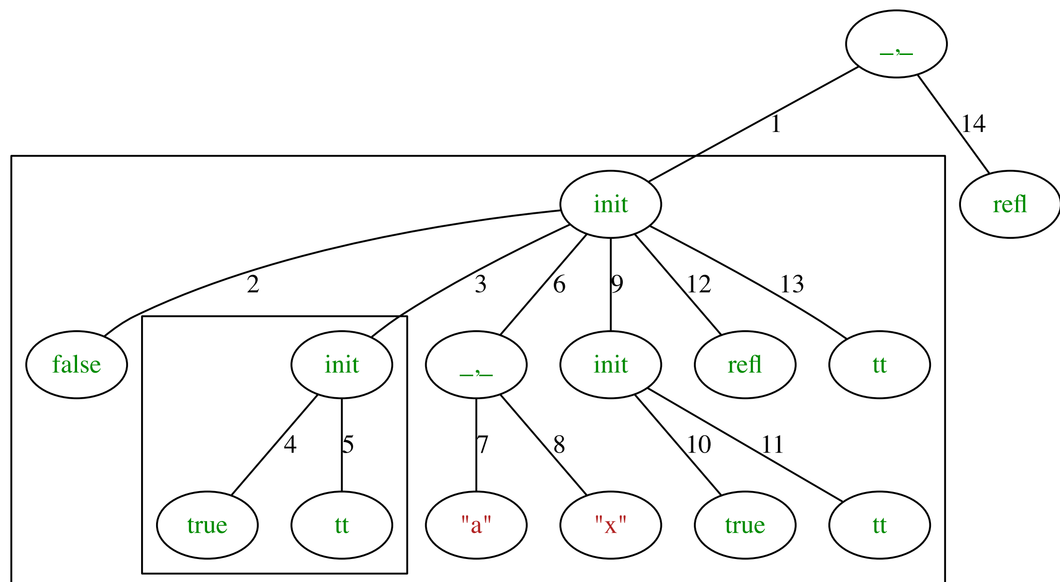

Figure 7.3: The length-1 vector of pairs of strings `[("a", "x")]`, as a closed algebraic type.

```
vec1₁ : ⟦ ‘Vec₁ (‘String ‘× ‘String) ⟧
vec1₁ = init
  ( false
  , init (true , tt)
  , ("a" , "x")
```

```
    , (λ _ → init (true , tt))
    , refl
    , tt)

vec1 : ⟦ ‘Vec (‘String ‘× ‘String) one ⟧
vec1 = vec1₁ , refl
```

To understand how vec1 is counted (as 15), we refer to our visual presentation in Figure 7.3, depicting the depth-first traversal of count. The root node is vec1 (of type ‘Vec), the dependent pair. Node 14 is the constraint (of type Id) that the left component has length one. Node 1 is the inductive-recursive vec1_1 (of type ‘Vec_1).

We establish the convention that suffixing a term or constructor by $_1$ refers to the inductive-recursive (of type ‘Vec_1) equivalent of the constraint pair of type ‘Vec. For example, nil_1 refers to an empty inductive-recursive vector of type ‘Vec_1, the left component of the constraint pair nil of type ‘Vec (mirroring the relationship between vec1_1 and vec1, above).

In Figure 7.3 (and all of our figures), we draw boxes around the outermost occurrence of an inductive value. For example, the root node does not have a box around it, because it is a non-inductive pair (of type ‘Σ). However, node 1 has a box around it, for the inductive ‘Vec_1 value (vec1_1). Within a box, any occurrence of init represents an inductive occurrence whose type shares the type of the box. For example, node 9 is a nil_1 value of type ‘Vec_1. Recall that each inductive argument of the inductive-recursive ‘Vec_1 is packaged with a constraint on its length (calculated by the inductive-recursive decoding function ‘Vec_2). Node 12 is the constraint that the length of node 9 (encoding the inductive occurrence of nil_1 within the box for vec1_1, at node 1) is 0.

Node 2 is false, representing the choice of the cons_1 constructor in the description of ‘Vec_1. Node 6 is the non-inductive pair (_,_) of strings "a" and "b"

contained by the vector. Node 3 contains a box around it, meaning it is an occurrence of an inductive type distinct from ‘Vec_1. Specifically, node 3 is the natural number zero, constrained to equal the length of the empty vector (nil_1) at node 9, in the type of the constraint (refl) at node 12. Finally, nodes 5, 11, and 13 all represent the terminating unit (tt) of an algebraic sequence of arguments.

```
count (‘Vec (‘String ‘× ‘String) one) vec1 ≡ 15
```

Finally, let's define the length-2 closed vector of pairs of strings `[("a", "x"), ("b", "y")]`. We can define vec2 with our closed constructors nil and cons of closed Vectors.

```
vec2 : ⟦ ‘Vec (‘String ‘× ‘String) two ⟧
vec2 = cons ("a" , "x") (cons ("b" , "y") nil)
```

The fully generic count of vec2 is 28, as justified by Figure 7.4. In Figure 7.4, node 12 is the length-1 `"b"` and `"y"` component of type ‘Vec_1. Node 14 is the natural number zero, the length of nil_1 at node 20. Node 3 is the natural number zero, the length of the `"b"` and `"y"` vector at node 12.

```
count (‘Vec (‘String ‘× ‘String) two) vec2 ≡ 28
```

We conclude this section by reflecting on how the relationship between algebraically defined length-indexed vectors and their algebraically defined natural numbers is elegantly captured in Figure 7.4. Notice how natural numbers (nodes 3 and 14) appear at the same level, and have the same height, as their vector equivalents (nodes 12 and 20). This visually demonstrates how the natural number argument of cons is mirrored by the structure of the inductive argument of cons, enforced by their relationship in the definition of the indexed type of vectors.

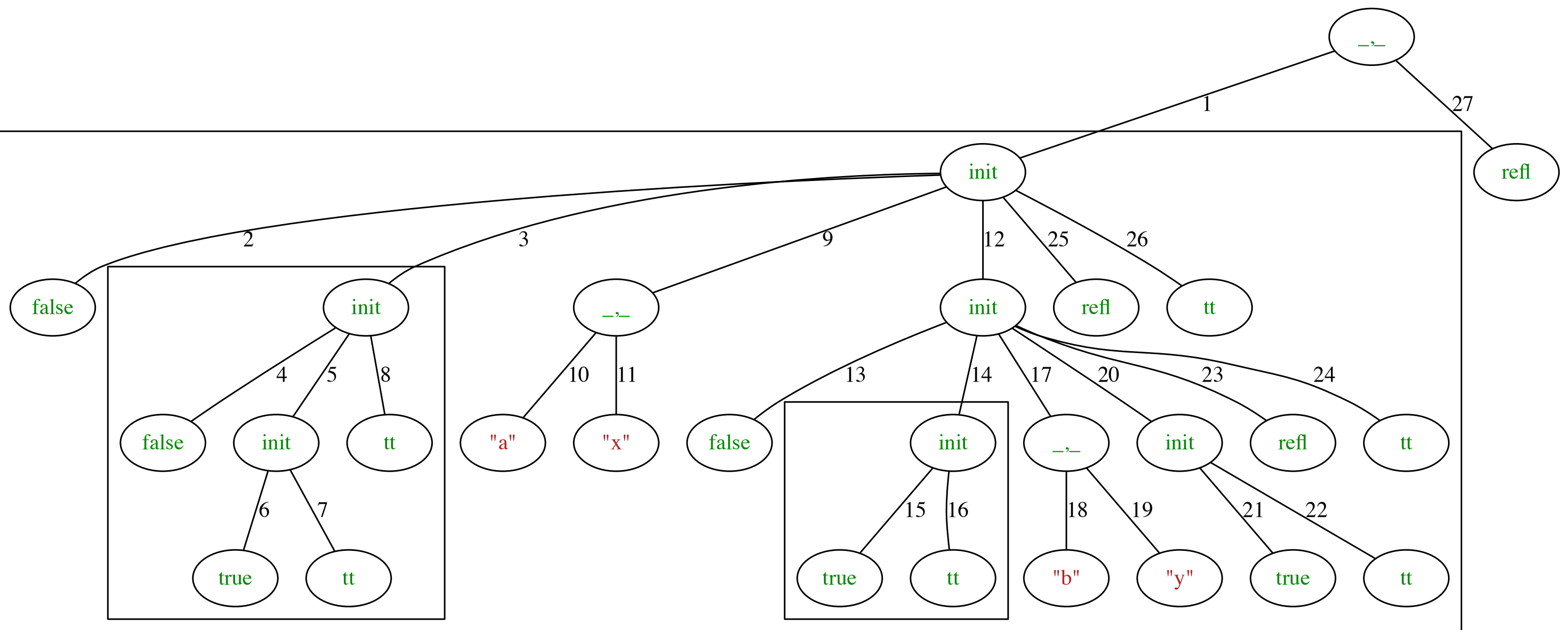

Figure 7.4: The length-2 vector of pairs of strings `[("a", "x"), ("b", "y")]`, as a closed algebraic type.

7.2 FULLY GENERIC LOOKUP

In this section, we develop a fully generic lookup function that can retrieve any node of a generically encoded value. The input to lookup is a value and an index into a position within the value. To prevent out-of-bounds errors during lookups, we generalize the type of lookup for vectors (Vec).

To retrieve a value within a vector, we apply lookup to a vector (Vec) and a finite set (Fin), where Fin acts as an index whose maximum value is constrained to equal the length of the vector. Recall the type of finite sets from Section 2.1.5.

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

We think of the type of finite sets as a 0-based index whose maximum value is the natural number that Fin is applied to, minus 1. For all n, there are n inhabitants of Fin n (representing indices 0 through n-1), where the first is here, and the rest are successive applications of there (ending in here). For example, the inhabitants of Fin 3 are here (for index 0), there here (for index 1), and there (there here) (for index 2).

To lookup a Vector of length n, we index by Fin n. The implementation of lookup returns the head of the vector (at index position 0) if the index is here. If the index is there, lookup recursively searches the tail of the vector (until it finally finds a value to return, indicated by peeling off enough theres to arrive at a here).

```
lookup : {A : Set} {n : ℕ} → Vec A n → Fin n → A
lookup (cons x xs) here = x
lookup (cons x xs) (there i) = lookup xs i
```

Instead of using vectors, we can define lookup equivalently over lists, by *computing* the maximum bound of the index (Fin) from the length of the List.

```
lookup : {A : Set} (xs : List A) → Fin (length xs) → A
```

```
lookup nil ()
lookup (cons x xs) here = x
lookup (cons x xs) (there i) = lookup xs i
```

The implementation for Lists is the same as the implementation for Vectors, other than needing to supply explicit evidence by pattern matching against the uninhabited empty Fin 0 index (using empty parentheses, which is Agda syntax for explicitly pattern matching against an uninhabited type) in the nil case.

Our fully generic lookup is defined similarly to the List lookup, except that length (calculating the bound of index Fin) is replaced by our fully generic count from Section 7.1. Recall that count sums the number of nodes in a generic value according to a depth-first traversal. Therefore, looking up a node in a generic value (using lookup) corresponds to supplying a Fin index representing the depth-first label of the node (seen in the figures of Section 7.1.4).

7.2.1 Generic Types

Before covering the details of implementing lookup, let's consider what its type should be. As mentioned above, we expect lookup to have a Fin index argument whose bound is calculated by the generic count of the value that lookup is applied to.

Looking up a List A results in an A, but looking up a node in a generic value causes the return type of lookup to depend on the type of node being looked up. Thus, we must define a computational return type (Section 3.4.4), named Lookup below, to determine what the return type of lookup should be. In other words, the Lookup function, with an uppercase 'L', computes the return type of the lookup function, with a lowercase 'l'.

```
Lookup : (A : `Set) (a : ⟦ A ⟧)
  → Fin (count A a) → Set
lookup : (A : `Set) (a : ⟦ A ⟧)
```

```
    (i : Fin (count A a)) → Lookup A a i
```

We must also mutually define Lookups, to compute the return type when looking up an argument of an algebraic constructor, via lookups. The lookups and Lookups functions are defined over all closed descriptions (‘Desc), analogous to how the lookup and Lookup functions are defined over all closed types (‘Set).

```
Lookups : {O : ‘Set} (D R : ‘Desc O) (xs : ⟦ « D » ⟧₁ « R »)
  → Fin (counts D R xs) → Set
lookups : {O : ‘Set} (D R : ‘Desc O) (xs : ⟦ « D » ⟧₁ « R »)
  (i : Fin (counts D R xs)) → Lookups D R xs i
```

The computational return types (Lookup and Lookups) are defined by pattern matching on some arguments. The functions using these types (lookup and lookups) must match the same arguments in the same way, in order for the computational return types to definitionally unfold. Instead of defining the value and type functions separately, thereby repeating the pattern matching structure twice, we will actually define single functions returning *dependent pairs* (Σ).

```
lookup : (A : ‘Set) (a : ⟦ A ⟧)
  → Fin (count A a) → Σ Set (λ T → T)
lookups : {O : ‘Set} (D R : ‘Desc O) (xs : ⟦ « D » ⟧₁ « R »)
  (i : Fin (counts D R xs)) → Σ Set (λ T → T)
```

The first component of the pair corresponds to the computational return type (Lookup or Lookups), and the second component of the pair corresponds to the function typed by the first component (the formerly named lookup or lookups). We can still recover the original type and value functions by taking the first (proj_1) and second (proj_2) projections of the dependent pairs (Σ) resulting from the new versions of lookup and lookups. By convention, we refer to the first projection (type component) of these functions by suffixing $_1$ (e.g., lookup_1), and to the second projection (value component) version by suffixing $_2$ (e.g., lookup_2).

7.2.2 Looking Up All Values

First, let's define lookup fully generically for all values of all types. This involves calling lookups in the 'μ_1 case, defined mutually (in Section 7.2.3) over all arguments of the initial algebra. Below, we restate the type of lookup, and then define lookup by case analysis and recursion over all of its closed types, *and* its Fin indices.

```
lookup : (A : 'Set) (a : ⟦ A ⟧) → Fin (count A a) → Σ Set (λ T → T)
```

Before we actually define lookup, let's consider what the type of the index Fin argument could be *before* we pattern match against it, and what lookup should return once we do match against the index. Below, we give a template of 3 different Fin types that appear when when defining lookup (as well as lookups). Each template represents a possible type of Fin, due to partially unfolding a case of count (in Section 7.1.2) or counts (in Section 7.1.3).

1. **Case** (Fin 1) There is only one possible index, so we define a here case that returns the value at this index.
2. **Case** (Fin $(1 + n)$) We define a here case (for the 1), returning the value at this index. We also define a there case (for the n), giving us a new argument of type Fin n. In the there case, we return the recursive call of lookup or lookups, depending on whether n is count or counts.
3. **Case** (Fin $(1 + n + m)$) We define a here case (for the 1), returning the value at this index. We also define a there case (for the $n + m$), giving us a new argument of type Fin $(n + m)$. Within the there case, we must translate the single Fin $(n + m)$ index to the disjoint union of the two potential indices Fin $n \uplus$ Fin m. After case-analyzing the disjoint union ($\uplus$), we make a recursive call using the index within the left (inj_1) or right (inj_2) injection. Once again, the recursive call is either lookup or lookups, depending on whether n or m (whichever one we find in the disjoint union) is count or counts.

To know which **Case** template to use for lookup at some type A, match the template with the case of count in Section 7.1.2 that A corresponds to.

Algebraic Fixpoint For the there case of algebraic fixpoints (**Case 2**), we recursively lookup its arguments (xs), using the mutually defined lookups.

```
- i : Fin (counts D D xs)
lookup (`μ₁ O D) (init xs) (there i) = lookups D D xs i
```

For clarity, we include the type of the index i (the argument of the there constructor) as a comment. In Agda, comments are colored red and are prefixed by a dash. In the type of i, the value that Fin is applied to corresponds to the value of n in **Case 2**.

Dependent Pair For the there case of dependent pairs (**Case 3**), we use the helper function splitΣ (defined in Figure 7.5) to turn i, a single index (Fin) containing a sum ($+$), into a disjoint union ($\uplus$) of two indices.

```
- i : Fin (count A a + count (B a) b)
lookup (`Σ A B) (a , b) (there i) with splitΣ A B a b i
- j : Fin (count A a)
... | inj₁ j = lookup A a j
- j : Fin (count (B a) b)
... | inj₂ j = lookup (B a) b j
```

If the disjoint union is the left injection (inj_1), we recursively lookup the first component of the pair (a). If the disjoint union is the right injection (inj_2), we recursively lookup the second component of the pair (b). The two possible values that Fin is applied to in the two possible types of j correspond to n and m in **Case 3**.

Remaining Values Finally, the here case can be defined uniformly over all types. If the index points to here, we simply return the value a at this position, along

with the type meaning function (⟦_⟧) applied to the closed type (A) of the value, as a dependent pair (,).[6]

```
lookup A@(‘Σ _ _) a here = ⟦ A ⟧ , a
```

For ‘μ_1 this is the here component of the definition of **Case 2**, and for ‘Σ this is the here component of the definition of **Case 3**. For all other types, this is the definition of **Case 1** (which does not have a there component).

7.2.3 Looking Up Algebraic Arguments

Second, let's define lookups fully generically for all arguments of the initial algebra. This involves calling lookup in the ‘σ and ‘δ cases, defined mutually (in Section 7.2.2) over all values of all types. Below, we restate the type of lookups, and then define lookups by case analysis and recursion over all of its closed descriptions, *and* its Fin indices.

```
lookups : {O : ‘Set} (D R : ‘Desc O) (xs : ⟦ « D » ⟧₁ « R »)
  → Fin (counts D R xs) → Σ Set (λ T → T)
```

We will also classify the cases in the definition of lookups by the **Case** template numbers. Below, we repeat the first 2 **Case** templates from Section 7.2.2 verbatim. However, the 3rd template is different because it lacks a "1 + ..." prefix.[7]

1. **Case** (Fin 1) There is only one possible index, so we define a here case that returns the value at this index.

[6] We use an @-pattern to bind A to the matched ‘Σ type. Unfortunately Agda makes us repeat this definition for all other remaining types, but at least the right-hand sides are all the same because of the @-pattern. The problem with lookup is that count appears in its type, which is defined using a catch-all pattern clause. Unfortunately, we cannot write lookup using the same catch-all pattern structure, and must instead enumerate all types and duplicate their right-hand sides manually. Defining lookup by repeating the catch-all structure of count would be possible if Agda were changed to type-check code *after* coverage checking.

