A Cosmology of Datatypes

Reusability and Dependent Types

Pierre-Évariste Dagand

Doctor of Philosophy
2013

University of Strathclyde
Department of Computer and Information Sciences
Declaration

This thesis is the result of the author’s original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

The copyright of this thesis belongs to the author under the terms of the United Kingdom Copyright Acts as qualified by University of Strathclyde Regulation 3.50. Due acknowledgement must always be made of the use of any material contained in, or derived from, this thesis.
Abstract

This dissertation defends the idea of a closed dependent type theory whose inductive types are encoded in a universe. Each inductive definition arises by interpreting its description – itself a first-class citizen in the type theory. Datatype-generic programming thus becomes ordinary programming. This approach is illustrated by several generic programs.

We then introduce an elaboration of inductive definitions down to the universe of datatypes. By elaborating an inductive definition – a syntactic artefact – to its code – its type theoretic denotation – we obtain an internalised account of inductive types inside type theory. This is a small step toward bootstrapping, i.e. implementing the inductive fragment in the type theory itself.

Building upon this universe of datatypes, ornaments let us treat datatypes as the combination of a structure and a logic: they relate datatypes through their common structure. We set out to rationalise this calculus of structures. We study a categorical model of ornaments, based on Cartesian morphisms of containers. We also illustrate the adequacy of our model by recasting the standard idioms into the categorical mould, and by translating the discovered categorical structures into type theoretic artefacts.

Nonetheless, the extreme accuracy of these finely indexed datatypes is a curse for code reuse. Similar functions must be duplicated across similarly structured – but logically incompatible – indexed datatypes. We shall see how code reuse can be achieved by ornamenting functions. We thus capture the relationship between functions such as the addition of natural numbers and the concatenation of lists. We also demonstrate how the implementation of the former informs the implementation of the latter.
Contents

Introduction 9
  1.1. Prospectus ................................................. 11
  1.2. Background .................................................. 17
  1.3. Structure .................................................... 19
  1.4. Notational Conventions ...................................... 20

I. Terms, Types & Programs 23
  2. The Type Theory 25
    2.1. A Minimal Calculus .......................................... 25
    2.2. Enumerations ............................................... 35
    2.3. Meta-Theoretical Properties ................................ 41
  3. A Notation for Programs 47
    3.1. Bidirectional Type Checking .................................. 47
    3.2. Elaborating Programs ........................................ 54
    3.3. Abuse of (Programming) Language ........................... 61

II. The Inductive Fragment of Type Theory 65
  4. A Universe of Inductive Types 67
    4.1. The Universe of Descriptions .............................. 68
    4.2. Initial Algebra Semantics .................................. 74
    4.3. Extending Type Propagation ................................ 84
  5. A Universe of Inductive Families 87
    5.1. The Universe of Indexed Descriptions ..................... 87
    5.2. Initial Algebra Semantics .................................. 91
    5.3. Categorical Semantics of Inductive Families .............. 99
## Contents

### III. Generic Programming 111

6. Bootstrapping Inductive Types 113
   6.1. The Art of Levitation 113
   6.2. A Few Generic Constructions 124
   6.3. Modelling Levitation 135

7. Elaborating Inductive Definitions 145
   7.1. Inductive Types 145
   7.2. Inductive Families 155
   7.3. Reflections on Inductive Types 171

### IV. A Calculus of Structures 181

8. Ornaments 183
   8.1. Universe of Ornaments 185
   8.2. Categorical Semantics of Ornaments 198
   8.3. Tapping into the Categorical Structure 204

9. Functional Ornaments 213
   9.1. From Comparison to Lookup, Manually 214
   9.2. A Universe of Functions and their Ornaments 217
   9.3. Lazy Programmers, Smart Constructors 227

Conclusion 239
   10.1. Further Work 240
   10.2. Implementation Work 241
   10.3. Epilogue 242

A. Overloaded Notations 245

Bibliography 247

Index 257

Acknowledgements 261
1. Introduction

This thesis stands at the crossroad between mathematics and computer science. At the origin of this collision is a fascinating inquiry about the nature of mathematics: what is a mathematical statement? What makes a proof valid or not? By a brutal (and rather limiting) reduction, a mathematical proof could be understood as a formal derivation of a new fact from axioms – facts taken for granted – and some general reasoning principles. For example, from the axioms that “all men are mortal” and that “all Greeks are men”, we conclude that “all Greeks are mortal”. The inference rule that allows us to derive this conclusion from the two premises is called a syllogism, a form of reasoning developed by Aristotle.

However, it is only through the work of Frege, Hilbert, and Russel that this questioning about the rules of mathematics resurfaced in mathematics itself: can we mathematically characterise a mathematical statement and its proof? Whitehead and Russel’s theory of types [Whitehead and Russell, 1910] was an attempt at building such a system. In that framework, our syllogism would be described with three types – Man, Greek, and Mortal – and two assumptions: the major premise asserts that being a man implies being mortal, denoted Man $\rightarrow$ Mortal; the minor premise asserts that being Greek implies being a man, denoted Greek $\rightarrow$ Man. To prove the conclusion that all Greeks are mortal – Greek $\rightarrow$ Mortal – we construct a proof through logical inferences. For example, to prove Greek $\rightarrow$ Mortal, it is sufficient to prove Mortal, provided an abstract inhabitant $x$ of the type Greek, or formally:

$$x: \text{Greek} \vdash \text{Mortal} \quad \vdash \text{Greek} \rightarrow \text{Mortal}$$

To exhibit an inhabitant of Mortal, we can apply our assumption stating that all men are mortals, thus focusing our effort on finding an inhabitant of Man, or formally:

$$x: \text{Greek} \vdash \text{Man} \rightarrow \text{Mortal} \quad x: \text{Greek} \vdash \text{Man} \quad x: \text{Greek} \vdash \text{Mortal}$$

We can apply the same inference rule to deduce an inhabitant of Man from the axiom that all Greeks are mortal and the inhabitant $x$ of the type Greek, or formally:

$$x: \text{Greek} \vdash \text{Greek} \rightarrow \text{Man} \quad x: \text{Greek} \vdash \text{Greek} \quad x: \text{Greek} \vdash \text{Man}$$

We have thus formally derived that Greek $\rightarrow$ Mortal from the assumptions that Man $\rightarrow$
1. Introduction

*Mortal and Greek → Man.*

In effect, type theory is a formalisation of the language of mathematics: it provides a system of types, with which to express mathematical statements, and inference rules, with which to establish the truth of these statements. A proof is then merely the combination of deductive steps, such as abstraction, application, or appeal to axioms. If the daily mathematical practice is far removed from these foundational questions, they provide interesting insights to programming language designers.

Indeed, similar questions arise in the context of programming: how to specify the behavior of our programs? What does it mean for a program to satisfy its specification? In this setting, types are used to classify the inputs and outputs of programs, such as “being a number”, or “being a character”, or more precise properties such as “being a positive number”, or “being a well-formed date”. Types provide an ideal specification language. A program *type checks* if it corresponds to a valid derivation in a particular system of inferences: a type system.

This propositions-as-types and proofs-as-programs interpretation of logic is at the heart of this dissertation. The programming language we shall study is not an ad hoc software construction: it is in fact a formal system in which mathematical facts can be stated and formally proved. From a programmer’s perspective, this also means that precise specifications of programs can be expressed and irrefutably established: *well-typed programs cannot go wrong* [Milner, 1978].

This extreme accuracy of types creates some tension. On one hand, types allow programmers to describe precisely the structure of the objects they manipulate. We thus obtain stronger guarantees about the behavior of programs. However, precise types induce code duplication: a program defined over a certain type cannot be used for another one, even if very similar. To account for these subtle variations, boilerplate code is written that implements the same functionality over and over again. For example, we must implement a function to traverse lists, another one to traverse 2-dimensional arrays, etc.

A programmer’s natural reaction to this issue is to automate these repetitive tasks. This leads to *generic programming*, in which one writes programs that write programs. However, to enable generic programming, we need to provide a representation of the programming language in itself. From a mathematical point of view, this corresponds to a *reflection principle* that enables the type theory to describe (part of) itself. The reflective capability of a logical system is necessarily limited: little trust can be put in a fully-introspective system that boldly asserts “I am not a liar”.

These mathematical considerations have informed and strongly influenced this dissertation. First, it is focused on datatypes, a key structuring medium in modern functional programming languages. Second, we design a type theory that reflects the language of datatypes in itself. Doing so, we obtain a typed calculus supporting generic programming from the ground-up. Elaboration bridges the gap between this calculus and a programming language: our programming artefacts translate to terms in the calculus, thus providing a semantics to the resulting programming language. This thesis is an experiment in turning pervasive mathematical structures into programming artefacts. We now embark on a fascinating apprenticeship to become “mathematical instrument
makers”, wandering in the Glaswegian footsteps of James Watt.

1.1. Prospectus

(1.1) This thesis is an exploration of the predicative fragment of type theory [Martin-Löf, 1984]. Type theory extends the simply-typed lambda calculus with (dependent) quantifiers: \(\Pi\)-types and \(\Sigma\)-types. Read as a proposition, a \(\Pi\)-type \((x : A) \rightarrow B\) is a universal quantifier: it lets us quantify over all the inhabitants – i.e. the terms – of the type \(A\). Operationally, it embeds terms – that is, computations – in the language of types, thus enabling us to write precise types that exactly capture the intended behaviors of our programs.

For example, an ML programmer implementing the \texttt{zip} function\(^1\) has to cope with two logically meaningless cases:

\[
\begin{align*}
\text{zip} & \; (\text{xs} : \text{List } A) \; (\text{ys} : \text{List } B) \quad : \quad \text{List } (A \times B) \\
\text{zip} & \; \text{nil} \; \text{nil} \quad \mapsto \quad \text{nil} \\
\text{zip} & \; (\text{cons } a \; \text{as}) \; (\text{cons } b \; \text{bs}) \quad \mapsto \quad \text{cons } (a, b) \; (\text{zip } \text{as } \text{bs}) \\
\text{zip} & \; \text{nil} \; (\text{cons } b \; \text{bs}) \quad \{ ?? \} \\
\text{zip} & \; (\text{cons } a \; \text{as}) \; \text{nil} \quad \{ ?? \}
\end{align*}
\]

Simple types do not let us express the invariant that the two input lists have the same length. Using dependent types, we can express this invariant at the type level and write the function we intended to:

\[
\begin{align*}
\text{zip} & \; (n : \text{Nat}) \; (\text{xs} : \text{Vec } A \; n) \; (\text{ys} : \text{Vec } B \; n) \quad : \quad \text{Vec } (A \times B) \; n \\
\text{zip} & \; 0 \; \text{nil} \; \text{nil} \quad \mapsto \quad \text{nil} \\
\text{zip} & \; (\text{suc } n) \; (\text{cons } a \; \text{as}) \; (\text{cons } b \; \text{bs}) \quad \mapsto \quad \text{cons } (a, b) \; (\text{zip } \text{as } \text{bs})
\end{align*}
\]

Intuitively, a vector \texttt{Vec }\(A \; n\) is a list of length \(n\). We shall encounter several, equivalent definitions of vectors below.

(1.2) Read as a proposition, a \(\Sigma\)-type \((x : A) \times B\) is an existential quantifier: it asserts the existence of an object of type \(A\) satisfying some property \(B\). This existence is to be understood in an intuitionistic sense: an existential genuinely contains a \textit{witness}, i.e. a term that justifies the validity of a proposition. For instance, we can describe the set of even natural numbers as those natural numbers that satisfy the \texttt{isEven} predicate:

\[
\begin{align*}
\text{EvenNat} & \; \triangleq \; (n : \text{Nat}) \times \text{isEven } n \quad \text{where} \\
\text{isEven} & \; (n : \text{Nat}) \quad : \quad \text{SET} \\
\text{isEven} & \; 0 \quad \mapsto \quad \text{1} \\
\text{isEven} & \; (\text{suc } 0) \quad \mapsto \quad \text{0} \\
\text{isEven} & \; (\text{suc } (\text{suc } n)) \quad \mapsto \quad \text{isEven } n
\end{align*}
\]

Vectors can also be defined through a \(\Sigma\)-type. They correspond to the lists whose

\(\)\(^1\)This introductory section is illustrated by several functions and datatypes. We rely on our eye for functional programming to grasp these definitions. Their syntax is formally presented in Chapter 3.

11
1. Introduction

length matches the index:

\[
\text{Vec} A n \triangleq (a : \text{List} A) \times \text{length} \ as = n
\]

Operationally, \(\Sigma\)-types are a generalisation of pairs: the witness is obtained by taking
the first projection, while the second projection is an inhabitant of a set computed from
that witness. Thanks to \(\Sigma\)-types, we get hold of a rich specification language: we can
write programs over natural numbers or lists, while attaching precise logical invariants
to those objects, such as, respectively, being even or of a certain length. Using these
more precise objects, we are able to write more precise functions. For example, we
specify the list append function as the function that takes two lists of respective lengths
\(m\) and \(n\) and returns a list of length \(m + n\):

\[
(\text{xs} : \text{Vec} A m) \ ++ \ (\text{ys} : \text{Vec} A n) : \ \text{Vec} A (m + n)
\]

\[
\begin{align*}
\text{nil} & \ ++ \ \text{ys} \mapsto \text{ys} \\
(\text{cons} \ a \ \text{xs}) & \ ++ \ \text{ys} \mapsto \text{cons} \ a \ (\text{xs} \ ++ \ \text{ys})
\end{align*}
\]

Because terms are entirely reflected in the types, every function in our calculus must
terminate, for type checking to be decidable. Far from being an inconvenience, this
means that such a programming language is also a formal system suitable for math-
ematical developments. Through this duality, type theory offers a uniform language
to write programs and reason about them. Let us reiterate that, despite the expressive
power of its type system, type checking is decidable: from a programmer’s perspective,
we have not stepped very far away from a typed programming language à la ML [Mil-
nner et al., 1997, Peyton Jones, 2003].