[7] The lack of the "1 + ..." prefix is because we allow indexing into any argument of a sequence, but prevent indexing into the sequence itself or subsequences. Instead, we hide that aspect of the algebraic encoding, making init (containing the sequence of arguments) seem like one big n-ary tuple, rather than containing n nested pairs.

2. **Case** (Fin ($1 + n$)) We define a here case (for the 1), returning the value at this index. We also define a there case (for the n), giving us a new argument of type Fin n. In the there case, we return the recursive call of lookup or lookups, depending on whether n is count or counts.

3. **Case** (Fin ($n + m$)) There is only one possible index, so we define its there case (for $n + m$). Within the there case, we must translate the single Fin ($n + m$) index to the disjoint union of the two potential indices Fin $n \uplus$ Fin m. After case-analyzing the disjoint union ($\uplus$), we make a recursive call using the index within the left (inj_1) or right (inj_2) injection. Once again, the recursive call is either lookup or lookups, depending on whether n or m (whichever one we find in the disjoint union) is count or counts.

To know which **Case** template to use for lookups at some description D, match the template with the D case of counts in Section 7.1.3.

Non-Inductive Argument For the (only) there case of a non-inductive argument (**Case 3**), in a sequence of arguments, we use the helper function splitσ (defined in Figure 7.5). The helper function turns i, a single index (Fin) containing a sum (+), into a disjoint union ($\uplus$) of two indices. While splitΣ operates over a dependent pair, splitσ is specialized to operate over a non-inductive argument (a) and its dependent sequence (xs).

```
- i :  Fin (count A a + counts (D a) R xs)
lookups (‘σ A D) R (a , xs) i with splitσ A D R a xs i
- j :  Fin (count A a)
... | inj₁ j = lookup A a j
- j :  Fin (counts (D a) R xs)
... | inj₂ j = lookups (D a) R xs j
```

If the disjoint union is the left injection (inj_1), we recursively lookup the non-inductive argument (a). If the disjoint union is the right injection (inj_2), we recursively lookups the tail of the sequence of arguments (xs).

Inductive Argument For the (only) there case of an inductive argument (**Case 3**), in a sequence of arguments, we use the helper function splitδ (defined in Figure 7.5). The helper function turns i, a single index (Fin) containing a sum ($+$), into a disjoint union ($\uplus$) of two indices. The splitδ function is specialized to work with an *inductive* (i.e., not infinitary) argument, and its dependent sequence (xs). Hence, we apply splitδ to (f tt), computing the inductive codomain from the trivially infinitary f.

```
- i :  Fin (count (‘μ₁ _ R) (f tt) + counts (D (μ₂ « R » ∘ f)) R xs)
lookups (‘δ ‘⊤ D) R (f , xs) i with splitδ (D ∘ const) R (f tt) xs i
- j :  Fin (count (‘μ₁ _ R) (f tt))
... | inj₁ j = lookup (‘μ₁ _ R) (f tt) j
- j :  Fin (counts (D (μ₂ « R » ∘ f)) R xs)
... | inj₂ j = lookups (D (μ₂ « R » ∘ f)) R xs j
```

Infinitary Argument For the there case of an infinitary argument (**Case 2**), in a sequence of arguments, we recursively call lookups on its arguments (xs).

```
lookups (‘δ A@‘Bool D) R (f , xs) (there i) = lookups (D (μ₂ « R » ∘ f)) R xs i
```

For the here case of an infinitary argument (**Case 1**), we return the infinitary function. The type of infinitary function has the type meaning (⟦_⟧) of A as its domain, and the type component of the fixpoint μ_1, applied to the description meaning («_») of R, as its codomain.

```
lookups D@(‘δ A@‘Bool _) R (f , xs) here = (⟦ A ⟧ → μ₁ _ « R ») , f
```

Recall that lookup (in Section 7.2.2, as a special case of **Remaining Values**) only has a here case for functions (‘Π). Similarly, there is only a here case of lookups for infinitary functions (‘δ, where A is not ‘⊤). This is because we treat functions as a black box, so we can point at an entire function (using here), but not something within its body (using there).

Last Argument Finally, the here case of the last argument (**Case 1**), described by ‘ι , simply returns the unit type and value.

lookups (‘ι o) R tt here = ⊤ , tt

Note also that ‘ι does not have a there case, because it encodes the final (trivial) argument, so there is nothing left to index.

7.2.4 Splitting Functions

When defining lookup (in Section 7.2.2) and lookups (in Section 7.2.3), we appealed to the helper functions splitΣ, splitσ, and splitδ. All three of these helpers are defined as shallow wrappers (in Figure 7.5) around a more general function, splitFin.

Recall that if we match against index there i in **Case 3** of Section 7.2.2, then i has type Fin $(n + m)$, where n and m each represent either count or counts. In an instance of **Case 3**, such as the ‘Σ case of lookup (Section 7.2.2), we need to convert i into the disjoint union of Fin n and Fin m, so that we may recursively lookup the first component of the pair (a) using the left injection, or the second component of the pair (b) using the right injection. The splitFin function helps us do this.

splitFin : $(n\ m : \mathbb{N})$ $(i :$ Fin $(n + m))$ → Fin n ⊎ Fin m

We only need to define splitFin by pattern matching on the first (n) and third (i) arguments, but not the second argument (m). This is related to the fact that addition $(+)$ is defined by pattern matching on its first argument, so we only need to match n (not m) for the type of i, namely Fin $(n + m)$, to reduce.

Because lookup is defined as a depth-first search, the splitFin function must be left-biased (i.e., biased to return a Fin index whose bound is n, the left component of the sum). Hence, splitFin should return the left injection if index i points to

here, and n is greater than 0 (i.e., it is the successor of some other number).

```
splitFin (suc n) m here = inj₁ here
```

If n *is* zero, then the left injection is uninhabited (because it has type Fin 0), so we are forced to return the right injection (of type Fin m).

```
splitFin zero m i = inj₂ i
```

Finally, if n is greater than 0 and the index is there i, then we recursively split i.

```
splitFin (suc n) m (there i) with splitFin n m i
... | inj₁ j = inj₁ (there j)
... | inj₂ j = inj₂ j
```

If the recursive call of splitFin on i results in a left injection, it will have type Fin n. Hence, we must wrap the left injection in another there to bump the index and get the expected return type (Fin (suc n)). If the recursive call of splitFin on i results in a right injection, it already has the expected return type (Fin (suc m)).

In the input pattern of case above, there i has type Fin (suc $(n + m)$). In the recursive call, i has type Fin $(n + m)$. Hence, in the recursive call n becomes smaller (compared to suc n), while m remains the same. This explains why we bump the index (using there) if the recursive call results in a left injection, but not if it results in a right injection.

7.2.5 Examples

Now we run lookup on some examples. The expected behavior of lookup is to return the value associated with the nth (where n is the Fin index) position in a depth-first search. Hence, here of type Fin corresponds to the position 0, and there i of type Fin corresponds to the successor of the position associated with i.

```
splitΣ : (A : ‘Set) (B : ⟦ A ⟧ → ‘Set)
  (a : ⟦ A ⟧) (b : ⟦ B a ⟧) →
  Fin (count A a + count (B a) b) →
  Fin (count A a) ⊎ Fin (count (B a) b)
splitΣ A B a b i = splitFin (count A a) (count (B a) b) i

splitσ : {O : ‘Set} (A : ‘Set) (D : ⟦ A ⟧ → ‘Desc O) (R : ‘Desc  O)
  (a : ⟦ A ⟧) (xs : ⟦ « D a » ⟧₁ « R ») →
  Fin (count A a + counts (D a) R xs) →
  Fin (count A a) ⊎ Fin (counts (D a) R xs)
splitσ A D R a xs i = splitFin (count A a) (counts (D a) R xs) i

splitδ : {O : ‘Set} (D : ⟦ O ⟧ → ‘Desc O) (R : ‘Desc  O)
  (x : μ₁ ⟦ O ⟧ « R ») (xs : ⟦ « D (μ₂ « R » x) » ⟧₁ « R ») →
  Fin (count (‘μ₁ O R) x + counts (D (μ₂ « R » x)) R xs) →
  Fin (count (‘μ₁ O R) x) ⊎ Fin (counts (D (μ₂ « R » x)) R xs)
splitδ D R x xs i = splitFin (count (‘μ₁ _ R) x) (counts (D (μ₂ « R » x)) R xs) i
```

Figure 7.5: Definitions of the helper splitting functions (splitΣ, splitσ, and splitδ) used in Section 7.2.2 and Section 7.2.3. The helpers are all just shallow wrappers around the splitFin function (Section 7.2.4).

Natural Numbers Let's lookup_2 some values in the closed natural number two.[8] To see the expected value of lookup_2 for two at some index, simply consult Figure 7.2. The labels in the Figure 7.2 correspond to natural number indices, ordered according to a depth-first search. Thus, by viewing here of Fin as zero of ℕ, and there of Fin as suc of ℕ, we can always consult the figure to determine the expected return value of lookup_2.

Looking up index here corresponds to position 0, or the root node in Figure 7.2.

[8] Recall that we define lookup as a dependent pair, where the first component is a type and the second component is a value (whose type is the first component). As a shorthand, lookup_1 refers to the type component, while lookup_2 refers to the value component.

Hence, lookup_2 using index here is the identity function.

lookup_2 ‘ℕ two here ≡ two

If we lookup position 1 (using index there here) of two (visualized by Figure 7.2), we get false (encoding the choice of the cons constructor).

lookup_2 ‘ℕ two (there here) ≡ false

If we lookup position 2 (using index there (there here)) of two (visualized by Figure 7.2), we get one (visualized by Figure 7.1).

lookup_2 ‘ℕ two (there (there here)) ≡ one

To make lookups of higher positions more readable, we use a helper function (#) coercing natural numbers to finite sets by converting zero to here, and suc to there. Therefore, we can repeat the lookup_2 of position 2 above as follows.

lookup_2 ‘ℕ two (# 2) ≡ one

Finally, looking up position 4 of two results in zero.

lookup_2 ‘ℕ two (# 4) ≡ zero

Vectors Looking up vectors is just as easy as looking up natural numbers, by considering the Fin argument as a natural number index of a depth-first traversal of the closed value. For example, the lookup_2 of vec2 (visualized by Figure 7.4) at position 3 is the natural number one (visualized by Figure 7.1).

lookup_2 (‘Vec (‘String ‘× ‘String) two) vec2 (# 3) ≡ one

The lookup_2 of vec2 (visualized by Figure 7.4) at position 10 is the string "x".

lookup_2 (‘Vec (‘String ‘× ‘String) one) vec1 (# 8) ≡ "x"

Finally, the lookup_2 of vec2 (visualized by Figure 7.4) at position 12 is the

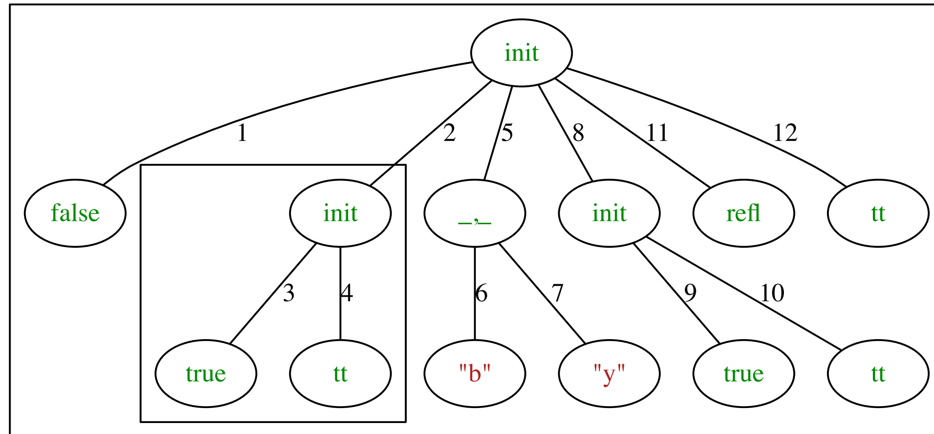

Figure 7.6: The inductive-recursive component of the length-1 vector of pairs of strings `[("b", "y")]`, as a closed algebraic type. This figure depicts the inductive-recursive first component of the vector encoded as a dependent pair (the second component is the length constraint).

inductive-recursive component of the vector `[("b", "y")]` (visualized by Figure 7.6).

```
lookup₂ (‘Vec (‘String ‘× ‘String) two) vec2 (# 12) ≡ cons₁ ("b" , "y") nil
```

7.3 FULLY GENERIC AST

In this section we develop a fully generic function (ast) to *marshal* values to an abstract syntax tree (AST). Previously (in Section 7.1.4 and Section 7.2.5), we visualized generically encoded data in figures (such as Figure 7.6). Those figures were created using fully generic programming, rather than drawn by hand. They are the result of applying ast to the value they visualize, converting the resulting AST to a graph in the DOT language [27], and rendering the DOT code using

Graphviz [25].[9]

The result of ast is a specialized version of a Rose tree. We use the standard List-based rose tree, rather than an infinitary version (Section 2.1.8). Additionally, throughout this section the "cons" constructor of List is denoted by (::), an infix constructor, and the "nil" constructor of List is denoted by [].

```
data Rose (A : Set) : Set where
  tree : A → List (Rose A) → Rose A
```

Specifically, the result of ast is a Rose tree containing Node values. A Node is one of the following constructors.

```
data Node : Set where
  non str : String → Node
  ind : Bool → Node
  lam : Node
```

Each Node constructor is translated to a DOT node differently (for example, the constructor determines the name of the DOT node, and the color of the name). Below, we describe what each Node constructor represents and how it affects the translation to DOT code:

- non: Used for non-inductive constructors, such as the pair constructor (_,_ of type Σ). The name of the node is determined by the String argument, and is colored *green*.
- str: Used for string values. The name of the node is determined by the String argument. The string argument is colored *red* and is enclosed in quotes.
- ind: Used for the inductive initial algebra constructor (init of type μ_1). The name of the node is "init", and is colored *green*. If the Bool argument is true, then a rectangle is drawn around the node.

[9] In this dissertation, we define the fully generic function ast to convert any value to an AST. The function to convert the AST to DOT code does not involve generic programming, so it can be found in the accompanying source code of this dissertation.

- ◇ lam: Used for higher-order values (i.e., the function case ‘Π and the infinitary case ‘δ). The name of the node is “λ”, and is colored *black*.

Finally, we abbreviate the result of ast as the type synonym AST, standing for Rose trees specialized to contain Nodes.

```
AST : Set
AST = Rose Node
```

7.3.1 Generic Types

Before implementing the fully generic marshalling functions, we consider the functions involved and their generic types. Two functions are unsurprising: ast defined over all closed types (‘Set) and asts defined over all closed descriptions (‘Desc). But, we will define one extra generic function, named astInd, also over all closed descriptions (‘Desc).

As expected, we will define (in Section 7.3.3) ast to fully generically translate any value to an AST.

```
ast : (A : ‘Set) (a : ⟦ A ⟧) → AST
```

Additionally, we will define (in Section 7.3.4) asts to fully generically translate algebraic arguments (of init), to a *list* of ASTs. Recall that the first argument of the tree constructor of AST (i.e., Rose specialized to Node) is a Node. The second argument to tree is a list of other rose trees (or ASTs). Hence, asts returns a List of ASTs, as it will be used for the second argument of tree.

```
asts : {O : ‘Set} (D R : ‘Desc O) → ⟦ « D » ⟧₁ « R » → List AST
```

Finally, we will define one additional helper function, astInd (in Section 7.3.2). The astInd function is defined fully generically over the fixpoint of any description.

```
astInd : {O : ‘Set} (D : ‘Desc O) → μ₁ ⟦ O ⟧ « D » → Bool → AST
```

Normally, we inline the definition of such a function by pattern matching on

init (in the 'μ_1 case of ast, and the 'δ '⊤ case of asts), and applying asts to the contained algebraic arguments. However, we prefer to extract the definition of astInd to define ast and asts.

Notice that astInd has an extra Bool argument. We will supply this argument to the ind constructor of Node, indicating whether or not to draw a box around the *inductive* node. Recall from Section 7.1.4 that we draw boxes in figures around the first occurrence of an inductive value, and suppress drawing boxes for any contained inductive arguments *of the same type*. However, inductive values of *different* types should start process over, beginning by drawing a box around the inductive node. In Section 7.3.3 and Section 7.3.4 (when defining ast and asts), we will see how passing the appropriate boolean to astInd implements this box drawing logic.

7.3.2 Marshalling Initial Algebras

First, let's define astInd fully generically over all descriptions and their fixpoints. Below, we restate the type of astInd, and define the only case that needs to be considered, the case for the lone initial algebra constructor of μ_1.

```
astInd : {O : 'Set} (D : 'Desc O) → μ₁ ⟦ O ⟧ « D » → Bool → AST
astInd D (init xs) b = tree (ind b) (asts D D xs)
```

The first argument of the rose tree constructor has type Node. Because initial algebras encode inductive types, we use the ind node. The boolean b argument is also passed along to the ind node.

The second argument of the rose tree constructor is a List of rose trees. Hence, the second argument to tree is the result of recursively applying the mutually defined asts function to the algebraic arguments (xs). Hence, the number of children of the resulting rose tree is equal to the number of arguments in xs.

7.3.3 Marshalling All Values

Second, let's define ast fully generically for all values of all types. Below, we restate the type of ast before defining it by its cases.

$$\text{ast} : (A : \text{`Set}) \; (a : [\![\, A \,]\!]) \to \text{AST}$$

Algebraic Fixpoint To define the fixpoint case, we simply apply the mutually defined astInd function to the algebraic argument x.

$$\text{ast} \; (\text{`}\mu_1 \;\; A \;\; D) \; x = \text{astInd} \;\; D \; x \; \text{true}$$

Importantly, we use true for the boolean argument of astInd. Hence, applying ast to any algebraic value (having type $\text{`}\mu_1$) results in drawing a box around it using the DOT language.

Dependent Pair The dependent pair case creates a rose tree with two children (in the second argument to tree, below), by recursively applying ast to each component of the pair (a and b).

$$\text{ast} \; (\text{`}\Sigma \;\; A \;\; B) \; (a \; , \; b) = \text{tree} \; (\text{non} \; \texttt{"_,_"}) \; (\text{ast} \;\; A \;\; a :: \text{ast} \; (B \;\; a) \;\; b :: [])$$

Because dependent pairs are non-inductive types, the first (Node) argument to tree is non. The argument to non is a string representing the name of the infix dependent pair constructor (_,_).

Dependent Function We treat higher-order values, like dependent functions, as black boxes. Hence, the ast of a function is a childless tree, whose Node is lam.

$$\text{ast} \; (\text{`}\Pi \;\; A \;\; B) \; f = \text{tree lam} \; []$$

String Strings are non-inductive values, so we use the str constructor of Node. Strings also have no additional arguments, so their AST tree has no children.

$$\text{ast}\ \text{`String}\ x = \text{tree}\ (\text{str}\ x)\ []$$

Note that the string value x is supplied as the argument to the str constructor of Node, so a quoted version of x can act as the name of the node when interpreted as DOT code.

Remaining Non-Inductive Values All remaining non-inductive values use the non constructor of Node, and are childless (except for the `⊥ value, which is uninhabited).

$$\begin{array}{l} \text{ast}\ \text{`}\bot\ () \\ \text{ast}\ \text{`}\top\ \text{tt} = \text{tree}\ (\text{non}\ \texttt{"tt"})\ [] \\ \text{ast}\ \text{`Bool true} = \text{tree}\ (\text{non}\ \texttt{"true"})\ [] \\ \text{ast}\ \text{`Bool false} = \text{tree}\ (\text{non}\ \texttt{"false"})\ [] \\ \text{ast}\ (\text{`Id}\ A\ x\ y)\ \text{refl} = \text{tree}\ (\text{non}\ \texttt{"refl"})\ [] \end{array}$$

Each occurrence of non is applied to a string name corresponding to the name of the marshalled constructor.

7.3.4 Marshalling Algebraic Arguments

Third, let's define asts fully generically for all arguments of the initial algebra. Below, we restate the type of asts before defining it by its cases.

$$\text{asts} : \{O : \text{`Set}\}\ (D\ R : \text{`Desc}\ O) \to [\![\ «\ D\ »\ ]\!]_1\ «\ R\ » \to \text{List AST}$$

Recall (from Section 7.1.3) that the arguments of the initial algebra are treated as one big n-tuple, rather than n nested pairs. This is why each case of asts returns a List of ASTs, rather than tree (non "_,_") applied to such a list.[10]

[10] If each case of asts did return such a tree (non "_,_") xs, then each init constructor in figures would have a pair (_,_) as its child node. The first component of the pair would be the head of xs. The second component of the pair child node would be a nested sequence of pairs, i.e., the

Non-Inductive Argument The non-inductive argument case results in a list whose head is the AST of the non-inductive argument (a), and whose tail is the List of ASTs for the remaining arguments (xs).

```
asts (‘σ A D) R (a , xs) = ast A a :: asts (D a) R xs
```

Inductive Argument The inductive argument case results in a list whose head is the AST of the inductive argument (f tt), and whose tail is the List of ASTs for the remaining arguments (xs).

```
asts (‘δ ‘⊤ D) R (f , xs) =
  astInd R (f tt) false :: asts (D (μ₂ « R » ∘ f)) R xs
```

Note that the AST of the inductive argument (f tt) is computed by astInd. Importantly, false is supplied as the boolean argument to astInd. This is because the inductive argument we are marshalling is known to be one of the arguments of some previous initial algebra (that was already marshalled with a box in the ‘μ_1 case of ast). Hence, we do *not* want to draw a box around this inductive argument occurrence, so we choose false as the argument to astInd.

Infinitary Argument The infinitary argument case results in a list whose head is a childless lam node, and whose tail is the List of ASTs for the remaining arguments (xs).

```
asts (‘δ A D) R (f , xs) = tree lam [] :: asts (D (μ₂ « R » ∘ f)) R xs
```

The reason why the head of the returned list is a lam node, is because we treat the higher-order infinitary function f as a black box. This is similar to how we treat functions as black boxes in the ‘Π of ast.

---

nested representation of the tail of arguments xs.

Last Argument Finally, the asts of the last argument (in the sequence of initial algebra arguments) results in a single element list.

```
asts (`ι o) R tt = tree (non "tt") [] :: []
```

The single element of the returned list is a childless non node (because the type of tt is ⊤, which is non-inductive). The name of the non is "tt", after the name of the trivial value tt.

7.3.5 Generic Template

We conclude this chapter by presenting a generic template that can be used to define fully generic algorithms. If the return type of a fully generic algorithm is *not* dependent on its inputs, then the algorithm can be implemented by mutually defining 2 functions.

```
generic : (A : `Set) → ⟦ A ⟧ → ···
generics : {O : `Set} (D R : `Desc O) → ⟦ « D » ⟧₁ « R » → ···
```

The first function (generic) is defined over all closed types (`Set) and their values. The second function (generics) is defined over all closed descriptions (`Data) and arguments of the described initial algebra.

If the return type of a fully generic algorithm *is* dependent on its inputs, then the algorithm can be implemented by mutually defining 4 functions.

```
Generic : (A : `Set) → ⟦ A ⟧ → Set
generic : (A : `Set) (a : ⟦ A ⟧) → Generic A a
Generics : {O : `Set} (D R : `Desc O) → ⟦ « D » ⟧₁ « R » → Set
generics : {O : `Set} (D R : `Desc O)
  (xs : ⟦ « D » ⟧₁ « R ») → Generics D R xs
```

The 2 uppercase functions (Generic and Generics) determine the return types of the 2 lowercase functions (generic and generics). Alternatively, we may mutually

define 2 functions that return dependent pairs (Σ).

```
generic : (A : `Set) (a : ⟦ A ⟧) → Σ Set (λ T → T)
generics : {O : `Set} (D R : `Desc O)
  (xs : ⟦ « D » ⟧₁ « R ») → Σ Set (λ T → T)
```

The first component of the pair corresponds to the generic dependent type of the function (Generic and Generics), and the second component corresponds to its generic inhabitant (formerly generic and generics, from the 4 mutually defined functions).

Chapter 8

CLOSED HIERARCHY OF UNIVERSES

Chapter 6 demonstrates closing a universe of algebraic (inductive-recursive) *types*, and Chapter 7 demonstrates fully generic programming over that universe. In this chapter[1], we expand the closed universe of Section 6.2 to also include *kinds*, *superkinds*, and an infinite hierarchy of such classifications. Types (Set) are classified by kinds (Set_1).

$$\mathsf{Set} : \mathsf{Set}_1$$

Going one level up, kinds (Set_1) are classified by superkinds (Set_2).

$$\mathsf{Set}_1 : \mathsf{Set}_2$$

This pattern repeats itself indefinitely. We refer to any such construction (e.g., a type, or a kind, or a superkind, etc.) as a *universe*.[2] In Section 6.2, we only considered the first (or zeroth, because we count universe levels by starting with 0) closed universe (i.e., the universe of types, classified by kinds). Now, we expand this notion to a closed infinite hierarchy of universes, where each universe in the hierarchy is classified by another universe, one level above.

$$\mathsf{Set}_n : \mathsf{Set}_{\mathsf{suc}\ n}$$

There are three primary things we achieve by creating a closed hierarchy of universes:

[1] This chapter is adapted from work by myself and Sheard [16], as explained in Section 9.4.

[2] In this context, a universe refers to a level in a hierarchy of types (e.g. the types level, or the kinds level, or the superkinds level, etc.). This is distinct from a universe as a formal device consisting of a type of codes and a meaning function. Although these are distinct concepts, the image of a particular meaning function could be a universe level of a hierarchy.

1. We can encode formers and constructors of kinds (as well as superkinds, etc.) in the closed hierarchy. This includes the two primitive kinds, closed types (‘Set) and closed descriptions (‘Desc). It also includes closed algebraic user-declared kinds, such as heterogenous lists (‘HList).

2. We can write *leveled* fully generic functions. By this we mean universe polymorphic fully generic functions, which can be applied to members of any universe in the hierarchy. Hence, we can extend fully generic functions (like count, lookup, ast, etc.) from working over all types and their values, to working over all kinds and their types (and all superkinds and their kinds, etc.).

3. We can *internalize* the kind (and superkind, etc.) signatures of formers, constructors, and functions, by writing them as the meaning of a closed code in our universe.

Let’s clarify what we mean by the third point, above. Throughout this dissertation we have written the signatures of closed formers, constructors, and functions using our Agda metatheory, which is *external* to our closed universe. For example, consider the *type* signature of the successor of closed natural numbers, below.

$$\mathsf{suc} : (n : [\![\ `\mathbb{N}\ ]\!]) \to [\![\ `\mathbb{N}\ ]\!]$$

The type signature of suc uses Agda’s *open* dependent function type (→). Instead, we may *internalize* the type signature of suc as the meaning ($[\![_]\!]$) of a *closed* dependent function (‘Π).

$$\mathsf{suc} : [\![\ `\Pi\ `\mathbb{N}\ (\lambda\ n \to `\mathbb{N})\ ]\!]$$

Another way to look at this, is that we can fit the entire type signature of suc into the meaning brackets ($[\![_]\!]$), as a single closed type (‘Set). In contrast, let’s

consider the *kind* signature of the cons constructor of closed parameterized lists.

$$\mathsf{cons} : (A : \text{`}\mathsf{Set}) \; (a : \llbracket \; A \; \rrbracket) \; (xs : \llbracket \; \text{`}\mathsf{List} \; A \; \rrbracket) \to \llbracket \; \text{`}\mathsf{List} \; A \; \rrbracket$$

We cannot internalize the *kind* signature of the cons function. Even though cons returns a list value, its signature is kinded because the A argument is a kind (‘Set). We would like to internalize the kind of cons as three nested uses of ‘Π (for arguments A, a, and xs). However, the domain of the first ‘Π would need to be a constructor of closed kinds (‘Set), which does not exist in the universe of closed types (Appendix C).

Similarly, we cannot internalize the kind signature of fully generic functions (like count), or even parametrically polymorphic functions (like the identity function), because they need to quantify over all closed types. By defining a closed hierarchy of universes in Section 8.2, we *can* internalize kind (and superkind, etc.) signatures, thereby fitting them within meaning brackets. In the final examples of Section 8.1.2, we also discuss a sanity check for our closed hierarchy of inductive-recursive universes, making sure that the signature of every constructor in the universe can be internalized (the evidence is in Appendix E).

Major Ideas The purpose of this chapter is to expand the closed universe of types from Section 6.2 to a hierarchy of universes, including kinds, superkinds, etc. Because the universe of Section 6.2 only models the *types* of a dependently typed language, it does not model a language that supports *polymorphism*. This is because there is no way to quantify over all types or descriptions (i.e., there is no code for ‘Set or ‘Desc). Because types and descriptions are *kinds*, a model of a language supporting polymorphism must model kinds in addition to types. We go one step further and model the entire closed hierarchy of universes (in Section 8.2.1). Hence, the closed hierarchy of universes in this chapter models a dependently typed language that supports polymorphic functions, including fully generic ones!

The fully generic functions of Chapter 7, like count of Section 7.1, are written using the function space of Agda (our metalanguage), by quantifying over ‘Set and ‘Desc. In contrast, in Section 8.3, we write a fully generic count *within* the dependently typed language that we are modeling, because the kind signature of the generic function can be internalized using codes from our hierarchy of universes (i.e., by quantifying over kind codes ‘Set and ‘Desc).

8.1 CLOSED HIERARCHY OF WELL-ORDER TYPES

In this section we extend the *Closed Well-Order Types* universe of Section 4.2.1 to a closed hierarchy of universes. At first, we present a formal model (in Section 8.1.1) of the hierarchy. Agda does not recognize our definition of the universe hierarchy type to be positive. However, we explain why the formal model presented in this section is consistent, and use it as motivation to define a model (in Section 8.1.3) that Agda recognizes as positive.

By extending the *Closed Well-Order Types* universe to a hierarchy, we can explain how a hierarchy is formalized in a simpler setting where Set is the only kind being closed over (Section 8.1.1 and Section 8.1.3). With this background material under our belt, we move on to extending the *Closed Inductive-Recursive Types* universe in Section 8.2.1. There, we must close over a hierarchy involving two kinds, Set and Desc.

8.1.1 Formal Model

Now we define a formal model of a *Closed Hierarchy of Well-Order Universes*. We do this by mutually defining a type of universe codes (‘Set[_]), *indexed* by the natural numbers, and a meaning function (⟦_|_⟧) mapping a universe in the hierarchy to an Agda type (i.e., a type of our metalanguage). Henceforth, we refer to ‘Set[_] as the *leveled types* and ⟦_|_⟧ as the *leveled type meaning function*.

The natural number index represents the universe level, in a hierarchy of universes. For example, ‘Set[0] models closed types (whose open equivalent is Set), ‘Set[1] models closed kinds (whose open equivalent is Set_1), ‘Set[2] models closed superkinds (whose open equivalent is Set_2), and so on.

Closed Leveled Types Below, we state the type former of of the closed leveled types, and subsequently define its constructors.

```
data ‘Set[_] : ℕ → Set where
```

The name of the indexed type (‘Set[_]) is Agda syntax for defining an infix operator, such that the natural number index appears where the underscore is located. For example, the universe of closed types is represented by ‘Set[*0*]. The left component of the infix operator name is ‘Set[, and the right component of the name is].

Closed Types Now let’s define the closed types. The closed types inhabit ‘Set[0], where the natural number index is 0, encoding the zeroth universe of types. However, we want a version of all closed types (especially closed type formers like ‘Π) to appear at higher universes as well.

```
‘⊥ ‘⊤ ‘Bool : ∀{ℓ} → ‘Set[ ℓ ]
‘Σ ‘Π ‘W : ∀{ℓ} (A : ‘Set[ ℓ ]) (B : ⟦ ℓ ∣ A ⟧ → ‘Set[ ℓ ]) → ‘Set[ ℓ ]
‘Id : ∀{ℓ} (A : ‘Set[ ℓ ]) (x y : ⟦ ℓ ∣ A ⟧) → ‘Set[ ℓ ]
```

Above, the index in the codomain of all constructors is ℓ. Thus, we have defined closed types as the special case where ℓ is 0, and a copy of the closed types at all higher levels. The constructors of these types are similar to the constructors of ‘Set in Section 4.2.1, but with an extra polymorphic level (ℓ) threaded throughout.

Closed Kinds Now let’s define the closed kinds. The closed types inhabit ‘Set[1], where the natural number index is 1, encoding the first universe of kinds. We

also want a version of all closed kinds to appear at higher universes.

```
`Set : ∀{ℓ} → `Set[ suc ℓ ]
`⟦_⟧ : ∀{ℓ} → `Set[ ℓ ] → `Set[ suc ℓ ]
```

Above, the index in the codomain of all constructors is suc ℓ. Thus, we have defined closed kinds as the special case where ℓ is 0, and a copy of the closed kinds at all higher levels.

At universe level 1, `Set is the closed kind of types (`Set : `Set[ 1 ]). At universe level 0, the `Set constructor is uninhabited because its index specifies that it should be greater than or equal to 1.

We have also added a closed meaning function constructor (`⟦_⟧), allowing us to *lift* a *type* from the previous universe to be a *kind* in the current universe. The closed meaning function (`⟦_⟧), or type lifting operator, ensures that our universes form a *hierarchy*. This is because we can apply the type lifting operator `⟦_⟧ to any universe `Set[ℓ], making the lifted value a member of the subsequent universe `Set[suc ℓ].

Meaning of Closed Leveled Types Second, we state the signature of the meaning function of closed leveled types.

```
⟦_|_⟧ : (ℓ : ℕ) → `Set[ ℓ ] → Set
```

The name of the meaning function (⟦_|_⟧) is Agda syntax for defining a mixfix operator. The natural number argument (ℓ) appears in the location of the first underscore, and the closed leveled type (`Set[_]) argument appears in the location of the second underscore.

Meaning of Closed Types The meaning of closed types (or their copies at higher levels) is straightforward.

```
⟦ ℓ | `⊥ ⟧ = ⊥
⟦ ℓ | `⊤ ⟧ = ⊤
```

$$
\begin{aligned}
&[\![\ \ell \mid \text{`Bool}\ ]\!] = \mathsf{Bool}\\
&[\![\ \ell \mid \text{`}\Sigma\ A\ B\ ]\!] = \Sigma\ [\![\ \ell \mid A\ ]\!]\ (\lambda\ a \to [\![\ \ell \mid B\ a\ ]\!])\\
&[\![\ \ell \mid \text{`}\Pi\ A\ B\ ]\!] = (a : [\![\ \ell \mid A\ ]\!]) \to [\![\ \ell \mid B\ a\ ]\!]\\
&[\![\ \ell \mid \text{`W}\ A\ B\ ]\!] = \mathsf{W}\ [\![\ \ell \mid A\ ]\!]\ (\lambda\ a \to [\![\ \ell \mid B\ a\ ]\!])\\
&[\![\ \ell \mid \text{`Id}\ A\ x\ y\ ]\!] = \mathsf{Id}\ [\![\ \ell \mid A\ ]\!]\ x\ y
\end{aligned}
$$

The leveled closed type meaning function ($[\![_|_]\!]$) is similar to the unleveled version ($[\![_]\!]$) in Section 4.2.1, but with an extra polymorphic level (ℓ) threaded throughout.

Meaning of Closed Kinds The meaning of closed kinds interprets the *code* of types ‘Set as a leveled type ‘Set[_], and the *code* of the closed meaning function ‘$[\![_]\!]$ as the leveled meaning function $[\![_|_]\!]$.

$$
\begin{aligned}
&[\![\ \mathsf{suc}\ \ell \mid \text{`Set}\ ]\!] = \text{`Set[}\ \ell\ \text{]}\\
&[\![\ \mathsf{suc}\ \ell \mid \text{`}[\![\ A\ ]\!]\ ]\!] = [\![\ \ell \mid A\ ]\!]
\end{aligned}
$$

Importantly, the level decreases, from suc ℓ to ℓ, when interpreting closed kinds (and their copies at higher universe levels). Hence, we interpret the code of types in this universe (‘Set) as the actual leveled type of the previous universe (‘Set[ℓ]).

Consider the case where ‘Set : ‘Set[1]. This implies that the interpretation $[\![\ 1 \mid \text{`Set}\ ]\!]$ is equal to ‘Set[0]. In this model, the level decreasing (from 1 to 0) captures the high-level notion that $\mathsf{Set}_0 : \mathsf{Set}_1$, preventing a “type in type” (or “kind in kind”, etc.) paradox (i.e., $\mathsf{Set}_1 : \mathsf{Set}_1$, if the level did not decrease).