Datatypes are a key convenience of modern functional programming languages. To-
gether with polymorphism, datatypes are the key contribution of ML over LISP. Intu-
itively, a datatype is a set closed under some operations [Aczel, 1977], which is therefore
endowed with a rich structure. The typical ML program starts with a datatype definition
that encodes the structure of the input problem. This carefully crafted type forms
the backbone of the program. Functions operating on this datatype are then defined
by pattern-matching: for each case and potentially iteratively, the programmer describes
the solution to the input problem.

A typical example of inductive type are the Peano numbers

\[
\text{data} \ \text{Nat} : \text{SET} \ \text{where}
\]

\[
\begin{align*}
\text{Nat} \ & \ni \ 0 \\
| \ & \text{suc} \ (n : \text{Nat})
\end{align*}
\]

that, concretely, provide a structure for counting. Reminiscent from the LISP system,
the datatype of lists is defined as the choice between an empty list or the concatenation
of an element of $A$ to a list of $A$:

$$\textbf{data} \ \text{List} \ [A: \text{SET}]: \text{SET} \ \text{where}$$

$\text{List} \ A \ \ni \ \text{nil}$

$\mid \ \text{cons} \ (a: A)(as: \text{List} \ A)$

For faster random access, we might consider a less linear structure than lists. To do so, we modify the $\text{cons}$ operation of lists by grafting another recursive argument. Doing so, we obtain binary trees:

$$\textbf{data} \ \text{Tree} \ [A: \text{SET}]: \text{SET} \ \text{where}$$

$\text{Tree} \ A \ \ni \ \text{leaf}$

$\mid \ \text{node} \ (lb: \text{Tree} \ A)(a: A)(rb: \text{Tree} \ A)$

Whilst these data-structure could be encoded in LISP, ML’s contribution was to integrate them, together with pattern-matching, in the language definition.

((1.5)) In type theory, our types are more precise than in ML. The same goes for datatypes: besides being data-structures, our datatypes also capture the data-logic. We have already seen vectors, i.e. lists of fixed length. Rather than encoding vectors as the pair of a list and a proof, we can define a datatype that integrates the constraint on the length in its definition:

$$\textbf{data} \ \text{Vec} \ [A: \text{SET}](n: \text{Nat}): \text{SET} \ \text{where}$$

$\text{Vec} \ A \ (n = 0) \ \ni \ \text{nil}$

$\text{Vec} \ A \ (n = \text{succ} \ n') \ \ni \ \text{cons} \ (n': \text{Nat})(a: A)(vs: \text{Vec} \ A n')$

For the reader familiar with Agda or Coq, the above definition translates almost literally to the following Agda code:

$$\textbf{data} \ \text{Vec} \ (A : \text{Set}) : \text{Nat} \rightarrow \text{Set} \ \text{where}$$

$\text{nil} : \text{Vec} \ A \ \text{zero}$

$\text{cons} : \forall \ \{n'\} \ (x : A)(xs : \text{Vec} \ A \ n') \rightarrow \text{Vec} \ A \ (\text{succ} \ n')$

Similarly, we can define AVL trees [Adelson-Velskii and Landis, 1962] and enforce the height-balancing invariant by the index:

$$\textbf{data} \ \text{AVL} \ (n: \text{Nat}): \text{SET} \ \text{where}$$

$\text{AVL} \ (n = 0) \ \ni \ \text{leaf}$

$\text{AVL} \ (n = \text{succ} \ n') \ \ni \ \text{nodeE} \ (n': \text{Nat})(lb: \text{AVL} \ n')(rb: \text{AVL} \ n')$

$\mid \ \text{nodeL} \ (n': \text{Nat})(lb: \text{AVL} \ n')(rb: \text{AVL} \ (\text{succ} \ n'))$

$\mid \ \text{nodeR} \ (n': \text{Nat})(lb: \text{AVL} \ (\text{succ} \ n'))(rb: \text{AVL} \ n')$

An inhabitant of $\text{AVL}$ is, by construction, well-balanced: at each node, the absolute difference of height of its subtrees is at most one. Note that, for conciseness, this definition does not actually store any data. As we shall see later, this definition can easily be extended into one containing some information.
1. Introduction

Another example is the datatype representing finite sets of size at most \( n \). This datatype is essentially a number – the cardinality of the finite set – whose index enforces that it is less than \( n \):

\[
\text{data Fin } (n : \text{Nat}) : \text{SET} \ \text{where}
\begin{align*}
\text{Fin } (n = \text{suc } n') & \ni f0 (n' : \text{Nat}) \\
\text{fsuc } (n' : \text{Nat})(k : \text{Fin } n')
\end{align*}
\]

We have introduced datatypes as, essentially, closed sets of operations. In type theory, we restrict ourselves to the least such sets, again to preserve termination of our calculus: this way, we capture (only) objects of finite depth. We model the operations by \textit{signature functors}. By studying the categorical structure of these functors, we obtain an abstract, mathematical toolkit to reason about inductive types [Abbott, 2003, Gambino and Hyland, 2004, Gambino and Kock, 2013]. In effect, we gain access to a wealth of mathematical results. Some of these results lead to interesting programming artefacts, such as the work of McBride [2013] for example.

By adding inductive definitions to type theory, we let the user define datatypes and functions over them. However, as such, inductive types are an external entity: it is not possible to manipulate inductive definitions from inside the language. By giving access to the grammar of inductive definitions in our language [Hinze et al., 2002], we enable \textit{datatype-generic} programming [Gibbons, 2007]: the programmer can define functions over all inductive types at once. For example, it becomes possible to define the catamorphism for all inductive types [Hinze, 2000a]. Similarly, through symbolic manipulations of the grammar of datatypes, we can define the \textit{derivative} of a datatype [Abbott et al., 2005], which is a key component in the definition of the Zipper data-structure [Huet, 1997].

Interestingly, this grammar of inductive definitions is itself nothing but an inductive type, which we can reify internally, \textit{i.e.} define as an inductive type in the object language. Usually, a quoting mechanism lets the user access the code of inductive definitions. This mechanism mediates the external representation of datatypes – as implemented in the programming language – with their internal representation – their reification in the programming language. Most systems [Jansson and Jeuring, 1997, Hinze et al., 2002] use a pre-processor to make these translations transparent to the user. However, it is tempting to simply collapse this isomorphism and \textit{implement} inductive definitions through their internalised encoding: the meta-level inductive definitions are exactly the ones described in the programming language. This removes the need for any mediating mechanism, be it a pre-processor or a quoting function.

Doing so, we bootstrap the inductive fragment of type theory. Many operations on inductive types can now be implemented in type theory itself, instead of requiring meta-theoretical support. Motivating examples include the generation of specialised induction and recursion principles, and a Haskell-like deriving mechanism. Another benefit of this approach is that inductive definitions – the syntactic artefact – are now \textit{translated} to type-theoretic objects. This translation defines the semantics of inductive definitions: at last, we can reason about this syntactic object through its denotation.
Because our datatypes precisely capture the invariants of our programs, programming with dependent types leads to an explosion of special-purpose datatypes. Indeed, our datatypes are a combination of a data-structure and a logic, enforcing the programmer’s dynamic and static invariants by type checking. Consequently, a single data-structure will be duplicated in many forms, each time to capture a different static invariant.