Failing Positivity Check The problem with the formal model presented above is that it fails Agda’s positivity checker. The meaning function ($[\![_|_]\!]$) appears in the domain of the B argument of the ‘Σ, ‘Π, and ‘W constructors of leveled types (‘Set[_]). If this meaning function is applied to the code of types (e.g., $[\![\ 1 \mid \text{`Set}\ ]\!]$), then the result will be a leveled type (e.g., ‘Set[0]), making B a *negative* infinitary argument.

By external analysis of the definition of the leveled types indexed by the natural numbers, we can see that the index decreases (from 1 to 0) when a negative occurrence manifests. Furthermore, there are no constructors of ‘Set[_] with an argument whose level increases. Therefore, no leveled type in the hierarchy contains types from levels above it (it only contains types from levels below it). Hence, argument B is not actually a negative occurrence, because it only contains lower types, which cannot contain any types at the current level. Currently, Agda's positivity checker cannot perform such an analysis, so Section 8.1.3 defines an Agda model that reifies our positivity argument in its structure.

8.1.2 Examples

Let's consider some examples where we internalize the signatures of functions using codes of our universe hierarchy.

Negation Function First, we define the negation function (not), whose type is defined using a dependent function (→), external to our closed hierarchy (i.e., from our Agda metalanguage). Below, we insert the type of not in open type theory as a comment.

```
- not :  (b :  Bool) → Bool
not : (b : ⟦ 0 | ‘Bool ⟧) → ⟦ 0 | ‘Bool ⟧
not true = false
not false = true
```

Note that the signature is a *type* because the universe level (i.e., the first argument to the meaning function) is 0. Now, we internalize the type signature of negation.

```
not : ⟦ 0 | ‘Π ‘Bool (λ b → ‘Bool) ⟧
not true = false
```

```
not false = true
```

It is good that we can internalize the *type* of negation, but we could already do that using our universe of closed types in Section 4.2.1. Our next example (the identity function) shows how to internalize a *kind*, which the universe of Section 4.2.1 cannot do.

Identity Function First, we define the identity function (id), whose type is defined using a dependent function (→) external to our closed hierarchy. Below, we insert the type of id in open type theory as a comment.

```
- id :  (A : Set) (a :  A) → A
id : (A : `Set[ 0 ]) (a : ⟦ 0 | A ⟧) → ⟦ 0 | A ⟧
id A a = a
```

The type of the identity function quantifies over all types in the zeroth universe. Hence, the universe of closed types (in Section 4.2.1) cannot internalize the signature of id, because it is a *kind* signature that requires quantifying over all types. The universe of closed types (in Section 4.2.1) does not have a code for closed types (`Set), making such a quantification impossible.

```
id : ⟦ 1 | `Π `Set (λ A → `Π `⟦ A ⟧ (λ a → `⟦ A ⟧)) ⟧
id A a = a
```

Above, we have internalized the *kind* signature of id. The signature is a kind, because the universe level (i.e., the first argument to the meaning function) is 1. At universe level 1, the closed type constructor `Set and closed meaning function constructor (`⟦_⟧) are inhabited, allowing us to internalize the signature of id as a closed kind.

Note that the argument A in the closed kind of id is the meaning of `Set. At *kind* level 1, the meaning function of `Set returns a closed *type* (`Set[ 0 ], at level 0). Hence, the second argument of id, and the codomain of id, must lift (using `⟦_⟧) the *type* A (at level 0), so that the entire signature of id can be a *kind* (at

level 1).

8.1.3 Agda Model

Now we define an Agda model of a *Closed Hierarchy of Well-Order Universes*. Previously (in Section 8.1.1), we defined a formal model of the hierarchy as a datatype *indexed* by the natural numbers, which Agda fails to recognize as a positive definition. The Agda model of a *Closed Hierarchy of Well-Order Universes* is due to McBride [42].

Now, we define the hierarchy in 2 stages, allowing Agda to recognize the positivity of the definition. In the first stage, we define an *open* datatype (SetForm), *parameterized* by an abstract notion of the previous universe level (Level). In the second stage, we define the *closed* hierarchy (‘Set[_]) of universes, *indexed* by the natural numbers, but as a *computational family* (Section 2.1.11). In other words, we model the indexed definition (‘Set[_]) by *deriving* it as a *function* from the natural numbers to types, and this function is defined in terms of the parameterized definition (SetForm). Correspondingly, we also define a meaning function abstracted over the previous universe level (⟦_/_⟧), which is used to derive the meaning function over all levels (⟦_|_⟧).

Abstract Universe Levels First, we define the abstract notion of the previous universe (whose level is the predecessor of the current universe), as the dependent record Level. The Level record is used as the parameter of the type defined in the first stage of our hierarchy construction.

```
record Level : Set₁ where
  field
    SetForm : Set
    ⟦_/_⟧ : SetForm → Set
```

The SetForm field represents a closed type from the previous universe, and the

⟦_/_⟧ field represents the closed type meaning function from the previous universe. Note that Level is isomorphic to the Univ record of Section 6.2.3, just with different field names. Additionally, note that SetForm is a Set, and the codomain of ⟦_/_⟧ is Set, so Level is an *open* kind.

Pre-Closed Leveled Types Next, we state the type former (SetForm) of a type at an arbitrary level, parameterized by the universe at the previous level. Technically, SetForm is an *open* type, due to its use of the open Level parameter. However, we plan to fill in the parameter with a closed universe in stage 2 of the construction. Hence, we refer to SetForm, and associated constructions, as being *pre-closed*.

```
data SetForm (ℓ : Level) : Set where
```

We name our parameterized pre-closed type "SetForm". Whereas ‘Set[_] is *indexed* by natural numbers, SetForm is *parameterized* by the previous universe level. We call this type SetForm, because we intend to "fill in" the abstract universe level with a concrete universe in the second stage of the construction (i.e., when deriving the indexed type ‘Set[_]), just like we would "fill in" a "form".

Pre-Closed Types The pre-closed type constructors of our parameterized type (SetForm) are similar to the corresponding constructors of the indexed formal model (‘Set[_]).

```
‘⊥ ‘⊤ ‘Bool : SetForm ℓ
‘Σ ‘Π ‘W : (A : SetForm ℓ) (B : ⟦ ℓ / A ⟧ → SetForm ℓ) → SetForm ℓ
‘Id : (A : SetForm ℓ) (x y : ⟦ ℓ / A ⟧) → SetForm ℓ
```

Compared to ‘Set[_], the main difference is that the constructors of SetForm do not take the level ℓ as a formal argument. This is because ℓ is now a parameter (because it appears to the left of the colon in the datatype declaration), hence it is an informal and implicit argument of all constructors. Importantly, this allows

SetForm to be a *type*, even though it is parameterized by Level, which is a *kind* (as explained in Section 6.4.4).

Pre-Closed Kinds The main change in the pre-closed kinds appears in the pre-closed meaning function constructor (‘⟦_⟧).

```
‘Set : SetForm ℓ
‘⟦_⟧ : Level.SetForm ℓ → SetForm ℓ
```

The indexed closed meaning function constructor takes ‘Set[ℓ] as an argument and returns a ‘Set[suc ℓ]. In this parameterized version of the constructor, we *cannot* return a SetForm ℓ, because the parameter ℓ must remain constant for all constructors. However, we *can* make the argument to the constructor be a pre-closed type from the previous universe, by projecting SetForm out of our Level record parameter ℓ. Hence, the argument in the indexed and parameterized version of the meaning function constructor (‘⟦_⟧) both represent a closed type from the previous universe, just in different ways.

Meaning of Pre-Closed Leveled Types Now let's define the meaning function for pre-closed types parameterized by the previous universe.

```
⟦_/_⟧ : (ℓ : Level) → SetForm ℓ → Set
```

The only difference in the syntax of the type signature is that we use a slash (/), instead of a pipe (|), to distinguish the abstract Level version of the meaning function (⟦_/_⟧) from the natural number version (⟦_|_⟧).

Meaning of Pre-Closed Types The meaning of pre-closed types using abstract levels is syntactically identical to the natural number version, besides replacing pipes with slashes.

```
⟦ ℓ / ‘⊥ ⟧ = ⊥
⟦ ℓ / ‘⊤ ⟧ = ⊤
```

$$
\begin{aligned}
&[\![\, \ell \,/\, \text{`Bool} \,]\!] = \text{Bool} \\
&[\![\, \ell \,/\, \text{`}\Sigma\ A\ B \,]\!] = \Sigma\ [\![\, \ell \,/\, A \,]\!]\ (\lambda\ a \to [\![\, \ell \,/\, B\ a \,]\!]) \\
&[\![\, \ell \,/\, \text{`}\Pi\ A\ B \,]\!] = (a : [\![\, \ell \,/\, A \,]\!]) \to [\![\, \ell \,/\, B\ a \,]\!] \\
&[\![\, \ell \,/\, \text{`W}\ A\ B \,]\!] = \text{W}\ [\![\, \ell \,/\, A \,]\!]\ (\lambda\ a \to [\![\, \ell \,/\, B\ a \,]\!]) \\
&[\![\, \ell \,/\, \text{`Id}\ A\ x\ y \,]\!] = \text{Id}\ [\![\, \ell \,/\, A \,]\!]\ x\ y
\end{aligned}
$$

Meaning of Pre-Closed Kinds The meaning of pre-closed kinds is interpreted by projecting the field in the Level record ℓ associated with the pre-closed kind being interpreted.

$$
\begin{aligned}
&[\![\, \ell \,/\, \text{`Set} \,]\!] = \text{Level.SetForm}\ \ell \\
&[\![\, \ell \,/\, \text{`}[\![\, A \,]\!] \,]\!] = \text{Level.}[\![\, \ell \,/\, A \,]\!]
\end{aligned}
$$

The meaning of a pre-closed type (‘Set) is a pre-closed type (SetForm) from the previous universe (ℓ). The meaning of the pre-closed meaning function is the meaning function ([[_/_]]) from the previous universe (ℓ).

Passing Positivity Check In the definition of SetForm, the codomain of the B argument of the ‘Σ, ‘Π, and ‘W constructors is still an application of the meaning function ([[_/_]]). However, now the meaning of ‘Set of is an abstract Set from the Level record parameter ℓ, whose field we happened to call SetForm. This name simply documents that we plan to instantiate the field with a SetForm of the previous universe, in the second stage of our indexed universe hierarchy construction. From the point of view of the definition of SetForm, SetForm contains an arbitrary Set, so positivity is not violated when checking the infinitary B argument.

Derived Indexed Hierarchy of Universes Now we derive closed leveled types (‘Set[_]), indexed by the natural numbers, from pre-closed leveled types (SetForm), parameterized by levels (Level).

For each natural number, we need to apply SetForm to a closed Level encoding the previous universe in the hierarchy that the natural numbers represent. To do

so, we define the level function that maps each natural number, representing the *current* universe, to a Level, encoding the *previous* universe.

```
level : (ℓ : ℕ) → Level
```

If the universe level is 0, then there is no previous universe. Hence, we define the previous closed types (SetForm) to be uninhabited (i.e., the bottom type ⊥). The meaning function ⟦_/_⟧ for these previous closed types is also uninhabited, as indicated by a λ term matching against its empty argument (empty parentheses, in an argument position, is Agda syntax for matching against a value of an uninhabited type).

```
level zero = record
  { SetForm = ⊥
  ; ⟦_/_⟧ = λ()
  }
```

If the universe level is the successor of some natural number, then the previous closed types (SetForm) are the pre-closed types (SetForm), whose parameter is instantiated with level applied to the predecessor of the input natural number. The previous closed meaning function (⟦_/_⟧) is defined by the previous pre-closed meaning function (⟦_/_⟧) in the same fashion.

```
level (suc ℓ) = record
  { SetForm = SetForm (level ℓ)
  ; ⟦_/_⟧ = ⟦_/_⟧ (level ℓ)
  }
```

Thus, we inductively define closed universe Levels, for any natural number, by applying pre-closed constructions to previous closed levels, and defining the zeroth level to be uninhabited.

Finally, we derive (indexed) closed leveled types (and their meaning functions) by composing pre-closed types (and their meaning functions) with level.

```
`Set[_] : ℕ → Set
```

$`\mathsf{Set}[\ \ell\ ] = \mathsf{SetForm}\ (\mathsf{level}\ \ell)$

$[\![_|_]\!] : (\ell : \mathbb{N}) \rightarrow `\mathsf{Set}[\ \ell\ ] \rightarrow \mathsf{Set}$
$[\![\ \ell \mid A\ ]\!] = [\![\ \mathsf{level}\ \ell\ /\ A\ ]\!]$

The *indexed* leveled types are derived from the *parameterized* pre-closed types, because the pre-closed types are used to define level.

8.2 CLOSED HIERARCHY OF INDUCTIVE-RECURSIVE TYPES

In Section 8.1, we extend the *Closed Well-Order Types* universe of Section 4.2.1 to a *Closed Hierarchy of Well-Order Universes*. In this section, we extend the *Closed Inductive-Recursive Types* universe of Section 6.2 to a *Closed Hierarchy of Inductive-Recursive Universes*. We define the Agda model of the hierarchy (as in Section 8.1.3), skipping the formal model (as in Section 8.1.1).

8.2.1 Agda Model

Now we define an Agda model of a *Closed Hierarchy of Inductive-Recursive Universes*. Just like the Agda model of the *Closed Hierarchy of Well-Order Universes* in Section 8.1.3, we derive closed leveled types and their meaning from closed level universes, defined in terms of pre-closed constructions parameterized by Level.

Recall (from Section 6.2) that the *Closed Inductive-Recursive Types* universe is mutually defined by closed types (`Set), closed descriptions (`Desc), and their respective closed meaning functions ($[\![_]\!]$ and «_»). Similarly, the *Closed Hierarchy of Inductive-Recursive Universes* is mutually defined by pre-closed leveled types (SetForm), pre-closed leveled descriptions (DescForm), and their respective pre-closed leveled meaning functions ($[\![_/_]\!]$ and «_/_»).

Abstract Universe Levels Once again, we define a dependent record (Level) as the abstract notion of the previous universe to be used as the parameter of

SetForm and DescForm.

```
record Level : Set₁ where
  field
    SetForm : Set
    ⟦_/_⟧ : (A : SetForm) → Set
```

As before (in Section 8.1.3), the SetForm field represents the closed types of the previous universe, and the ⟦_/_⟧ field is their meaning function. Now we require three additional fields.

```
DescForm : (O : SetForm) → Set
⟦_/_⟧₁ : {O : SetForm} (D R : DescForm O) → Set
μ₁' : (O : SetForm) (D : DescForm O) → Set
```

The DescForm represents the closed descriptions of the previous universe. The μ_1' field represents the type component of the fixpoint operator of closed descriptions from the previous universe. Recall (from Section 5.4.2) that the type component of fixpoints (μ_1) is defined in terms of the type component of the interpretation function for descriptions ($⟦_⟧_1$). The $⟦_/_⟧_1$ field represents the type component of the interpretation function for closed descriptions of the previous universe.

Three Versions of Fixpoints We now take a brief intermission to warn the reader that there are three versions of *closed* fixpoints in this closed leveled universe construction. We explain why all three are necessary as this chapter unfolds, but for now just recognize that they are distinct (watch out for backtick prefixes and prime prefixes in the names of the three closed fixpoints):

1. `μ_1 This is a constructor of SetForm, and represents the fixpoint of descriptions in the *current* universe.
2. μ_1' This is a record field of Level, and represents the *abstract* version of the fixpoint of descriptions in the *previous* universe.

3. 'μ_1' This is a constructor of SetForm, and represents the *concrete* version of the fixpoint of descriptions in the *previous* universe. Hence, 'μ_1' is the concrete version of μ_1'.

Recall that there is also μ_1, the *open* fixpoint operator of Section 5.4.2. We now return you to your regularly scheduled generic programming.

Pre-Closed Leveled Types Below we give the type former of pre-closed leveled types, this time parameterized by our Level record containing description components, in addition to type components, from the previous universe.

```
data SetForm (ℓ : Level) : Set where
```

Pre-Closed Types The pre-closed types are no different from the well-order hierarchy of Section 8.1.3. The only exception is that we exchange the well-order constructor ('W) for the fixpoint constructor ('μ_1).

```
'⊥ '⊤ 'Bool : SetForm ℓ
'Σ 'Π : (A : SetForm ℓ) (B : ⟦ ℓ / A ⟧ → SetForm ℓ) → SetForm ℓ
'Id : (A : SetForm ℓ) (x y : ⟦ ℓ / A ⟧) → SetForm ℓ
'μ₁ : (O : SetForm ℓ) (D : DescForm ℓ O) → SetForm ℓ
```

Notice that the D argument of 'μ_1 is a mutually defined DescForm, in the same universe level (ℓ) as our current SetForm. This is a natural generalization of 'μ_1 from the closed universe in Section 6.2, which which takes a 'Desc and constructs a 'Set.

Pre-Closed Kinds The pre-closed kind of types ('Set) and their meaning function ('⟦_⟧) are no different from the well-order hierarchy of Section 8.1.3.

```
'Set : SetForm ℓ
'⟦_⟧ : Level.SetForm ℓ → SetForm ℓ
'Desc : Level.SetForm ℓ → SetForm ℓ
```

$`\llbracket_\rrbracket_1$: $\{O$: Level.SetForm $\ell\}$ $(D\ R$: Level.DescForm $\ell\ O) \rightarrow$ SetForm ℓ
$`\mu_1'$: $(O$: Level.SetForm $\ell)$ $(D$: Level.DescForm $\ell\ O) \rightarrow$ SetForm ℓ

We now add the pre-closed kind of descriptions (`Desc), and pre-closed kinds for the interpretation ($`\llbracket_\rrbracket_1$) and fixpoint ($`\mu_1'$) of descriptions. Recall that the pre-closed meaning function of types ($`\llbracket_\rrbracket$) can also be considered a function that *lifts* a *type* from the previous universe to the current universe. Similarly, the interpretation ($`\llbracket_\rrbracket_1$) and fixpoint ($`\mu_1'$) of descriptions both *lift* a description from the previous universe to the current universe.

Finally, we highlight the difference between the pre-closed fixpoint ($`\mu_1$), taking a DescForm of the *current* universe, and the pre-closed *lifting* fixpoint ($`\mu_1'$, notice the "prime" suffix), taking a DescForm from the *previous* universe. The former is used to construct algebraic *types* (like the natural numbers) in the zeroth universe or higher, while the latter is used to construct algebraic *kinds* (like heterogenous lists) in the first universe or higher.

If we use `Π to quantify over a `Set, then the domain (A) of the dependent argument (B) will be the meaning of `Set, which is a TypeForm of the previous universe. Thus, if we want to use $A$ in the type of $B$, we must lift it to the current universe with $`\llbracket_\rrbracket$. Similarly, if we use `Π to quantify over `Desc, then we can use the argument A in the type of B by lifting A (a DescForm) from the previous universe to the current universe via $`\llbracket_\rrbracket_1$ or $`\mu_1'$. Distinctively, we could not apply $`\mu_1$ to $A$, because $`\mu_1$ expects a DescForm from the current universe, not a DescForm from the previous universe.

Meaning of Pre-Closed Leveled Types Let's define the meaning function for pre-closed leveled types, having the signature below.

$\llbracket_/_\rrbracket$: $(\ell$: Level$) \rightarrow$ SetForm $\ell \rightarrow$ Set

Meaning of Pre-Closed Types The meaning of pre-closed types is no different from the well-order hierarchy version (Section 8.1.3), except we replace the ‘W case with the $‘\mu_1$ case.

```
⟦ ℓ / ‘⊥ ⟧ = ⊥
⟦ ℓ / ‘⊤ ⟧ = ⊤
⟦ ℓ / ‘Bool ⟧ = Bool
⟦ ℓ / ‘Σ A B ⟧ = Σ ⟦ ℓ / A ⟧ (λ a → ⟦ ℓ / B a ⟧)
⟦ ℓ / ‘Π A B ⟧ = (a : ⟦ ℓ / A ⟧) → ⟦ ℓ / B a ⟧
⟦ ℓ / ‘Id A x y ⟧ = Id ⟦ ℓ / A ⟧ x y
⟦ ℓ / ‘μ₁ O D ⟧ = μ₁ ⟦ ℓ / O ⟧ « ℓ / D »
```

In the fixpoint case ($‘\mu_1$), we compute the meaning of the description argument (D) using the mutually defined meaning of leveled pre-closed descriptions («_/_»).

Meaning of Pre-Closed Kinds The meaning of each pre-closed kind code is defined using its corresponding Level field, using the previous universe level ℓ.

```
⟦ ℓ / ‘Set ⟧ = Level.SetForm ℓ
⟦ ℓ / ‘⟦ A ⟧ ⟧ = Level.⟦ ℓ / A ⟧
⟦ ℓ / ‘Desc O ⟧ = Level.DescForm ℓ O
⟦ ℓ / ‘⟦ D ⟧₁ R ⟧ = Level.⟦ ℓ / D ⟧₁ R
⟦ ℓ / ‘μ₁’ O D ⟧ = Level.μ₁’ ℓ O D
```

Note that the arguments of each pre-closed kind code have exactly the types expected by the Level fields, so meaning translations (via ⟦_/_⟧ or «_/_») are unnecessary.

Pre-Closed Leveled Descriptions Let’s define the meaning function for pre-closed leveled descriptions, having the signature below.

```
data DescForm (ℓ : Level) (O : SetForm ℓ) : Set where
```

Note that pre-closed leveled descriptions are parameterized by O, a pre-closed type (SetForm) at the same level (ℓ) as the current pre-closed description (DescForm),

encoding the codomain of the decoding function for this inductive-recursive description.

Pre-Closed Descriptions The leveled pre-closed description constructors are just like the closed descriptions of Section 6.2. The only difference is that we replace closed constructions (‘Desc, ‘Set, and ⟦_⟧) with their pre-closed leveled counterparts (DescForm, SetForm, and ⟦_/_⟧), at level ℓ.

```
‘ι : (o : ⟦ ℓ / O ⟧) → DescForm ℓ O
‘σ : (A : SetForm ℓ) (D : ⟦ ℓ / A ⟧ → DescForm ℓ O) → DescForm ℓ O
‘δ : (A : SetForm ℓ) (D : (o : ⟦ ℓ / A ⟧ → ⟦ ℓ / O ⟧) → DescForm ℓ O)
   → DescForm ℓ O
```

Meaning of Pre-Closed Leveled Descriptions Let’s define the meaning function for pre-closed leveled descriptions, having the signature below.

```
«_/_» : (ℓ : Level) {O : SetForm ℓ} → DescForm ℓ O → Desc ⟦ ℓ / O ⟧
```

Meaning of Pre-Closed Descriptions The meaning of leveled pre-closed descriptions is also just like the meaning of closed descriptions in Section 6.2. This time we replace closed meaning functions (⟦_⟧ and «_») with their pre-closed leveled counterparts (⟦_/_⟧ and «_/_»), at level ℓ.

```
« ℓ / ‘ι o » = ι o
« ℓ / ‘σ A D » = σ ⟦ ℓ / A ⟧ (λ a → « ℓ / D a »)
« ℓ / ‘δ A D » = δ ⟦ ℓ / A ⟧ (λ o → « ℓ / D o »)
```

Derived Indexed Hierarchy of Universes Now that we’ve defined pre-closed leveled types and descriptions, *parameterized* by levels (Level), we can *derive* closed leveled types and descriptions, *indexed* by natural numbers (as a computational family). First, we define level to map each natural number to a Level representing

the *previous* universe (i.e., a natural number n is mapped to universe n-1).

```
level : (ℓ : ℕ) → Level
```

At level 0, there is no previous universe. Thus, field SetForm is bottom, field DescForm is a constant function returning bottom, and the meaning functions match against their uninhabited arguments (signified in Agda by empty parentheses in the argument position).

```
level zero = record
  { SetForm = ⊥
  ; ⟦_/_⟧ = λ()
  ; DescForm = λ O → ⊥
  ; ⟦_/_⟧₁ = λ ()
  ; μ₁' = λ ()
  }
```

If the universe level is the successor of some natural number, then the previous closed type and description fields (SetForm and DescForm) are the pre-closed types and descriptions (SetForm and DataForm), whose parameters are instantiated with level applied to the predecessor of the input natural number. The previous closed meaning function for types field (⟦_/_⟧) is defined by the previous pre-closed meaning function for types (⟦_/_⟧) in the same fashion.

```
level (suc ℓ) = record
  { SetForm = SetForm (level ℓ)
  ; ⟦_/_⟧ = ⟦_/_⟧ (level ℓ)
  ; DescForm = DescForm (level ℓ)
  ; ⟦_/_⟧₁ = λ D R → ⟦ « level ℓ / D » ⟧₁ « level ℓ / R »
  ; μ₁' = λ O D → μ₁ ⟦ level ℓ / O ⟧ « level ℓ / D »
  }
```

The closed description interpretation and fixpoint fields ($⟦_/_⟧_1$ and μ_1') are defined using the open description interpretation function and fixpoint ($⟦_⟧_1$ and μ_1) from Appendix B.

The open description interpretation function ($[\![_]\!]_1$) expects open description arguments, but the field $[\![_/_]\!]_1$ has leveled closed description arguments (D and R). Thus, we translate the leveled closed descriptions (D and R) using the leveled description meaning function («_/_») at the predecessor level (ℓ).

Similarly, the open description fixpoint (μ_1) expects an open type and an open description, but the field μ_1' has a leveled closed type argument (O) and a leveled closed description argument (D). The closed type (O) is translated using the leveled type meaning function ($[\![_/_]\!]$), and the closed description (D) is translated using the leveled description meaning function («_/_»). Both of the leveled meaning functions are translated at the predecessor level (ℓ).

Finally, we can derive indexed closed leveled types (‘Set[_]) from parameterized pre-closed leveled types (SetForm), by instantiating the parameter with the result of applying level to the input natural number index, as in Section 8.1.3. The leveled closed type meaning function ($[\![_|_]\!]$) is also derived from the pre-closed version ($[\![_/_]\!]$), as in Section 8.1.3.

```
‘Set[_] : ℕ → Set
‘Set[ ℓ ] = SetForm (level ℓ)

⟦_|_⟧ : (ℓ : ℕ) → ‘Set[ ℓ ] → Set
⟦ ℓ | A ⟧ = ⟦ level ℓ / A ⟧
```

Now, we additionally derive the indexed closed leveled descriptions (‘Desc[_]) from parameterized pre-closed leveled descriptions (DescForm), also by instantiating the parameter with the result of applying level to the input index. The leveled closed description meaning function («_|_») is derived from the pre-closed version («_/_») in the same way.

```
‘Desc[_] : (ℓ : ℕ) → ‘Set[ ℓ ] → Set
‘Desc[ ℓ ] O = DescForm (level ℓ) O

«_|_» : (ℓ : ℕ) {O : ‘Set[ ℓ ]} → ‘Desc[ ℓ ] O → Desc ⟦ ℓ | O ⟧
```

$$« \ell \mid D » = « \mathsf{level}\ \ell \ / \ D »$$

8.2.2 Examples

The *Closed Inductive-Recursive Types* universe examples in Section 6.2.2 correspond to examples that we can demonstrate in the zeroth universe of our hierarchy. The *Closed Inductive-Recursive Types* universe does not include the kinds ‘Set and ‘Desc, hence all of the signatures (e.g., NatDs, ‘ℕ, etc.) used to construct the examples were defined *externally* to the universe (using types from our Agda metalanguage, like the function space).

We can port all of the examples in Section 6.2.2 to the zeroth universe of our hierarchy by patching them using the table below. For each definition (in its signature and body), replace occurrences of the left table column with the right table column.

Closed Types Universe	Universe 0 in Hierarchy
‘Set	‘Set[0]
‘Desc	‘Desc[0]
⟦ A ⟧	⟦ 0 | A ⟧
« D »	« 0 | D »

However, we can also choose to *internalize* the signatures used in the examples, as we see below. By “internalize” we mean that each signature can be represented as the leveled type meaning (⟦_|_⟧), of some closed type, at some level in our hierarchy.

Natural Numbers Let’s internalize the signatures used in the natural number examples. The definition bodies remain the same as those in Section 6.2.2, so we only present the signatures below. First, we internalize the signatures of the closed

description and type *kinds* (i.e., at universe level 1).

```
NatDs : ⟦ 1 | ‘Bool ‘→ ‘Desc ‘⊤ ⟧
NatD : ⟦ 1 | ‘Desc ‘⊤ ⟧
‘ℕ : ⟦ 1 | ‘Set ⟧
```

Crucially, internalizing the kinds above relies on having codes for closed types (‘Set) and closed descriptions (‘Desc). If an internalized signature needs to refer to a type, it must refer to the internalized “backtick” version of the type. Because we can internalize *all* signatures, we no longer need to define non-backtick versions of types (e.g., ℕ). We can always recover a non-backtick version of a type by applying the meaning function (⟦_|_⟧) to the backtick version, at the appropriate level.

```
zero : ⟦ 0 | ‘ℕ ⟧
suc : ⟦ 0 | ‘ℕ ‘→ ‘ℕ ⟧
```

Above, we internalize the *value* (i.e., typed at universe level 0) constructors of the natural numbers.

Vectors Now, let’s internalize the *kinds* used to derive indexed vectors from inductive-recursive vectors.

```
VecDs : ⟦ 1 | ‘Set ‘→ ‘Bool ‘→ ‘Desc ‘ℕ ⟧
VecD : ⟦ 1 | ‘Set ‘→ ‘Desc ‘ℕ ⟧
‘Vec₁ : ⟦ 1 | ‘Set ‘→ ‘Set ⟧
‘Vec₂ : ⟦ 1 | ‘Π ‘Set (λ A → ‘⟦ ‘Vec₁ A ⟧ ‘→ ‘⟦ ‘ℕ ⟧) ⟧
‘Vec : ⟦ 1 | ‘Set ‘→ ‘⟦ ‘ℕ ⟧ ‘→ ‘Set ⟧
```

Notice that the decoding function (‘Vec_2) quantifies over the *kind* ‘Set, binding variable A. The bound variable A is a *type*, the inhabitant of the *kind* ‘Set. Hence, in order to ask for argument of ‘Vec_1 applied to A, we must first *lift* this type to the kind level (using ‘⟦_⟧). Also recall that ‘ℕ is defined to be a *type*. Hence, when asking for a natural number argument, in kind signatures of ‘Vec_2 and ‘Vec, we

also lift the ‘ℕ type to the kind level (using ‘⟦_⟧).

```
nil : ⟦ 1 | ‘Π ‘Set (λ A → ‘⟦ ‘Vec A zero ⟧) ⟧
cons : ⟦ 1 | ‘Π ‘Set (λ A → ‘Π ‘⟦ ‘ℕ ⟧ (λ n →
  ‘⟦ A ⟧ ‘→ ‘⟦ ‘Vec A n ⟧ ‘→ ‘⟦ ‘Vec A (suc n) ⟧)) ⟧
```

Above, we internalize the *value* constructors of the vectors. Even though the signatures of nil and cons are kinds (at universe level 1), their codomains return lifted (using ‘⟦_⟧) vector *types* (at universe level 0). For similar reasons, the natural number argument of cons is actually a value of *type* ‘ℕ, which has merely been lifted to the kind level to fit in the signature of cons.

To determine what level an argument or codomain lives at, substract the number of liftings (i.e., nested occurrences of ‘⟦_⟧) from the level of the signature (i.e., the number to the left of the pipe in the meaning function). For example, the codomain of nil is 1 minus 1 lifting, thus nil returns a value of *type* (i.e., universe level 0) ‘Vec, even though its signature is kinded (i.e., at universe level 1).

Finally, note that both nil and cons have explicit type arguments, and cons also has an explicit natural number argument. To change these to be implicit arguments, we would need to update our universe to include an implicit version of the ‘Π code (this is easy to to do).

Heterogenous Lists Previously, we defined types, like the natural numbers, whose signatures were kinds (at universe level 1). Now, we give an example of defining a kind, the heterogenous lists, whose signature is a superkind (at universe level 2). Defining the *kind* of heterogenous lists is not possible in the *Closed Inductive-Recursive Types* universe of Section 6.2, which only supports *types*. First, let's review the kind of heterogenous lists.

```
data HList : Set₁ where
  nil : HList
```

```
cons : (A : Set) → A → HList → HList
```

The signatures of the closed description and closed type, used to define heterogenous lists, are superkinded at universe level 2.

```
HListDs : ⟦ 2 | `Bool `→ `Desc `⊤ ⟧
HListDs true = `ι tt
HListDs false =
  `σ `Set λ A →
  `σ `⟦ A ⟧ λ a →
  `δ `⊤ λ xs →
  `ι tt

HListD : ⟦ 2 | `Desc `⊤ ⟧
HListD = `σ `Bool HListDs

`HList : ⟦ 2 | `Set ⟧
`HList = `μ₁ `⊤ HListD
```

Notice that the description of the first argument of the cons constructor (the false case of HListDs) takes a type as an argument (`Set), and the second argument takes a value of the lifting of that type. We can also see that `HList is a closed *kind*, because it is classified as a `Set at universe level 2. The meaning of `Set at universe level 2 is the SetForm of the previous universe level, or Set[*1*]. Hence, closed `HList is classified as a closed kind (Set[*1*]), just like open HList is classified as an open kind (Set_1).

```
nil : ⟦ 1 | `HList ⟧
nil = init (true , tt)

cons : ⟦ 1 | `Π `Set (λ A → `⟦ A ⟧ `→ `HList `→ `HList) ⟧
cons A a xs = init (false , A , a , (λ u → xs) , tt)
```

Above, we define the *kind* (i.e., universe level 1) constructors of the heterogenous lists. We know that nil and cons construct kinds, because their codomains do not have any liftings (i.e., occurrences of `⟦_⟧), so 1 - 0 leaves the codomains at

universe level 1, the level of kinds.

Identity Function In Section 8.1.2, we demonstrate internalizing the signature of the identify function in level 0 of the *Closed Hierarchy of Well-Order Universes*. We can still do this in our *Closed Hierarchy of Inductive-Recursive Universes*, as the internalized type below demonstrates.

```
id : ⟦ 1 | `Π `Set (λ A → `Π `⟦ A ⟧ (λ a → `⟦ A ⟧)) ⟧
id A a = a
```

For reference, we also present the external type signature that the meaning of our internal type above expands to.

```
id : (A : `Set[ 0 ]) (a : ⟦ 0 | A ⟧) → ⟦ 0 | A ⟧
id A a = a
```

Dependent Pair As a sanity check for the construction of our *Closed Hierarchy of Inductive-Recursive Universes* (Section 8.2.1), we should be able to internalize each signature (whether it be a type or kind) of every constructor of every datatype in the universe. This sanity check can be found in Appendix E.

As one illustrative example, we show how to internalize the pair constructor of dependent pairs. In open type theory (Appendix B), the pair constructor has the following type.

```
_,_ : {A : Set} {B : A → Set} (a : A) → B a → Σ A B
```

Below, we define pair' to be pair constructor init, while internalizing the kind signature of _,_.

```
pair' : ⟦ 1 | `Π `Set (λ A → `Π (`⟦ A ⟧ `→ `Set) (λ B →
    `Π `⟦ A ⟧ (λ a → `Π `⟦ B a ⟧ (λ b →
    `Σ `⟦ A ⟧ (λ a → `⟦ B a ⟧))))) ⟧
pair' A B a b = a , b
```

Internalizing the kind of the pair constructor (,) as pair' takes advantage of being

able to quantify over closed types (‘Set), and the closed type meaning function (‘⟦_⟧), used to lift types to the kind level. Really, it is just a slightly more involved example of internalizing the signature of the identity function (id).

Note that we must use explicit function arguments for A and B, as our universe does not currently support an implicit version of dependent functions (‘Π). For reference, we also present the external type signature that the meaning of our internal type above expands to.

```
pair' : (A : 'Set[ 0 ]) (B : ⟦ 0 | A ⟧ → 'Set[ 0 ])
  (a : ⟦ 0 | A ⟧) (b : ⟦ 0 | B a ⟧)
  → Σ ⟦ 0 | A ⟧ (λ a → ⟦ 0 | B a ⟧)
pair' A B a b = a , b
```

Initial Algebra As our final example, we internalize the signature of the initial algebra constructor (init) of fixpoints. The internalization of the signature for init is unique, as it quantifies over the closed kind of *descriptions* (‘Desc), and must be defined with description-lifting operations. First, review the type of the init constructor in open type theory (Appendix B).

```
init : {O : Set} {D : Desc O} → ⟦ D ⟧₁ D → μ₁ O D
```

Below, we define init' to be init, while internalizing the kind signature of init.

```
init' : ⟦ 1 | 'Π 'Set (λ O → 'Π ('Desc O) (λ D →
  '⟦ D ⟧₁ D '→ 'μ₁' O D)) ⟧
init' O D xs = init xs
```

We internalize the D argument by quantifying over a closed description (‘Desc). Because D is a description from the previous universe, the subsequent argument uses the lifting description interpretation function (‘$⟦_⟧_1$). Similarly, the codomain uses the lifting fixpoint constructor (‘μ_1’). Importantly, the codomain of init is internalized with the prime-variant of closed fixpoint constructor (‘μ_1’), defined over descriptions of the previous universe, not the non-prime fixpoint constructor

(‘μ_1), defined over descriptions of the current universe.