We are thus lead to study the relation between these similarly-structured datatypes. For example, we notice a relation between natural numbers and lists:

\[
\text{data Nat : SET where} \quad \text{data List}[A:SET] : SET \quad \text{where}
\]
\[
\begin{align*}
\text{Nat} & \ni 0 \implies \text{List} A \ni \text{nil} \\
& \mid \text{suc}(n: \text{Nat}) \implies \text{cons}(a:A)(a: \text{List} A)
\end{align*}
\]

Indeed, they share the same linear structure, or put otherwise: for each constructor of the list datatype, we can find a constructor of natural numbers with the same arity:
- the constructor nil is related to the constructor 0 – both of arity 0;
- the constructor cons is related to the constructor suc – both of arity 1.

In this process, we forget the information attached to the cons constructor: an element of A.

**1.10 Remark.** Interestingly, the relation between lists and natural numbers has also been studied by Okasaki [1998] as an instance of a numerical representation. We leave the ornamental interpretation of numerical representations to further work.

Another example is the relation between natural numbers and finite sets:

\[
\text{data Nat : SET where} \quad \text{data Fin}(n: \text{Nat}) : \text{SET where}
\]
\[
\begin{align*}
\text{Nat} & \ni 0 \implies \text{Fin}(n = \text{suc} n') \ni \text{f0}(n': \text{Nat}) \\
& \mid \text{suc}(n: \text{Nat}) \implies \text{fsuc}(n': \text{Nat})(k: \text{Fin} n')
\end{align*}
\]

This time, we notice that the latter datatype has exactly the same operational content as the former: it is nothing but a number. Put otherwise, a function provided a finite set dynamically behaves (i.e. at run-time) exactly as if it were a number. However, it is logically more discriminative: a finite set is a bounded number. Vectors are yet another example of a structure-preserving transformation of natural numbers:

\[
\text{data Nat : SET where} \quad \text{data Vec}[A:SET](n: \text{Nat}) : \text{SET where}
\]
\[
\begin{align*}
\text{Nat} & \ni 0 \implies \text{Vec} A (n = 0) \ni \text{nil} \\
& \mid \text{suc}(n: \text{Nat}) \implies \text{Vec} A (n = \text{suc} n') \ni \text{cons}(n': \text{Nat})(a:A)(a: \text{Vec} A n')
\end{align*}
\]

This transformation introduces new information, as in the transformation from numbers to lists. It is also logically more discriminative, capturing the length of the vector in its type.

Studying these examples, it becomes obvious that a general principle is at play behind these transformations. Our categorical tools provide a perfect language to study this “calculus of structures”, i.e. operations relating datatypes by their structure. This
1. Introduction

dissertation focuses on structure-preserving transformations, as they are key to regain control of the zoo of finely indexed dependent datatypes.

However, we should not forget that datatypes are but a structuring means toward an end: programming. The calculus of structures is a powerful tool to easily create custom datatypes. Yet, for these new datatypes to be useful, we need to be able to adapt our programs to account for these transformations. For example, we noticed that natural numbers evolve into lists. On natural numbers, we implement a comparison function:

\[(m: \text{Nat}) < (n: \text{Nat}) : \text{Bool} \]
\[m < 0 \mapsto \text{false} \]
\[0 < \text{suc } n \mapsto \text{true} \]
\[\text{suc } m < \text{suc } n \mapsto m < n \]

On lists, we are lead to implement a strikingly similar operation:

\[\text{lookup } (m: \text{Nat}) (xs : \text{List } A) : \text{Maybe } A \]
\[\text{lookup } m \text{ nil } \mapsto \text{nothing} \]
\[\text{lookup } 0 (\text{cons } a \text{ xs}) \mapsto \text{just } a \]
\[\text{lookup } (\text{suc } n) (\text{cons } a \text{ xs}) \mapsto \text{lookup } n \text{ xs} \]

Interestingly, Booleans and the option type are also subject to a structure-preserving transformation: they both have no recursive structure. At the type level, it is therefore easy to relate both functions: lookup evolves from \(-<-) by individually transforming the datatypes in a structure-preserving manner.

The relation is even more interesting at the term level, which describes their operational behavior. Both definitions follow the same recursive pattern, as witnessed by their pattern-matching clauses:

\[(m: \text{Nat}) < (n: \text{Nat}) : \text{Bool} \quad \text{lookup } (m: \text{Nat}) (xs : \text{List } A) : \text{Maybe } A \]
\[m < 0 \mapsto \text{false} \]
\[0 < \text{suc } n \mapsto \text{true} \]
\[\text{suc } m < \text{suc } n \mapsto m < n \]

Besides, both definitions return a “related” constructor in the first two cases:

\[(m: \text{Nat}) < (n: \text{Nat}) : \text{Bool} \quad \text{lookup } (m: \text{Nat}) (xs : \text{List } A) : \text{Maybe } A \]
\[m < 0 \mapsto \text{false} \mapsto \text{nothing} \]
\[0 < \text{suc } n \mapsto \text{true} \mapsto \text{just } a \]
\[\text{suc } m < \text{suc } n \mapsto \text{lookup } (\text{suc } n) (\text{cons } a \text{ xs}) \mapsto \text{lookup } n \text{ xs} \]

This suggests that, as we organise datatypes by their structure, we could lift functions along structure-preserving transformations. We are then interested in characterising these liftings, both intensionally and categorically. On one hand, we can derive special-purpose datatypes from general-purpose ones, through structure-preserving transformations. On the other hand, we can semi-automatically lift functions across
these transformations. The combination of these two techniques is an interesting first step toward code reuse in a dependently-typed setting. We benefit from finely indexed datatypes, while avoiding code duplication thanks to our lifting infrastructure.

Finally, this thesis is also a report from the front line. Indeed, it sprung as part of a prototype implementation of the Epigram2 programming language. The Epigram project aimed at designing a type theory supporting generic programming from the ground-up. This work is an exploration of the design space opened up by a peculiar presentation of inductive types. This foray into dependently-typed generic programming leads to novel and interesting programming artefacts. Part of this thesis is an account of these discoveries.

1.2. Background

This thesis stands on the shoulders of giants. The concept of reflection is the cornerstone of Martin-Löf type theory [Martin-Löf, 1984]. This idea appears in the very definition of the $\Pi$ and $\Sigma$ quantifiers. Compared to a strictly stratified logic such as System $F^\omega$, type theory lets us reflect terms at the level of types: the codomain of quantifiers depends on a witness – a term inhabiting the type that is quantified over. Types describe properties of terms and, in turn, terms influence the meaning of types. This idea is pushed even further by the hierarchy of types. The ground level type formers have type $\text{SET}_0$. $\text{SET}_0$ is itself a type former, of type $\text{SET}_1$. Again, in $\text{SET}_1$, we can quantify over objects in $\text{SET}_0$, as well as their inhabitants. And so on, as far as we wish to [Palmgren, 1995]. In effect, the hierarchy of types reflects the logic into itself, level after level. Our work follows this core “design principle” of type theory: we aim at reflecting inductive types in type theory.

The first attempt at internalising inductive definitions comes from Martin-Löf himself, in the form of wellorderings [Martin-Löf, 1984], or W-types for short. The idea is to capture tree-like structures with the pair of a set $\text{Op}$ – the set of operations – and an indexed set $\text{Ar}$ – the set of arities of each constructor. The W-type is the least set closed under $\text{Op}$-nodes

\[
\begin{array}{c}
\text{Op}: \text{SET} \\
\text{Ar}: \text{SET} \\
\text{WOp Ar}: \text{SET}
\end{array}
\]

whose inhabitants are introduced by choosing a kind of node $\text{op}$ and providing its arguments with $\text{xs}$:

\[
\begin{array}{c}
\text{op}: \text{Op} \\
\text{xs}: \text{Ar op} \\
\sup \text{op xs}: \text{W Op Ar}
\end{array}
\]

The elimination principle of W-types corresponds to transfinite induction over such a tree structure.

However, this representation has a strong extensional flavor. For instance, despite
their first-order nature, natural numbers are encoded by:

\[ \text{Nat} \triangleq \mathcal{W} \text{Bool} \lambda \begin{cases} \text{true} \mapsto 0 \\ \text{false} \mapsto 1 \end{cases} \]

Consequently, the constructor of \( \text{Nat} \) relies on functions, respectively from the empty type and from the unit type:

\[ 0 \triangleq \text{sup} \text{true} \text{-elim} \]
\[ \text{suc} (n: \text{Nat}) \triangleq \text{sup} \text{false} \ast \ldots n \]

Extensionally, this is not an issue: we can collapse the trivial function spaces \( X^0 \cong 1 \) and \( X^1 \cong X \). When programming in an intensional type theory, we only work up to definitional equality, \( i.e. \) the unfolding of definitions. While it is possible to identify all functions from the unit type up to their return value, it is not safe to identify all functions from the empty set while retaining a sensible definitional equality in a non-empty context. There is therefore (definitionally) not one “0” but a countable infinitude. In an intensional setting, wellorderings are inadequate [Dybjer, 1997, Goguen and Luo, 1993]. To be fit for programming, we need a more intensional representation of our inductive types. In particular, first-order datatypes ought to have a first-order representation.

Such an intensional approach has been pioneered by Pfeifer and Ruess [1998] and further explored by Benke et al. [2003]. To finely capture the structure of subclasses of inductive definitions, the authors relied on a key type-theoretic concept: universes. A universe is the data of a code and its interpretation. The code is an intensional characterisation of the object of enquiry. In the case of inductive types, it corresponds to a grammar of signatures. The interpretation maps the intensional code to its extension in type theory. In the case of inductive types, we map a code describing a signature functor to an endofunctor on \( \text{S}ET \). Taking the fixpoint of that functor gives an object extensionally equivalent to a W-type, but, ideally, computationally better behaved. This approach for inductive types was further refined by Morris [2007]. It was also used by Dybjer and Setzer [1999] to give a finite axiomatisation of induction-recursion. Our work on descriptions originates from a combination of these two presentations, restricted to purely inductive definitions.

The Curry-Howard correspondence is at the heart of this work: propositions as types, proofs as programs, and conversely. By reflecting terms in types, type theory is built around the Curry-Howard correspondence from the ground-up. The resulting type theory is not only a system in which to formalise constructive mathematics. It can also be seen as a powerful language of specifications [Martin-Löf, 1985, Nordström et al., 1990]. Under this interpretation, cut elimination becomes evaluation. Type theory thus automatically provides a static and dynamic semantics for programs. It is very tempting to develop a full-fledged programming language on top of this minimal calculus.

McBride and McKinna [2004] systematised this approach by presenting the elaboration of a language with pattern-matching to such a minimal type theory. In their system, the user writes programs that are automatically translated to terms in type theory. The
1.3. Structure

static and dynamic semantics of the programming language is thus entirely justified by the low-level type theory. In their original presentation, inductive definitions were excluded from the elaboration process: they were part of the meta-theory, in a syntactic form. Such a presentation precluded the type theory from manipulating inductive definitions. This thesis takes the next step and reflects the grammar of inductive definitions in type theory. We gain the ability to elaborate inductive definitions down to type theory, thus swallowing the entire programming language in type theory. Our objective to support elaboration disciplines the very definition of the type theory: types are not merely policing what terms can be, they also describe their presentation, so as to convey our high-level intents.

(1.20) Having reflected inductive types in type theory, we can write new programs: polytypic programs, or datatype-generic programs [Pfeifer and Ruess, 1998, 1999, Benke et al., 2003, Gibbons, 2007]. Indeed, since the grammar of datatypes is fixed in advance, we can write programs on all datatypes definable in the system by proceeding over the code of inductive types. This kind of program is thus defined “generically” for all datatypes at once. This thesis describes the design of a type theory for generic programming. Apart from LISP and its reflexive derivatives, this is – to our knowledge – the first attempt at designing a typed language that supports generic programming from the ground-up.

1.3. Structure

(1.21) This thesis is organised as follows. In Part I, we walk through the foundations. We study a minimal type theory closed under Π-types, Σ-types, and finite sets. This type theory is strongly inspired by Martin-Löf intensional type theory. Upon this minimal calculus, we grow a programming language. By setting up a bidirectional type checker, we take advantage of the type information to simplify our term language. We also clarify the notations for function definitions and inductive definitions. This provides an account of the syntactic conventions this thesis relies upon.

(1.22) In Part II, we extend our base calculus with inductive types. We do so by giving a presentation of datatypes based on a universe construction. The universe reflects the syntax of inductive definitions within type theory. Being closed, we can write generic programs over the structure of inductive types. We shall study a few examples of generic programs. Our presentation captures strictly-positive families. We prove this claim by showing an equivalence between our universe and containers, a widely studied categorical object. As a result, we obtain a more abstract characterisation of our datatypes and some powerful mathematical tools to reason about them.

(1.23) In Part III, we turn our universe of datatypes into a first-class object: we describe the code of the universe in the universe itself. As a consequence, we can manipulate the code of inductive definitions just like any datatype: generic programming is just programming! We are then able to write programs over the universe of datatypes itself, or even derive new datatypes from others. We study a few examples of such generic operations. We also elaborate inductive definitions down to codes in our universe of types. This is a necessary step to make the universe-based approach practical. Indeed,
1. Introduction

in a programming language, we cannot expect the users to define inductive types by coding them in a universe. We formally specify this translation of inductive definitions and prove that the resulting code type checks, by construction. We also present a few extensions that participate to the bootstrapping of the inductive fragment.