It is not obvious that the definition of our hierarchy needs fixpoints of descriptions in the current (‘μ_1) and previous (‘μ_1’) universes. It is also not obvious that the hierarchy needs to internalize the description interpretation function (‘$⟦_⟧_1$), for descriptions of the previous universe. However, our sanity check, in Appendix E, exposes that both ‘μ_1’ and ‘$⟦_⟧_1$ are necessary to internalize the kind signature of the init constructor. For reference, we also present the external type signature that the meaning of our internal type above expands to.

```
init' : (O : 'Set[ 0 ]) (D : 'Desc[ 0 ] O)
  → ⟦ « 0 | D » ⟧₁ « 0 | D » → μ₁ ⟦ 0 | O ⟧ « 0 | D »
init' O D xs = init xs
```

Lifting Functions We conclude this section by reflecting upon the internalization of the kind signatures for the pair (_,_) and initial algebra (init) constructors (as pair’ and init’), in the examples above.

The former is evidence that we need to quantify over the kind of closed types (‘Set), and then lift the quantifier to the kind level using the closed meaning function of types (‘$⟦_⟧$).

The latter is evidence that we need to quantify over the kind of closed descriptions (‘Desc), and then lift the quantifier to the kind level using the closed interpretation function (‘$⟦_⟧_1$) of descriptions, and the (lifting) closed fixpoint operator (‘μ_1’) of descriptions.

Hence, our sanity check in Appendix E, that the signature of all datatype constructors can be internalized in our closed universe hierarchy, drives the need for quantification over closed kinds (‘Set and ‘Desc). In turn, quantification over closed kinds drives results in types (i.e., the previous universe), which drives the need for lifting functions appropriate to each kind (‘$⟦_⟧$, ‘$⟦_⟧_1$, and ‘μ_1’). Thus, we recognize the sanity check in Appendix E as a good way to measure whether

we have appropriately closed our hierarchy, and are grateful for the structure that the check provides to the definition of our hierarchy.

As one final note, we emphasize that it is not enough that we can exhibit kind signatures for every datatype constructors. It is also important that the meaning of our closed kind signatures reduce to exactly the signatures expected by the underlying Agda constructors of our open type theory model.

8.3 LEVELED FULLY GENERIC FUNCTIONS

Chapter 7 demonstrates writing fully generic functions (like count, lookup and ast) over all *values* of the *Closed Inductive-Recursive Types* universe (of Section 6.2). In this section, we show how to write *leveled* fully generic functions, or fully generic functions at any level of the *Closed Hierarchy of Inductive-Recursive Universes* (of Section 8.2).

In Section 8.3.1, we patch fully generic count (of Section 7.1.2), converting it to work in level 0 of our hierarchy, over all *values* of types. Subsequently, in Section 8.3.2, we define fully generic Count in level 1 of our hierarchy, over all *types* of kinds. As we shall see, the Count function at level 1 must be defined in terms of the count function at level 0, because the values of level 0 are lifted to the type level 1, which can be expected because our universes form a *hierarchy*.

We only patch count to work at level 0 (and extend it to work at level 1), but other fully generic functions (like lookup and ast) can be similarly defined as *leveled* fully generic functions. Leveling a function primarily involves 2 things:

1. The type of the fully generic function must be *internalized* as a kind (i.e., we move from level 0, to the subsequent level, 1).
2. Additional cases must be handled, for the closed kinds ‘Set and ‘Desc, and their associated lifting functions (‘⟦_⟧, ‘⟦_⟧$_1$, and ‘μ_1’).

```
one : ⟦ 0 | ‘ℕ ⟧
one = suc zero

two : ⟦ 0 | ‘ℕ ⟧
two = suc one

_+_ : ⟦ 0 | ‘ℕ ‘→ ‘ℕ ‘→ ‘ℕ ⟧
init (true , tt) + m = m
init (false , n , tt) + m = n tt + m
```

Figure 8.1: Closed natural number definitions in universe level 0.

8.3.1 Counting in Universe Zero

Step 1 of patching the count function (defined over all values in Section 7.1.2), and the mutually defined counts function (defined over all algebraic arguments in Section 7.1.3), to be defined in level 0 of our hierarchy, is *internalizing* their signatures, as follows.

```
count : ⟦ 1 | ‘Π ‘Set (λ A → ‘⟦ A ⟧ ‘→ ‘⟦ ‘ℕ ⟧) ⟧
counts : ⟦ 1 | ‘Π ‘Set (λ O → ‘Π (‘Desc O) (λ D → ‘Π (‘Desc O) (λ R →
  ‘⟦ D ⟧₁ R ‘→ ‘⟦ ‘ℕ ⟧))) ⟧
```

Because count and counts quantify over kinds (‘Set and ‘Desc, respectively), they have internalized *kind* signatures (universe level 1). However, the A argument of count, and the returned natural number (‘ℕ) codomain are *types*, because they are lifted using ‘⟦_⟧. Similarly, ‘⟦_⟧$_1$ is used to lift the last argument of counts from the type level to the kind level. Hence, count and counts operate on *values*, classified by *types*, albeit lifted to the kind level in the signatures of count and counts.

Both count and counts now return *internalized* natural numbers (‘ℕ), hence we must patch the body of count from Section 7.1.2 and counts from Section 7.1.3, according to the table below. The left column of the table contains values *external*

```
count : ⟦ 1 | ‘Π ‘Set (λ A → ‘⟦ A ⟧ ‘→ ‘⟦ ‘ℕ ⟧) ⟧
count (‘Σ A B) (a , b) = one + count A a + count (B a) b
count (‘μ₁ O D) (init xs) = one + counts O D D xs
count ‘Set ()
count (‘Desc ()) ()
count (‘⟦ () ⟧) a
count (‘⟦ () ⟧₁ ()) xs
count (‘μ₁’ () ()) x
count A a = one

counts : ⟦ 1 | ‘Π ‘Set (λ O → ‘Π (‘Desc O) (λ D → ‘Π (‘Desc O) (λ R →
    ‘⟦ D ⟧₁ R ‘→ ‘⟦ ‘ℕ ⟧))) ⟧
counts O (‘σ A D) R (a , xs) = count A a + counts O (D a) R xs
counts O (‘δ ‘⊤ D) R (f , xs) = count (‘μ₁ O R) (f tt) +
    counts O (D (‘μ₂ R ∘ f)) R xs
counts O (‘δ A D) R (f , xs) = one + counts O (D (‘μ₂ R ∘ f)) R xs
counts O (‘ι o) R tt = one
```

Figure 8.2: Fully generic counting of values (count) and algebraic arguments (counts) in universe level 0.

to our closed hierarchical type theory, and the right side contains their *internal* equivalents.[3]

Closed Types Universe	Universe 0 in Hierarchy
$1 : \mathbb{N}$	one : ⟦ 0 | ‘ℕ ⟧
$+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$	+ : ⟦ 0 | ‘ℕ ‘→ ‘ℕ ‘→ ‘ℕ ⟧

The definitions of count and counts in universe level 0, which are the result of patching their equivalents in Section 7.1.2 and Section 7.1.3, are in Figure 8.2. Recall that step 2 of the patching process is to handle cases for the closed kinds

[3] We use the closed definition of natural numbers at level 0 from Section 8.2.2, and the closed definitions of one and +, appearing in the right column of the table, are defined in Figure 8.1.

(‘Set and ‘Desc), and their lifting functions (‘⟦_⟧, ‘⟦_⟧$_1$, and ‘μ_1’), in the definition of count.

In Figure 8.2, the first argument of count is ‘Set, and the second argument is its meaning (or lifting). However, at universe level 0 the meaning of ‘Set is ⊥, so the second argument is empty parentheses, which is Agda syntax for matching against an uninhabited argument. This makes sense intuitively because count at level 0 is defined over *values*, hence we do not need to define a case for counting *types* (inhabitants of ‘Set). The same is true for the ‘Desc case. Finally, each lifting function constructor (‘⟦_⟧, ‘⟦_⟧$_1$, and ‘μ_1’) takes a closed type or description as one of its arguments. Because we know that closed types and descriptions are not inhabited at universe level 0, we also do not need to define cases for the lifting functions.

8.3.2 Counting in Universe One

Previously (Section 8.3.1), we defined count and counts to count the inhabitants of universe level 0 in our closed hierarchy. Now, we define fully generic functions to count the inhabitants of universe level 1 in our closed hierarchy.

Counting Values Even though we think of level 1 as the level of *types*, there are copies of type constructors (like dependent pairs, or ‘Σ) at every level of our hierarchy, whose *values* we must be able to count. Thus, we mutually define (in Figure 8.3) Count for values at level 1, and Counts for algebraic arguments at level 1. Notice the capitalization of Count and Counts, indicating that they are the universe level 1 equivalents of count and counts (from universe level 0).

```
Count : ⟦ 2 | ‘Π ‘Set (λ A → ‘⟦ A ⟧ ‘→ ‘⟦ ‘⟦ ‘ℕ ⟧ ⟧) ⟧
Counts : ⟦ 2 | ‘Π ‘Set (λ O → ‘Π (‘Desc O) (λ D → ‘Π (‘Desc O) (λ R →
  ‘⟦ D ⟧₁ R ‘→ ‘⟦ ‘⟦ ‘ℕ ⟧ ⟧))) ⟧
```

Notice that because the internalized *superkind* signatures of Count and Counts

```
Count : ⟦ 2 | ‘Π ‘Set (λ A → ‘⟦ A ⟧ ‘→ ‘⟦ ‘⟦ ‘ℕ ⟧ ⟧) ⟧
Count (‘Σ A B) (a , b) = one + Count A a + Count (B a) b
Count (‘μ₁ O D) (init xs) = one + Counts O D D xs
Count ‘Set A = CountSet A
Count (‘Desc O) D = CountDesc O D
Count (‘⟦ A ⟧) a = count A a
Count (‘⟦ D ⟧₁ R) xs = counts _ D R xs
Count (‘μ₁’ O D) (init xs) = one + counts O D D xs
Count A a = one

Counts : ⟦ 2 | ‘Π ‘Set (λ O → ‘Π (‘Desc O) (λ D → ‘Π (‘Desc O) (λ R →
  ‘⟦ D ⟧₁ R ‘→ ‘⟦ ‘⟦ ‘ℕ ⟧ ⟧))) ⟧
Counts O (‘σ A D) R (a , xs) = Count A a + Counts O (D a) R xs
Counts O (‘δ ‘⊤ D) R (f , xs) = Count (‘μ₁ O R) (f tt) +
  Counts O (D (‘μ₂ R ∘ f)) R xs
Counts O (‘δ A D) R (f , xs) = one + Counts O (D (‘μ₂ R ∘ f)) R xs
Counts O (‘ι o) R tt = one
```

Figure 8.3: Fully generic counting of values (Count) and algebraic arguments (Counts) in universe level 1.

are at level 2, we must lift the return type of natural numbers twice (because ‘ℕ is defined in level 0). However, the A argument must only be lifted once, which lifts the quantified *kind* (‘Set at level 1) to level 2 (the level of the *superkind* signature). Recall (from Section 6.2) that the lifting constructor ‘⟦_⟧ is defined at every level of our universe hierarchy (so is ‘Σ), but ‘ℕ is only defined at level 0.

The definitions of Count and Counts are in Figure 8.3. All cases are the same as the level 0 count and counts variants of Figure 8.2, except for the kind (‘Desc and ‘Desc) and lifting (‘⟦_⟧, ‘⟦_⟧$_1$, and ‘μ_1’) cases. In the lifting cases, the inhabitant argument comes from the *previous* universe, so we count the lifted inhabitants using level 0 functions (count and counts). For the kind cases (‘Set and ‘Desc), the inhabitants are closed types and descriptions. Hence, we must additionally

mutually define (in Figure 8.4) CountSet to count types and CountDesc to count descriptions.

```
CountSet : ⟦ 1 | ‘Set  ‘→ ‘⟦ ‘ℕ ⟧ ⟧
CountSet (‘Σ A B) = two + CountSet A
CountSet (‘Π A B) = two + CountSet A
CountSet (‘Id A x y) = one + CountSet A + count A x + count A y
CountSet (‘μ₁ O D) = one + CountSet O + CountDesc O D
CountSet (‘Desc ())
CountSet (‘⟦ () ⟧)
CountSet (‘⟦ () ⟧₁ ())
CountSet (‘μ₁’ () ())
CountSet A = one

CountDesc : ⟦ 1 | ‘Π ‘Set (λ O → ‘Desc O  ‘→ ‘⟦ ‘ℕ ⟧) ⟧
CountDesc O (‘ι o) = one + count O o
CountDesc O (‘σ A D) = two + CountSet A
CountDesc O (‘δ A D) = two + CountSet A
```

Figure 8.4: Fully generic counting of types (CountSet) and algebraic arguments (CountDesc) in universe level 1.

Counting Types and Descriptions To write fully generic functions at level 1, to count closed types and descriptions, we must internalize their signatures as follows.

```
CountSet : ⟦ 1 | ‘Set  ‘→ ‘⟦ ‘ℕ ⟧ ⟧
CountDesc : ⟦ 1 | ‘Π ‘Set (λ O → ‘Desc O  ‘→ ‘⟦ ‘ℕ ⟧) ⟧
```

Notice that CountSet and CountDesc are defined in level 1. This is because they are applied to the inhabitants of the ‘Set and ‘Desc cases of Count (Figure 8.3). Because the inhabitants are classified as the meaning of the closed kind of ‘Set or ‘Desc, the inhabitants live at the previous level. Hence, while Count is defined at level 2, CountSet and CountDesc are defined at level 1.

The definitions of CountSet and CountDesc are in Figure 8.4. They count each type (e.g., `Σ) and description (e.g., `σ) the same way that count and counts (Figure 8.2) count values.

For example, the `Σ case of CountSet is counted as 2 plus a recursive call for the $A$ type. We count 1 for the `Σ itself, and add another 1 for the dependent and higher-order B argument, which we treat as a black box (just like we do when counting functions in Section 7.1.2, or infinitary arguments in Section 7.1.3). For the same reason, the `σ case of CountDesc is counted as 2 plus a recursive call for the A type. Once again, the dependent and higher-order D argument is treated as a black box.

Notice that the x and y arguments of the identity type `Id are actually *values*. Hence, we apply count to them, rather than CountSet. The same is true for $o$ in the `ι case of CountDesc. Finally, notice that the *kind* (`Set and `Desc) and lifting function cases of CountSet are undefined. This is because CountSet counts *types* at level 1, so kinds at level 2 are uninhabited. If we defined another version of count and all the associated function at universe level 3 (of superkinds), then the kind and lifiting cases of CountSet at level 3 would call their variants at level 2 (e.g., the `Set case of CountSet at level 3 would pass its argument to CountSet of level 2).

8.3.3 Leveled Generic Template

In Section 7.3.5, we conclude Chapter 7, on fully generic programming, with a template for writing fully generic functions over all types (in universe 0). We conclude this chapter similarly, but this time we present a generic template for writing fully generic functions over all types (in universe 1). In other words, we generalize the signatures of Section 8.3.2, requiring the mutual definition of 4 functions.

Generic : ⟦ 2 | `Π `Set (λ A → `⟦ $A$ ⟧ `→ ⋯) ⟧

```
Generics : ⟦ 2 | ‘Π ‘Set (λ O → ‘Π (‘Desc O) (λ D → ‘Π (‘Desc O) (λ R →
    ‘⟦ D ⟧₁ R ‘→ ···))) ⟧

GenericSet : ⟦ 1 | ‘Set ‘→ ··· ⟧
GenericDesc : ⟦ 1 | ‘Π ‘Set (λ O → ‘Desc O ‘→ ···) ⟧
```

These 4 functions are defined over different things, described below, but all functions inhabit universe level 1.

1. Generic is defined over all values.
2. Generics is defined over all algebraic arguments of the initial algebra.
3. GenericSet is defined over all types.
4. GenericSet is defined over all descriptions.

Recall (from Section 8.3.2) that the types that make up universe 0 are included in the collection of values of universe 1. Hence, Generic must call GenericSet (in the ‘Set case), as well as a version of lowercase generic (like count in Section 8.3.1) of universe 0 (in the ‘⟦_⟧ case).

The ellipses (···) in the first two functions (Generic and Generics) represents a closed type (‘Set[1]). The ellipses (···) in the next two functions (GenericSet and GenericDesc) represents a closed kind (‘Set[2]). If our leveled fully generic function has a dependent type, then we would need to define 8 functions instead of 4. The additional 4 functions would compute the types of the 4 functions given above. The additional 4 functions would be applied in the ellipses (···) positions of the 4 functions given above.

Part IV

Postlude

Chapter 9

RELATED WORK

The topic of this dissertation falls under the broad practice of generic programming, but we will only discuss work related to generic programming within dependent type theory. Namely, intrinsically type-safe generic programming as dependent functions over some universe, taking a code argument (A : Code) and a subsequent dependently typed argument, whose type is the meaning of the code ($[\![A]\!]$) within type theory:

$$\text{generic} : (A : \text{Code})\ (a : [\![A]\!]) \to \cdots$$

9.1 FIXED OPEN OR CLOSED UNIVERSES

By a *fixed* universe, we mean a universe that encodes some fixed collection of type formers, but does not support encoding user-declared datatypes. Generic programming over fixed universes, whether they are open (as in Section 2.2.2) or closed (as in Section 2.2.3), is standard dependently typed programming practice.

File Formats For example, Oury and Swierstra [49] demonstrate "The Power of Pi" (or dependently typed programming), by creating a file Format universe, and writing *fully generic* parse and print functions for all file formats that the universe encodes. The universe is closed under (among other things) dependent pair formation (whose code they call Read), as well as a base universe (U) encoding bits, characters, natural numbers, and even vectors.