(1.24) In Part IV, we further explore the mathematical structure of datatypes. We start by adapting the notion of ornaments [McBride, 2013] to our universe. Ornaments let us relate datatypes by their common structure. We shed some light on their definition in type theory. We also present a categorical model. This model gives us a another perspective on the type-theoretic definitions. It also points to novel ornamental constructions. We then introduce functional ornaments. Functional ornaments put ornaments at work by relating structure-preserving functions. By identifying, in type theory, the commonality between two functions, we can use one to define the other. We thus present some machinery that lets us semi-automatically lift a function across its ornament.

(1.25) This dissertation makes the following contributions:

- It gives an intensional presentation of inductive types in type theory, which is orthogonal to propositional equality;
- It reflects inductive types within type theory itself, allowing us to conflate generic programming with mere programming;
- It is organised around the elaboration of high-level programming concepts down to low-level type theoretic constructions, which strongly influences our design of the type theory;
- It adapts the concept of ornament to our presentation of inductive types and give a generalisation to functions, demonstrating how code reuse could be achieved in a dependently-typed setting;
- It benefits from and is justified by the categorical study of the structures at play, which lets us grasp the essence of our type-theoretic constructions.

1.4. Notational Conventions

(1.26) This thesis is subject to the following conventions. First of all, it has been produced in technicolor: for an optimal experience, it should therefore be printed in colour. Colours are mainly used to classify the terms of type theory. Their meaning together with our type theoretic notations are described separately in Part I.

(1.27) For ease of reference, every block of text that stands for a single logical unit is numbered. This shall help the reader to refer to a precise point of the thesis. This also gives a Deleuzian dimension to it: through these cross-references, parts of the text echo each other. We have tried to reflect these non-linear narratives – the rhizome [Deleuze and Guattari, 1987] – without overwhelming the reader with cross-references. The electronic version of this document contains hyperlinks that make navigation easier.

(1.28) For the sake of illustration, we sometimes consider incorrect statements. To visually distinguish these statements as erroneous, they are crossed as follows.

(1.29) For pedagogical reasons, we shall develop the type theory incrementally, spreading its definition across the first two parts. It consists of three major components: the syn-
1.4. Notational Conventions

tax, the dynamic and static semantics, and the elaboration of programming constructs into type theory. It is therefore crucial to mark which definitions belong to which components, and to differentiate them from standard definitions within type theory. To this effect, these definitions are enclosed in frames, as follows:

The three components correspond to the following frames:
- **Syntax** frames define the syntax of terms;
- **Meta-theory** frames define the typing judgments and judgmental equality;
- **Elaboration** frames define the elaboration judgments, mapping expressions to terms.

Some frames that one would usually present as part of the meta-theory are here only specified in Part II to be bootstrapped in Part III. Both cases are marked with the frames:
- **Specification** frames specify the type signatures of the objects that shall be implemented within type theory in Part III;
- **Levitation** frames provide the implementations – in type theory – of the objects specified earlier in Part II.

(1.30) We specify the syntax of terms and expressions in Backus-Naur Form (BNF). For example, the syntax of Peano numbers is captured by the grammar:

\[
\langle n \rangle ::= 0 \mid \text{suc} \langle n \rangle
\]

where \( \langle n \rangle \) denotes a non-terminal symbol “\( n \)”, \( \text{suc} \) denotes to a terminal symbol “suc”, and the vertical bar “\( | \)” indicates a choice.

(1.31) **Categorical conventions.** Our categorical notations follow standard practice. For the sake of completeness, here are the notational conventions we have adopted throughout this thesis:

- Category: \( C, D, \ldots \)
- Objects of a category \( C \): \( |C| \)
- Hom-set of a category \( C \): \( C(A, B) \)
- Category of morphisms: \( C \rightarrow \)
- Slice category: \( C/I \)
- Adjunction: \( L \dashv R \), or \( R \vdash L \)

These categorical concepts are folklore, and their definition can be found in Mac Lane’s textbook [Mac Lane, 1998]. To distinguish the categorical concepts from their type-theoretic incarnations, categorical notations are always typeset in mathematical mode, without special colour or font face.

(1.32) **Morphisms in slices.** The morphisms of families of sets \( X, Y : I \rightarrow \text{SET} \) – or equiva-
1. Introduction

lently, morphisms of predicates – are defined pointwise by

\[ X \rightarrow Y \triangleq \forall i : I. X i \rightarrow Y i \]

where the index \( i \) is kept implicit.

Published Material

(1.33) This thesis builds upon four papers published in peer-reviewed conferences:

1. Chapters 4, 5, and 6 give an extended presentation of the ideas developed in an ICFP paper [Chapman et al., 2010] ;
2. Chapter 7 generalises the elaboration of inductive types given in a JFLA paper [Dagand and McBride, 2013b] to inductive families ;
3. Chapter 8 is an expounded version of a LICS paper [Dagand and McBride, 2013a] ;

My contributions to the publications are respectively:

1. The article sprung from a prototype implemented by James Chapman and Peter Morris. I wrote the entire article, which was later edited by Conor McBride, and developed the Agda models from the prototype ;
2. I wrote the entire article and did the research on my own ;
3. I wrote the entire article and did the research on my own ;
4. Following an initial idea from Conor McBride, I developed it and wrote the entire article.

1.34 Remark (Agda model). An Agda model of the various constructions presented in this thesis can be found on the author’s website. To relate our pen and paper presentation with its mechanised model, the modelled sections (e.g. Section 4.1) are followed by a MODEL subheading pointing to a specific file (or directory) of the model. In the electronic version of this document, these entries are hyperlinks.

The research presented in this thesis was supported by a grant of the Engineering and Physical Sciences Research Council (EPSRC, Grant EP/G034699/1).
Part I.

Terms, Types & Programs

In this first part, we introduce a dependently-typed calculus (Chapter 2). Seen as programming language, this calculus provides the computational foundations of this thesis. Seen as a proof system, it is a framework for constructive mathematics.

We then grow a programming language on top of this core calculus (Chapter 3): this affords us a more convenient syntax for describing type-theoretic objects. We take this opportunity to present the syntax used throughout this thesis.
2. The Type Theory

(2.1) We now introduce the type theory this thesis is built upon. We start in Section 2.1 by presenting a basic calculus. This calculus provides dependent types, in the form of Π-types and Σ-types, an infinite hierarchy of types, and some basic set formers.

In Section 2.2, we extend this initial setup with enumerations, i.e. finite sets. Doing so, we follow a long tradition [Martin-Löf 1984, Nordström et al., 1990] that consists in introducing a full-blown type theory by small increments. In particular, this extension strengthens our calculus with the first infinite ordinal. This lets us, in the calculus, represent finite sets, the well-ordered sets that belong to this ordinal class. It also enables us to index into these sets. Put in more operational terms, this extension lets us describe (finite) enumerations of objects and index into such enumerations.

Finally, in Section 2.3, we recall the meta-theoretical properties of such a logical system. We prove a few lemmas about structural rules. We also state the expected proof-theoretic properties of our system, in light of previous models of similar type theories.

2.1. A Minimal Calculus

2.2 Definition (Terms). The syntax of terms is defined by the following grammar:

```
| (t), ⟨T⟩ ::= x          | 0-elim ⟨t⟩
|                  SETₖ   | *
|                  1       |
| (x:⟨T₁⟩) → ⟨T₂⟩     | ⟨t₁⟩ ⟨t₂⟩
| (x:⟨T₁⟩) × ⟨T₂⟩    | ⟨t₁⟩⟨t₂⟩, ⟨t₂⟩:⟨T₂⟩
| (x=⟨t₁⟩, ⟨t₂⟩:⟨T₂⟩) | π₀ ⟨t⟩ | π₁ ⟨t⟩
| ⟨Γ⟩ ::= ε           |
|                  ⟨Γ⟩; x:⟨T⟩ |
```

By convention, the non-terminal ⟨T⟩ ranges over types – i.e. ⟨T⟩ has type SETₖ – while ⟨t⟩ ranges over terms – i.e. ⟨t⟩ has type ⟨T⟩, a type. Because our language is dependently-typed, terms and types belong to the same syntactic category: the non-terminals ⟨t⟩ and ⟨T⟩ are identical and the difference in case is only an aesthetic convenience. The terminal x ranges over variable names. By convention, the literal ℓ denotes a universe level. Π-types are denoted (x:A)→B and Σ-types are denoted (x:A)×B,
2. The Type Theory

where \( x \) is bound in \( B \). Similarly, abstraction is denoted \( \lambda x : T. b \), where \( x \) is bound in \( b \).

The unit type is denoted \( 1 \).

2.3 Remark (Color convention). In a style reminiscent of Epigram [McBride and McKinna, 2004], our syntax of terms is subject to the following colour and font face conventions:

- Bound variables are written in purple, e.g. \( x \);
- The hierarchy of types is written in blue capital letters, e.g. \( \text{SET}_i \);
- Canonical objects, i.e. inhabitants of \( \text{SET} \), are written in blue as well, e.g. \( T \);
- Value constructors are written in red and sans serif, e.g. \( \text{cons} \);
- Computations, and therefore eliminators, are written in green and sans serif, e.g. \( \text{func} \).

2.4 Remark (Abbreviations). When \( x \) is not free in \( B \), we use the following abbreviations

\[
(x : A) \to B \triangleq A \to B \\
(x : A) \times B \triangleq A \times B
\]

hence obtaining the (non-dependent) implication and conjunction from the dependent quantifiers.

2.5 Remark (Telescope notation). We also use a telescope notation for right-nested \( \Pi \)-types and \( \Sigma \)-types:

\[
(x_1 : A_1)(x_2 : A_2) \ldots (x_n : A_n) \to B \triangleq (x_1 : A_1) \to (x_2 : A_2) \to \ldots \to (x_n : A_n) \to B \\
(x_1 : A_1)(x_2 : A_2) \ldots (x_n : A_n) \times B \triangleq (x_1 : A_1) \times (x_2 : A_2) \times \ldots \times (x_n : A_n) \times B
\]

Following standard practice, the arrow and product (be they dependent or not) scope as far right as possible. When mixing \( \Pi \)-types and \( \Sigma \)-types, we expect the arrow to bind more tightly than the product:

\[
A \times B \to C \triangleq (A \times B) \to C \\
(a : A) \times B \to C \triangleq ((a : A) \times B) \to C
\]

2.6 Definition (Substitution). We denote \( t[x/s] \) the substitution of the variable \( x \) for the term \( s \) in \( t \). We shall always consider terms up to \( \alpha \)-conversion: for example, \( \lambda x : T. x \) and \( \lambda y : T. y \) denote the same terms.

(2.7) Our type theory is presented through three mutually defined systems of judgments: context validity, typing, and equality, with the following forms:

\[
\Gamma \vdash \text{VALID} \quad \text{\( \Gamma \) is a valid context, giving types to variables and definitions} \\
\Gamma \vdash (t) : (T) \quad \text{the term} \ t \ \text{has type} \ T \ \text{in context} \ \Gamma \\
\Gamma \vdash (s) \equiv (t) : (T) \quad \text{the terms} \ s \ \text{and} \ t \ \text{are equal at type} \ T \ \text{in context} \ \Gamma
\]
2.1. A Minimal Calculus

### 2.1.1. Context validity

**2.8 Definition** (Context validity). Context validity (Figure 2.1) ensures that variables inhabit well-formed sets. A context is either:

- empty; or
- the extension of a context $\Gamma$ with a variable $x$ of type $S$; or
- the extension of a context $\Gamma$ with a definition $x$ that stands for a term $s$ of type $S$.

**2.9 Remark** (Use of definitions). From a proof-theoretic standpoint, the rule `C TXT-D EF`, which introduces definitions in the context, is unnecessary. Indeed, we can always translate a derivation using a definition to one relying solely on a function application:

$$\Gamma; x \mapsto s : S \vdash t : T \quad \Rightarrow \quad \Gamma \vdash (\lambda x : S. t) s : T$$

However, from a programming standpoint, this rule corresponds to having local definitions, akin to the `let` binding in a simply-typed (but not polymorphic) setting. We shall use this convenience in Chapter 7, as we define type-theoretic objects from high-level inductive definitions.

### 2.1.2. Typing judgments

**2.10 Definition** (Typing judgments). The typing judgments (Figure 2.2) for this dependent calculus are standard. We introduce an infinite hierarchy of types $\text{SET}_\ell$, each of which is closed under the empty type, the unit type, $\Sigma$-types, and $\Pi$-types.

**2.11 Remark** (Cumulativity). Cumulativity of universes (rule `C UML`) lets us shift types to higher universes [Luo, 1994]. This rule captures the inclusion of universes $\text{SET}_0 \subset$
### Meta-theory

\[
\langle \Gamma \rangle \vdash \langle t \rangle : \langle T \rangle
\]

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma; x : S; \Delta \vdash \text{VALID})</td>
<td>(\Gamma; x \mapsto t; S; \Delta \vdash \text{VALID})</td>
</tr>
<tr>
<td>(\Gamma; x : S; \Delta \vdash x : S)</td>
<td>(\Gamma; x \mapsto t; S; \Delta \vdash x : S)</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{VALID})</td>
<td>(\Gamma \vdash S: \text{SET}_\ell)</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{SORT})</td>
<td>(\Gamma \vdash S: \text{SET}_{\ell+1})</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{BOTTOM})</td>
<td>(\Gamma \vdash b : \emptyset)</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{UNIT})</td>
<td>(\Gamma \vdash A: \text{SET})</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{VOID})</td>
<td>(\Gamma \vdash *: \text{I})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma \vdash S; \text{SET}_\ell)</td>
<td>(\Gamma \vdash (x: S) \to T : \text{SET}_\ell)</td>
</tr>
<tr>
<td>(\Gamma; x : S \vdash t : T)</td>
<td>(\Gamma \vdash f : (x : S) \to T)</td>
</tr>
<tr>
<td>(\Gamma \vdash \lambda x : S. t : (x : S) \to T)</td>
<td>(\Gamma \vdash s : S)</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{APP})</td>
<td>(\Gamma \vdash f s : T[s/x])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma \vdash S; \text{SET}_\ell)</td>
<td>(\Gamma \vdash (x : S) \times T : \text{SET}_\ell)</td>
</tr>
<tr>
<td>(\Gamma \vdash s : S)</td>
<td>(\Gamma; x : S \vdash T : \text{SET}_\ell)</td>
</tr>
<tr>
<td>(\Gamma; x : S \vdash t : T)</td>
<td>(\Gamma \vdash t : T[s/x])</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{PAIR})</td>
<td>(\Gamma \vdash (x = s, t : T) : (x : S) \times T)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Gamma \vdash p : (x : S) \times T)</td>
<td>(\Gamma \vdash \pi_0 p : S)</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{FST})</td>
<td>(\Gamma \vdash p : (x : S) \times T)</td>
</tr>
<tr>
<td>(\Gamma \vdash p : (x : S) \times T)</td>
<td>(\Gamma \vdash \pi_1 p : T[\pi_0 p/x])</td>
</tr>
<tr>
<td>(\Gamma \vdash \text{SND})</td>
<td>(\Gamma \vdash \text{CONV})</td>
</tr>
</tbody>
</table>