Even though parse and print are *fully generic functions*, they are defined over a fixed universe of types. This makes sense for the problem at hand, where file formats should be able to use dependent pairs and vectors to encode the length of the remaining file format, after reading a natural number specifying said length. In their setting, it does not make sense to support arbitrary user-declared types when defining file formats. In contrast, our goal is to model an entire closed dependently typed programming language (as in Section 6.2 or Section 8.2), rather than file formats, so this dissertation concerns itself with a closed *extendable* universe (by user-declared datatypes).

Termination A more theoretical example of generic programming is Coquand's proof [10] that an operational semantics of type theory terminates. This is achieved using a logical relation defined as an inductive-recursive universe, which can be viewed as an extension of a universe of natural numbers (`ℕ), closed under dependent function formation (`Π). Below, we give the signature for the type of expressions (ε), the indexed logical relation type (Ψ), and the logical relation meaning function (ψ), used in Coquand's formal development.

```
data ε : Set where
mutual
  data Ψ : (A : ε) → Set where
  ψ : (A : ε) → Ψ A → (a : ε) → Set
```

The codes (Ψ) of the logical relation are additionally indexed by a syntax of expressions (A : ε). The codes are inhabited for all the expressions corresponding to types in the language. The meaning function (ψ) of the logical relation is indexed by two expressions, where the first represents the type (A) and the second represents values of that type (a). The meaning function is inhabited whenever the expression value is a valid member of the expression type.

The meaning function is also indexed by the result of applying the code type

former (Ψ) to the expression index representing the type (A), or evidence that the type is well-formed. One final difference between the logical relation and an ordinary universe of types, is that the logical relation also contains termination evidence, in the form of inhabitants of the operational semantics judgement (defined as a type that is indexed by expressions).

Once again, we emphasize that the logical relation for a dependent type theory can be considered a *universe*, albeit one with additional indexing and containing additional data in the form of termination witnesses. The fundamental theorem, used to prove that the operational semantics terminates, is defined over this universe (i.e., the logical relation is one of its arguments). Hence, the fundamental theorem can be seen as a *fully generic* function. Many lemmas used in the proof of termination can likewise be seen as fully generic functions. Finally, we note that even though these functions are fully generic, they operate over a fixed universe of natural numbers, closed under dependent function formation.

9.2 EXTENDABLE OPEN OR CLOSED WELL-ORDER UNIVERSES

Open Universes Morris [44] demonstrates generic programming over small *indexed* containers in an *open* universe. Because indexed containers can represent arbitrary user-declared datatypes, the universe is also *extendable*.

Morris writes generic functions, like map, over the open universe of indexed containers. This corresponds to writing generic functions over the open universe of inductive-recursive types in Section 5.4.2, because small induction-recursion and small indexing are equivalent [31].

Recursive containers are represented using the W type of well-orderings, which is the fixpoint of containers. As we explained in Section 4.2.3, W types inadequately encode first-order types in intensional type theory, which is why we use the more complicated (but adequate) algebraic semantics of Section 5.4.4, defined in terms of Desc and μ_1.

Closed Universes We expect that it would be straightforward to extend the generic functions that Morris wrote over an open universe of containers, to operate over a closed universe of well-orderings (like the universe in Section 4.2.1). Once again, we were not interested in this option for adequacy reasons (Section 4.2.3).

9.3 EXTENDABLE OPEN ALGEBRAIC UNIVERSES

There is a lot of work on generic programming over an *open* algebraic universe, similar to the one in Section 5.4.2. It should be possible to extend any such generic functions, over an *open* universe, to be fully generic, over a *closed* universe (or hierarchy of universes), using techniques from Chapter 7 (and Section 8.3).

Universal Algebra Benke et al. [6] perform generic programming in the domain of universal algebra. Various restrictions of the open inductive-recursive universe of Section 5.4.2 are used for each algebra (e.g., one-sorted term algebras, many-sorted term algebras, parameterized term algebras, etc.). Some of these algebras restrict the universe to be finitary, some remain infinitary, but all of them restrict the use of induction-recursion. As they state, their work could have been instead defined as restrictions over a universe of indexed inductive types without induction-recursion.

Induction Principles Chapman et al. [7] define Descriptions for indexed dependent types (without induction-recursion). Defining generic induction principles for types encoded by Descriptions requires a computational argument type for all the inductive hypotheses (All, also called Hyps). Although Desc is not inductive-recursive, it is still infinitary so generic functions over such types, like ind, share many of the same properties as our generic functions.

Our previous work [17] expands upon the work of Chapman et al. [7], defining an alternative interface to induction as generic type-theoretic eliminators for Descriptions. Defining these eliminators involves several nested constructions,

where both computational argument types (to collect inductive hypotheses) and return types (to produce custom eliminator types for each description) are used for information retrieval but not modification of infinitary descriptions.

Ornaments McBride [41] builds a theory of Ornaments on top of Descriptions for indexed dependent types (without induction-recursion). Ornaments allow a description of one type (such as a Vector) to be related to another type (such as a List) such that a forgetful map from the more finely indexed type to the less finely indexed type can be derived as a generic function. This allows the length function over lists (List) to be derived from the length function of (the more finely) indexed vectors (Vec). Dagand and McBride [13] expand this work to also work in the opposite direction, allowing functions over more finely indexed types to be derived from functions over less finely indexed types, after providing some structured missing information.

Disjointness and Injectivity Goguen et al. [29] demonstrate how to elaborate a high-level syntax of dependent pattern matching to low-level uses of eliminators. Part of this elaboration process depends upon proofs that constructors are injective and disjoint. McBride et al. [43] define these proofs externally, at the level of metatheory. Dagand [12] also internalizes these proofs, as generic programs over the open universe of algebraic datatypes (using Desc and μ).

Strictly Positive Families In addition to writing generic functions over open container-based datatype encodings, Morris also writes generic functions over an open universe of "Strictly Positive Families" (whose type is called SPT). He writes functions like generic map, a generic decision procedure for equality (over the first-order subset of the universe), and generic zipper operations. The SPT universe can be considered an alternative way to define Desc and μ. Due to the way

SPT is defined, you can write functions that can make recursive calls on inductive arguments of varying types, in a way that feels very similar to fully generic programming. Nonetheless, ultimately SPT is still an open universe, as function domains and infinitary domains are still encoded using the open Set type.

In Section 7.1, we define fully generic count to specialize the way it operates over inductive arguments (infinitary argument whose domain is the unit type ‘⊤), as opposed to truly infinitary arguments (whose domain is a type other than unit). This would not be possible in the SPT universe, because we could not match on the domain argument (of open kind Set, rather than closed type ‘Set).

Static Constructors and Arguments Sijsling [50] defines an open algebraic universe, using a “static” variant of the datatype of descriptions (Desc). This universe statically encodes the structure of constructors and their arguments, so that we statically know the number of constructors and arguments of a datatype. In contrast, the type of the second argument of the σ constructor (of Section 5.3.2), depends on the value of the first argument. Hence, we cannot statically determine the number of remaining constructor arguments, encoded by the second argument of σ, because its type may depend on the first argument of σ (i.e., a value, only dynamically available).

Sijsling reflects datatype declarations written in high-level Agda (using Agda’s reflection machinery), and uses the reflected declarations to automatically derive encodings of the datatypes in terms of his static Desc. He then writes generic programs over his static Desc, some of which can be automatically converted between their high-level Agda representations and the low-level static Desc-based representations.

Sijsling leaves extending his static Desc to account for infinitary arguments and induction-recursion as future work, which we believe is possible. We also do not foresee any problems with defining a closed universe in terms of such static Desc

types, by applying our closing procedure from Section 6.3.

Arity-Generic and Datatype-Generic Programming Weirich and Casinghino [53] demonstrate writing arity-generic and datatype-generic functions, such as a map function for any type (i.e., the datatype-generic part) with any number of datatype parameters (i.e., the arity-generic part). They also define generic zip and equality functions. Note that all of these functions are traditional generic programs, because they do not recurse into the structure of datatype parameters. Instead, functions like equality are parameterized by a function to compare the values of the parameterized types.

The universe used by Weirich and Casinghino captures the class of datatypes that can be built from non-dependent functions, the unit type, natural numbers, non-dependent pairs, and disjoint unions. Some indexed types can be built this way, like vectors. But, their universe cannot represent indexed types whose constructor arguments have indices that are structurally larger than the index returned by the constructor. This is because their indexed types are derived as computational families (Section 2.1.11), as a non-dependent function in their universe, so a function deriving such an indexed type would not terminate. Additionally, their universe cannot represent indexed types with dependencies between indices, because their function-space is non-dependent.

9.4 PREVIOUS WORK

Now we discuss how the contributions of this dissertation relate to our previously published work.

Closed Universe Zero and Fully Generic Programming In a previous publication [18], Sheard and I defined the closed universe of inductive-recursive algebraic types, and wrote fully generic functions over the universe. That work is the

basis of Chapter 6 and Chapter 8. An important contribution of our dissertation from Chapter 6, not present in our previous publication [18], is the generic procedure to close *any* universe of kinds (Section 6.3).

The fully generic function count (Section 7.1) and ast (Section 7.3) functions of Chapter 6 are also novel to this dissertation. In Section 7.2, we define a generic lookup function, that takes a finite set (Fin) argument, which is indexed by the count of the argument being looked up. We also define a lookup function in our previous publication, but it is indexed by a custom index type (unique to each type being looked up), rather than using Fin and a dependent application of count. Our previous publication also features a generic update function. While this dissertation treats higher-order arguments as black-boxes, our previous publication [18] uses domain supplements (Section 3.4.5) to also recurse into higher-order arguments.

Closed Universe Hierarchy and Leveled Fully Generic Programming In another previous publication [16], Sheard and I defined a closed hierarchy of algebraic (but not infinitary or inductive-recursive) types. That work is the basis of Chapter 8. The novel part of Chapter 8 is adapting McBride's *Closed Hierarchy of Well-Order Universes* [42] (reviewed in Section 8.1.3) to a *Closed Hierarchy of Inductive-Recursive Universes* (presented in Section 8.2).

While our previous publication featured both description lifting functions, '⟦_⟧$_1$ and 'μ_1', it did *not* feature the non-lifting fixpoint operator 'μ_1. At the time, we did not know how to adequately represent datatypes of the current universe level. This resulted in needing to inadequately define certain types at one level higher in the hierarchy, so that they may be defined in terms of the lifting fixpoint 'μ_1'.

My (i.e., Diehl's) novel solution to this problem appears in Chapter 8, where I add a non-lifting fixpoint 'μ_1, whose argument is a *mutually* defined DescForm. Hence, the novelty of Chapter 8 is combining the idea of mutually defined code types ('Set and 'Set) and mutually defined translation functions (⟦_⟧ and «_»),

from Chapter 6, with the idea of description lifting functions (‘$[\![_]\!]_1$ and ‘μ_1’) from our previous publication [16].

Chapter 10

FUTURE WORK

This dissertation demonstrates that leveled fully generic programming is possible, using a universe modeling a closed dependently typed language supporting user-declared datatypes. But, there is still much work left to do! We discuss a small slice of this future work, below.

10.1 UNIVERSE POLYMORPHISM

In Section 8.2.2, we define the *type* (in universe 0) of closed natural numbers, whose signature is a *kind* (in universe 1).

```
‘ℕ : ⟦ 1 | ‘Set ⟧
```

In Section 8.3.1, we define fully generic count over all values of all types (in universe 0), whose signature is also a kind (in universe 1). When we use the type of natural numbers in the kind signature of count, it must be lifted to the kind level via ‘⟦_⟧.

```
count : ⟦ 1 | ‘Π ‘Set (λ A → ‘⟦ A ⟧ ‘→ ‘⟦ ‘ℕ ⟧) ⟧
```

In Section 8.3.2, we define fully generic Count over all types of all kinds (in universe 1), whose signature is a superkind (in universe 2). When we use the type of natural numbers in the superkind signature of Count, it must be lifted to the kind level by using ‘⟦_⟧ twice.

```
Count : ⟦ 2 | ‘Π ‘Set (λ A → ‘⟦ A ⟧ ‘→ ‘⟦ ‘⟦ ‘ℕ ⟧ ⟧) ⟧
```

Types like dependent pairs (‘Σ) are built into the universe, and appear at

every level of the hierarchy. Therefore, we must handle the ‘Σ case of Count (in universe 1) in exactly the same way that we handled it for count (in universe 0). Furthermore, if we want to count all kinds of superkinds (in universe 2), we must define yet fully generic counting function (and so on, for every level). We could eliminate a lot of duplications by defining both algebraic datatypes and functions *universe polymorphically*, so they can be instantiated at any level of the universe.

```
‘ℕ : (ℓ : ℕ) → ⟦ suc ℓ | ‘Set ⟧
count : (ℓ : ℕ) → ⟦ suc ℓ | ‘Π ‘Set (λ A → ‘⟦ A ⟧ ‘→ ‘ℕ (suc ℓ)) ⟧
```

Notice that the natural number codomain of count does not need to be lifted, because we can just request a version of the natural numbers at the succesor to level ℓ. Also notice that we can define count once at every level, so we do not need to separately define Count.

Unfortunately, universe polymorphic definitions rely on quantifying over levels in our metalanguage. In other words, universe polymorphic definitions do not model fully generic programs that we could write in our modeled closed dependently typed language. For future work, we would like to add universe level quantification as a code of our universe, so that the types of definitions like ‘ℕ and count can be *internalized* (i.e., made to appear within the brackets).

10.2 LARGE INDUCTION-RECURSION

In this dissertation, we close over inductive-recursive types (in Section 6.2), but they are *small*. Inductive-recursive types are small if the codomain of the decoding function can be any type, but it cannot be any kind (like ‘Set or ‘Desc). We are not sure if it is possible to define a closed universe of *large* inductive-recursive types, but we would like to try. It may be the case that we need a more expressive type theory, like Homotopy Type Theory [52], to close over large inductive-recursive types of Martin-Löf’s type theory [39].

If we are able to close over large inductive-recursive types, then we would need to encode indexed inductive-recursive algebraic types. This is because the isomorphism between indexed types and inductive-recursive types only holds in the small case [31], so inductive-recursive algebraic types would not be enough.

Finally, note that we cannot achieve large induction-recursion simply by moving up a universe level. At universe level 1, the codomain of a small decoding function could be any kind, but a large decoding function would allow the codomain to be any superkind.

10.3 INDUCTION-INDUCTION

We close over small inductive-recursive types in Section 6.2. We have also applied our closing procedure (Section 6.3) to close over an encoding of small indexed inductive-recursive types [24]. This dissertation does not cover the closed universe of indexed inductive-recursive because no problems arise when applying our closing procedure. Nordvall has shown how to formally model inductive-inductive types [46]. An inductive-inductive type is defined as a pair of mutually defined types, where the second type is indexed by the first. We have not yet attempted to close over a universe of inductive-inductive types, but we plan to in future work.

10.4 HIGH-LEVEL GENERIC PROGRAMMING

In Chapter 7, we mention that we can hide our algebraic encodings via smart constructors and pattern synonyms, when defining *concrete* functions (i.e., over concrete datatypes). However, we need to understand the underlying initial-algebra base encoding, when defining *fully generic functions*.

McBride [29] defines how to elaborate dependent pattern matching, a high-level language feature, to eliminators, which can be considered low-level induction principles of a core language. We would like to explore implementing a closed

dependently typed language. It would be nice if we had a high-level feature for writing fully generic functions, that could be elaborated to the fully generic functions of this dissertation, which explicitly match on low-level encodings (like the initial algebra).

Just as McBride allows users to define functions by dependent pattern matching, without understanding how to program directly with eliminators, we would like users to be able to define fully generic functions, without understanding our closed universe model. Finally, note that Dagand [14] has already shown how to elaborate high-level datatype declarations to their description-based (Desc) encodings.

10.5 EFFICIENT IMPLEMENTATION

Al-Sibahi [2] shows how to efficiently implement traditional generic programming over a (description-based) open algebraic universe of non-infinitary indexed types.[1] He uses partial evaluation to remove most of the overhead associated with encoding datatypes as fixpoints of functor descriptions. It would be interesting to explore extending Al-Sibahi's work to fully generic programming over a closed universe, to see if any complications arise in the closed setting. We expect that his open-universe optimizations will continue to work in the closed-universe setting, and we hope that further optimizations will be possible in the closed-universe setting.

10.6 TERMINATION OF INTENSIONAL CLOSED TYPE THEORY

In our related work (Chapter 9), we discus how Coquand [10] proves that an operational semantics of type theory, consisting of the natural numbers closed

[1] Al-Sibani also demonstrates a form of fully generic programming by writing functions like a generic pretty printer (gshow). Rather than closing the universe of descriptions, he adds extra arguments that constrain non-inductive constructor arguments (inspired by type class constraints). Unlike fully generic programming over a closed universe, custom constraints must be defined for each fully generic function.

under dependent function formation, terminates.

The logical relation of this termination proof can be considered an extended version of the closed universe model of natural numbers closed under dependent function formation. We would like to explore extending our model of closed inductive-recursive types (Appendix C) to a logical relation (following Coquand's approach), and proving that an operational semantics of closed type theory, supporting user-declared datatypes (via Desc and μ_1), terminates.

10.7 INDUCTIVE DEFINITIONS OVER INFINITARY DOMAIN

Our fully generic count (Section 7.1), lookup (Section 7.2), and ast (Section 7.3) functions all treat the case of inductive arguments of the initial algebra as a special case of infinitary arguments. Being able to pattern match against the domain of an infinitary argument, and ensuring that it is the unit type (‘⊤), demonstrates the power of closed type theory (because we can match against a type).

In our previous work [18], we have also defined higher-order domain supplements (Section 3.4.5) that allow us to write fully generic functions over higher-order data (like the body of functions, or truly infinitary arguments). However, neither the definition of count in Section 7.1 (which requires matching against ‘⊤), nor the fully generic functions using higher-order domain supplements [18], are defined by *induction* on the higher-order domains (like the domain of functions or the domain of infinitary arguments). In the future, we would like to explore what horizons have opened up to us now that our closed universe allows us to write functions by induction on closed higher-order domains.

Chapter 11

CONCLUSION

Generic programming, within dependently typed programming languages, over a universe closed under a fixed collection of type formers (e.g., Section 4.1) has a rich history. If we consider such a universe to be a model of a *closed* dependently typed programming language, then users of that language may use its fixed collection of types, but may not declare their own domain-specific types.

Inspired by categorical models of algebraic semantics, which model algebraic datatypes as least-fixed points of pattern functors, type theorists have also defined formal models (i.e., in type theory) of algebraic semantics. We can view strictly-positive polynomial functors (Desc) as codes of a universe, whose meaning is their fixpoints (μ_1). Generic dependently typed programming over this algebraic universe also has a rich history. Algebraic semantics is modeled as an *open* universe, which grows as users of the underlying open type theory declare new datatypes.

The *first* major contribution of this dissertation (Chapter 6) is creating a closed universe, modeling the types of a *closed* dependently typed programming language that supports user-declared datatypes (‘Desc). We still do this by defining a universe closed under a fixed collection of type formers, but one of the type formers is a *closed* variant of the fixpoint operator (‘μ_1) from algebraic semantics. This variant is parameterized by a mutually defined *closed* variant of strictly-positive polynomial functors (‘Desc).

The *second* major contribution of this dissertation (Chapter 7) is demonstrating what we call *fully generic programming*. Fully generic functions are defined over a closed universe including user-declared datatypes. They can be defined once,

working over all current datatypes, but they continue to work as users declare additional datatypes in the future.

The *third* major contribution of this dissertation (Chapter 8) is extending the model of closed types (supporting user-declarations) to also model closed kinds, closed superkinds, etc. Hence, we model a closed *hierarchy* of universes supporting user-declared datatypes. The closed hierarchy of universes models a closed dependently typed programming language with a universe hierarchy, supporting user-declared datatypes at every level of the hierarchy. We also demonstrate leveled fully generic programming, or writing fully generic functions at any level in the universe hierarchy (over values, types, kinds, superkinds, etc.). This achieves our goal, of modeling fully generic programming in a closed dependently typed programming language, supporting user-declared datatypes.

REFERENCES

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[2] Ahmad Salim Al-Sibahi. The Practical Guide to Levitation. Master's thesis, IT University of Copenhagen, 2014.

[3] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Generic Programming. Proceedings of the IFIP TC2 Working Conference on Generic Programming*, pages 1–20. Springer, 2003. Held in Schloss Dagstuhl.

[4] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming languages meets program verification*, PLPV '07, pages 57–68, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-677-6. doi: 10.1145/1292597.1292608. URL `http://doi.acm.org/10.1145/1292597.1292608`.

[5] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming. In *International School on Advanced Functional Programming*, pages 28–115. Springer, 1998.

[6] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.*, 10(4):265–289, 2003.

[7] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The Gentle Art of Levitation. In *Proceedings of the 15th ACM SIGPLAN*

International Conference on Functional Programming, ICFP '10, pages 3–14, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: 10.1145/1863543.1863547. URL `http://doi.acm.org/10.1145/1863543.1863547`.

[8] Alonzo Church. *The calculi of lambda-conversion*, volume 6. Princeton University Press, 1941.

[9] Robert L. Constable, Stuart F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, Scott F. Smith, James T. Sasaki, and S. F. Smith. *Implementing Mathematics with The Nuprl Proof Development System*. Prentice-Hall, 1986.

[10] Catarina Coquand. A realizability interpretation of Martin-Löf's type theory. In *Twenty-Five Years of Constructive Type Theory. Proceedings of a Congress Held in Venice*, Oxford, 1998. Clarendon Press.

[11] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG-88*, pages 50–66. Springer, 1990.

[12] Pierre-Evariste Dagand. *A Cosmology of Datatypes*. PhD thesis, University of Strathclyde, 2013.

[13] Pierre-Evariste Dagand and Conor McBride. Transporting Functions Across Ornaments. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 103–114, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364544. URL `http://doi.acm.org/10.1145/2364527.2364544`.

[14] Pierre-Evariste Dagand and Conor Mcbride. Elaborating inductive definitions. In *JFLA-Journées francophones des langages applicatifs*, 2013.

[15] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[16] Larry Diehl and Tim Sheard. Leveling Up Dependent Types: Generic Programming over a Predicative Hierarchy of Universes. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming*, DTP '13, pages 49–60, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2384-0. doi: 10.1145/2502409.2502414. URL `http://doi.acm.org/10.1145/2502409.2502414`.

[17] Larry Diehl and Tim Sheard. Generic constructors and eliminators from descriptions: type theory as a dependently typed internal DSL. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*, pages 3–14. ACM, 2014.

[18] Larry Diehl and Tim Sheard. Generic Lookup and Update for Infinitary Inductive-recursive Types. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, pages 1–12, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4435-7. doi: 10.1145/2976022.2976031. URL `http://doi.acm.org/10.1145/2976022.2976031`.

[19] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. *Logical frameworks*, 2:6, 1991.

[20] Peter Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994.

[21] Peter Dybjer. Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoretical Computer Science*, 176(1-2):329–335, 1997.

[22] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *The Journal of Symbolic Logic*, 65(02):525–549, 2000.

[23] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications, volume 1581 of Lecture Notes in Computer Science*, pages 129–146. Springer, 1999.

[24] Peter Dybjer and Anton Setzer. Indexed induction-recursion. In *Proof Theory in Computer Science*, pages 93–113. Springer, 2001.

[25] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

[26] Nicola Gambino and Martin Hyland. Wellfounded trees and dependent polynomial functors. In *International Workshop on Types for Proofs and Programs*, pages 210–225. Springer, 2003.

[27] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot, 2006.

[28] Jean-Yves Girard. *Functional interpretation and elimination of superior order arithmetic breaks*. PhD thesis, Universite Paris VII, 1972.

[29] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*, pages 521–540. Springer, 2006.

[30] Cordelia V Hall, Kevin Hammond, Simon L Peyton Jones, and Philip L Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996.

[31] Peter Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. Small induction recursion. In *International Conference on Typed Lambda Calculi and Applications*, pages 156–172. Springer, 2013.

[32] Ralf Hinze. *Generic Programs and Proofs*. PhD thesis, Universität Bonn, 2000.

[33] William A Howard. The formulae-as-types notion of construction. In *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, volume 44, pages 479–490. Academic Press, 1980.

[34] Antonius JC Hurkens. A simplification of Girard's paradox. In *International Conference on Typed Lambda Calculi and Applications*, pages 266–278. Springer, 1995.

[35] Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[36] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell workshop*, pages 1–16, 1997.

[37] Per Martin-Löf. Hauptsatz for the intuitionistic theory of iterated inductive definitions. *Studies in Logic and the Foundations of Mathematics*, 63:179–216, 1971.

[38] Per Martin-Löf. An intuitionistic theory of types: Predicative part. *Studies in Logic and the Foundations of Mathematics*, 80:73–118, 1975.

[39] Per Martin-Löf. Intuitionistic type theory. *Notes by Giovanni Sambin*, 1984. Bibliopolis, Naples.

[40] Conor McBride. W-types: good news and bad news. Blog post, March 2010. URL `http://mazzo.li/epilogue/index.html%3Fp=324.html`.

[41] Conor McBride. Ornamental algebras, algebraic ornaments. 2011.

[42] Conor McBride. Hier Soir, an OTT Hierarchy. Blog post, November 2011. URL `http://mazzo.li/epilogue/index.html%3Fp=1098.html`.

[43] Conor McBride, Healfdene Goguen, and James McKinna. A few constructions on constructors. In *Types for Proofs and Programs*, pages 186–200. Springer, 2006.

[44] Peter Morris. *Constructing Universes for Generic Programming*. PhD thesis, University of Nottingham, 2007.

[45] Bengt Nordström, Kent Petersson, and Jan M Smith. *Programming in Martin-Löf's type theory*, volume 85. Oxford University Press, 1990.

[46] Fredrik Nordvall Forsberg. *Inductive-inductive definitions*. PhD thesis, Swansea University, 2013.

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

[48] Ulf Norell. Dependently Typed Programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-420-1. doi: 10.1145/1481861.1481862. URL `http://doi.acm.org/10.1145/1481861.1481862`.

[49] Nicolas Oury and Wouter Swierstra. The Power of Pi. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 39–50, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-919-7. doi: 10.1145/1411204.1411213. URL `http://doi.acm.org/10.1145/1411204.1411213`.

[50] Yorick Sijsling. Generic programming with ornaments and dependent types. Master's thesis, Utrecht University, 2016.

[51] Christopher Strachey. Fundamental Concepts in Programming Languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, April 2000. ISSN 1388-3690. doi: 10.1023/A:1010000313106. URL `http://dx.doi.org/10.1023/A:1010000313106`.

[52] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[53] Stephanie Weirich and Chris Casinghino. Arity-generic datatype-generic programming. In *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification*, PLPV '10, pages 15–26, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-890-2. doi: 10.1145/1707790.1707799. URL `http://doi.acm.org/10.1145/1707790.1707799`.

Appendix A

OPEN NON-ALGEBRAIC TYPES

```
data ⊥ : Set where

record ⊤ : Set where
  constructor tt

data Bool : Set where
  true false : Bool

infixr 4 _,_
record Σ (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁

data Id (A : Set) (x : A) : A → Set where
  refl : Id A x x
```

Appendix B

OPEN UNIVERSE OF ALGEBRAIC TYPES

```
data Desc (O : Set) : Set₁ where
  ι : (o : O) → Desc O
  σ : (A : Set) (D : A → Desc O) → Desc O
  δ : (A : Set) (D : (A → O) → Desc O) → Desc O

mutual
  ⟦_⟧₁ : {O : Set} (D R : Desc O) → Set
  ⟦ ι o ⟧₁ R = ⊤
  ⟦ σ A D ⟧₁ R = Σ A (λ a → ⟦ D a ⟧₁ R)
  ⟦ δ A D ⟧₁ R = Σ (A → μ₁ _ R) λ f → ⟦ D (λ a → μ₂ R (f a)) ⟧₁ R

  ⟦_⟧₂ : {O : Set} (D R : Desc O) → ⟦ D ⟧₁ R → O
  ⟦ ι o ⟧₂ R tt = o
  ⟦ σ A D ⟧₂ R (a , xs) = ⟦ D a ⟧₂ R xs
  ⟦ δ A D ⟧₂ R (f , xs) = ⟦ D (λ a → μ₂ R (f a)) ⟧₂ R xs

  data μ₁ (O : Set) (D : Desc O) : Set where
    init : ⟦ D ⟧₁ D → μ₁ O D

  μ₂ : {O : Set} (D : Desc O) → μ₁ O D → O
  μ₂ D (init xs) = ⟦ D ⟧₂ D xs
```

Appendix C

CLOSED UNIVERSE OF ALGEBRAIC TYPES

```
mutual
  data `Set : Set where
    `⊥ `⊤ `Bool `String : `Set
    `Σ `Π : (A : `Set) (B : ⟦ A ⟧ → `Set) → `Set
    `Id : (A : `Set) (x y : ⟦ A ⟧) → `Set
    `μ₁ : (O : `Set) (D : `Desc O) → `Set

  ⟦_⟧ : `Set → Set
  ⟦ `⊥ ⟧ = ⊥
  ⟦ `⊤ ⟧ = ⊤
  ⟦ `Bool ⟧ = Bool
  ⟦ `String ⟧ = String
  ⟦ `Σ A B ⟧ = Σ ⟦ A ⟧ (λ a → ⟦ B a ⟧)
  ⟦ `Π A B ⟧ = (a : ⟦ A ⟧) → ⟦ B a ⟧
  ⟦ `Id A x y ⟧ = Id ⟦ A ⟧ x y
  ⟦ `μ₁ O D ⟧ = μ₁ ⟦ O ⟧ « D »

  data `Desc (O : `Set) : Set where
    `ι : (o : ⟦ O ⟧) → `Desc O
    `σ : (A : `Set) (D : ⟦ A ⟧ → `Desc O) → `Desc O
    `δ : (A : `Set) (D : (o : ⟦ A ⟧ → ⟦ O ⟧) → `Desc O)
      → `Desc O

  «_» : {O : `Set} → `Desc O → Desc ⟦ O ⟧
  « `ι o » = ι o
  « `σ A D » = σ ⟦ A ⟧ (λ a → « D a »)
  « `δ A D » = δ ⟦ A ⟧ (λ o → « D o »)
```

Appendix D

CLOSED HIERARCHY OF UNIVERSES

```
record Level : Set₁ where
  field
    SetForm : Set
    ⟦_/_⟧ : (A : SetForm) → Set
    DescForm : (O : SetForm) → Set
    ⟦_/_⟧₁ : {O : SetForm} (D R : DescForm O) → Set
    μ₁' : (O : SetForm) (D : DescForm O) → Set

mutual
  data SetForm (ℓ : Level) : Set where
    `⊥ `⊤ `Bool `String : SetForm ℓ
    `Σ `Π : (A : SetForm ℓ) (B : ⟦ ℓ / A ⟧ → SetForm ℓ) → SetForm ℓ
    `Id : (A : SetForm ℓ) (x y : ⟦ ℓ / A ⟧) → SetForm ℓ
    `μ₁ : (O : SetForm ℓ) (D : DescForm ℓ O) → SetForm ℓ
    `Set : SetForm ℓ
    `⟦_⟧ : Level.SetForm ℓ → SetForm ℓ
    `Desc : Level.SetForm ℓ → SetForm ℓ
    `⟦_⟧₁ : {O : Level.SetForm ℓ} (D R : Level.DescForm ℓ O) → SetForm ℓ
    `μ₁' : (O : Level.SetForm ℓ) (D : Level.DescForm ℓ O) → SetForm ℓ

  ⟦_/_⟧ : (ℓ : Level) → SetForm ℓ → Set
  ⟦ ℓ / `⊥ ⟧ = ⊥
  ⟦ ℓ / `⊤ ⟧ = ⊤
  ⟦ ℓ / `Bool ⟧ = Bool
  ⟦ ℓ / `String ⟧ = String
  ⟦ ℓ / `Σ A B ⟧ = Σ ⟦ ℓ / A ⟧ (λ a → ⟦ ℓ / B a ⟧)
  ⟦ ℓ / `Π A B ⟧ = (a : ⟦ ℓ / A ⟧) → ⟦ ℓ / B a ⟧
  ⟦ ℓ / `Id A x y ⟧ = Id ⟦ ℓ / A ⟧ x y
  ⟦ ℓ / `μ₁ O D ⟧ = μ₁ ⟦ ℓ / O ⟧ « ℓ / D »
```

```
    ⟦ ℓ / ‘Set ⟧ = Level.SetForm ℓ
    ⟦ ℓ / ‘⟦ A ⟧ ⟧ = Level.⟦ ℓ / A ⟧
    ⟦ ℓ / ‘Desc O ⟧ = Level.DescForm ℓ O
    ⟦ ℓ / ‘⟦ D ⟧₁ R ⟧ = Level.⟦ ℓ / D ⟧₁ R
    ⟦ ℓ / ‘μ₁’ O D ⟧ = Level.μ₁’ ℓ O D

    data DescForm (ℓ : Level) (O : SetForm ℓ) : Set where
      ‘ι : (o : ⟦ ℓ / O ⟧) → DescForm ℓ O
      ‘σ : (A : SetForm ℓ) (D : ⟦ ℓ / A ⟧ → DescForm ℓ O) → DescForm ℓ O
      ‘δ : (A : SetForm ℓ) (D : (o : ⟦ ℓ / A ⟧ → ⟦ ℓ / O ⟧) → DescForm ℓ O)
        → DescForm ℓ O

    «_/_» : (ℓ : Level) {O : SetForm ℓ} → DescForm ℓ O → Desc ⟦ ℓ / O ⟧
    « ℓ / ‘ι o » = ι o
    « ℓ / ‘σ A D » = σ ⟦ ℓ / A ⟧ (λ a → « ℓ / D a »)
    « ℓ / ‘δ A D » = δ ⟦ ℓ / A ⟧ (λ o → « ℓ / D o »)

level : (ℓ : ℕ) → Level
level zero = record
    { SetForm = ⊥
    ; ⟦_/_⟧ = λ()
    ; DescForm = λ O → ⊥
    ; ⟦_/_⟧₁ = λ ()
    ; μ₁’ = λ ()
    }
level (suc ℓ) = record
    { SetForm = SetForm (level ℓ)
    ; ⟦_/_⟧ = λ A → ⟦ level ℓ / A ⟧
    ; DescForm = DescForm (level ℓ)
    ; ⟦_/_⟧₁ = λ D R → ⟦ « level ℓ / D » ⟧₁ « level ℓ / R »
    ; μ₁’ = λ O D → μ₁ ⟦ level ℓ / O ⟧ « level ℓ / D »
    }

‘Set[_] : ℕ → Set
‘Set[ ℓ ] = SetForm (level ℓ)

⟦_|_⟧ : (ℓ : ℕ) → ‘Set[ ℓ ] → Set
⟦ ℓ | A ⟧ = ⟦ level ℓ / A ⟧
```

```
‘Desc[_] : (ℓ : ℕ) → ‘Set[ ℓ ] → Set
‘Desc[ ℓ ] O = DescForm (level ℓ) O

«_|_» : (ℓ : ℕ) {O : ‘Set[ ℓ ]} → ‘Desc[ ℓ ] O → Desc ⟦ ℓ | O ⟧
« ℓ | D » = « level ℓ / D »
```

Appendix E

INTERNALIZED CONSTRUCTOR SIGNATURES

```
bot' : ⟦ 0 | '⊥ '→ '⊥ ⟧
bot' p = p

tt' : ⟦ 0 | '⊤ ⟧
tt' = tt

true' : ⟦ 0 | 'Bool ⟧
true' = true

false' : ⟦ 0 | 'Bool ⟧
false' = false

pair' : ⟦ 1 | 'Π 'Set (λ A → 'Π ('⟦ A ⟧ '→ 'Set) (λ B →
    'Π '⟦ A ⟧ (λ a → 'Π '⟦ B a ⟧ (λ b →
    'Σ '⟦ A ⟧ (λ a → '⟦ B a ⟧))))) ⟧
pair' A B a b = a , b

lambda' : ⟦ 1 | 'Π 'Set (λ A → 'Π ('⟦ A ⟧ '→ 'Set) (λ B →
    'Π ('Π '⟦ A ⟧ (λ a → '⟦ B a ⟧)) (λ f →
    'Π '⟦ A ⟧ (λ a → '⟦ B a ⟧)))) ⟧
lambda' A B f = λ a → f a

refl' : ⟦ 1 | 'Π 'Set (λ A → 'Π '⟦ A ⟧ (λ a → 'Id '⟦ A ⟧ a a)) ⟧
refl' A a = refl

init' : ⟦ 1 | 'Π 'Set (λ O → 'Π ('Desc O) (λ D →
    '⟦ D ⟧₁ D '→ 'μ₁' O D)) ⟧
init' O D xs = init xs
```