\(\Gamma \vdash s : S \quad \Gamma \vdash s : T \quad \Gamma \vdash s : \text{SET}_\ell\)

Figure 2.2.: Typing judgments
2.1. A Minimal Calculus

![Figure 2.3: Judgmental equality](image-url)
2. The Type Theory

\( \text{SET}_1 \subset \ldots \) For example, it enables us to write the type-polymorphic identity function:

\[
\text{id} \ (X : \text{SET}) \ (x : X) : X
\]

\[
\text{id} \ (X) \ x \mapsto x
\]

However, it is not syntactic and thus significantly complicates type checking [Harper and Pollack, 1989, Courant, 2002]. As our calculus grows closer to a programming language, we shall come back to cumulativity (§ 3.61). Also, Chapter 6 sits on the edge of a self-referential paradox: in that chapter, we avoid cumulativity altogether and we will be fully explicit about the universes of our constructions.

2.12 Remark (Church-style typing). Our syntax is presented in Church-style: pairs and \( \lambda \)-abstraction are explicitly annotated by types. This information is needed for type inference. When obvious from the context, we informally skip these annotations to improve readability. In Section 3.1, we present an equivalent system in Curry-style: by setting up a bidirectional type checker, we show how local type information can be used to deduce these annotations, without any guesswork or appeal to unification.

2.1.3. Judgmental equality

Unlike traditional presentations [Luo, 1994] based on Pure Type Systems (PTS), we do not define our notion of definititional equality as the transitive, reflexive closure of an (untyped) reduction rule. Instead, we choose to specify definitional equality according to our needs and expectations. Such an axiomatic presentation is called a judgmental equality, or typed conversion [Martin-Löf, 1996]. This presentation should adapt easily to regional variations, as implemented by Coq [The Coq Development Team] or Agda [Norell, 2007].

2.14 Definition (Judgmental equality). The judgmental equality (Figure 2.3) comprises the computational rules (rules BETA, BETA-FST, BETA-SND), the unfolding of definitions (rule DELTA), and it is closed under reflexivity (rule REFL), symmetry (rule SYM), transitivity (rule TRANS), and structural congruences. We do not recall the mundane rules which ensure these structural congruences, typical examples include:

\[
\Gamma \vdash s_1 \equiv s_2 : S \\
\Gamma \vdash t_1 \equiv t_2 : T[x/s_1] \\
\Gamma \vdash (x = s_1, t_1 : S) \equiv (x = s_2, t_2 : S) : (x : S) \times T
\]

\[
\Gamma \vdash f_1 \equiv f_2 : (x : S) \rightarrow T \\
\Gamma \vdash s_1 \equiv s_2 : S \\
\Gamma \vdash f_1 s_1 \equiv f_2 s_2 : T[x/s_1]
\]

(2.15) By specifying equality as a judgment, we leave open the problem of implementing equality, requiring only a congruence including ordinary computation (\( \beta \)-rules). Such an equality can be decided, for example, by testing \( \beta \)-normal forms up to \( \alpha \)-equivalence [Adams, 2006]. Richer equalities, involving various degrees of \( \eta \)-expansion and/or proof-irrelevance, can be supported by our calculus [Coquand, 1996, Abel et al., 2008]. Agda implements such features, Coq currently does not. Decidability of equality induces decidability of type checking, as we shall see in Section 2.3.
2.16 Remark (Polarity of types). In the Martin-Löf tradition [Martin-Löf, 1996], type theory is introduced through a Logical Framework (LF). The logical framework is a meta-language for specifying logical systems, such as type theory. The object language, in our case type theory, is thus specified in an inductive manner: the types of our logic are characterised by their constructors. Thus, quoting Nordström et al. [1990, §7.2], “for most sets, the non-canonical forms and their computation rules are based on the principle of structural induction. This principle says that to prove that a property \( B(a) \) holds for an arbitrary element \( a \) in the set \( A \), prove that the property holds for each of the canonical elements in \( A \).” In modern, proof-theoretic terms, the types thus specified are positive, i.e. characterised by their constructors. Formally, a type is positive if its elimination form is invertible: from the conclusion of the elimination rule, we can derive its premises. Being invertible, the elimination form only plays an administrative rôle: the logical contribution of the type is entirely contained in the (non-invertible) introduction form. For example, the (additive) disjunction is positive:\[1\]

\[
\frac{\Gamma \vdash u : A \to C \quad \Gamma \vdash v : B \to C}{\Gamma \vdash \langle u, v \rangle : A + B \to C}
\]

A type theory presented in the LF-style will thus have a slightly different flavour of \( \Sigma \)-types than ours: its elimination principle will be the \textit{sigma-split} rule that, intuitively, corresponds to pattern-matching a pair, i.e. currying:

\[
\frac{\Gamma \vdash p : (s : S) \times T \quad \Gamma \vdash f : (s : S)(t : T) \to P(s, t)}{\Gamma \vdash \text{sigma-split } f \, p : P \, p}
\]

This corresponds to a positive presentation of \( \Sigma \)-types. Similarly, one could treat \( \Pi \)-types as positive types and eliminate them through the \textit{funsplit} rule [Nordström et al., 1990, §7.2][Garner, 2009].

Perhaps controversially, our presentation (Figure 2.2) is entirely negative: our types are characterised by their observations, \textit{i.e.} elimination forms. Formally, a type is negative if its introduction form is invertible. The exponential is the typical example of a negative type in intuitionistic logic:

\[
\frac{\Gamma ; x : A \vdash b : B}{\Gamma \vdash \lambda x . b : A \to B}
\]

In our type theory, we adopt this traditional presentation of functions for \( \Pi \)-types, eliminating a function by \textit{observing} its behavior on an argument. Similarly, we treat \( \Sigma \)-types negatively, by defining them through their two observations, the first and second projections. The respective introduction forms are invertible.

Negative types can sometimes enjoy a richer definitional equality than positive types:

\[\text{The use of a double line, reminiscent of the categorical notation for adjunctions, to signify the invertibility of a connective is not a coincidence.}\]
2. The Type Theory

with relative ease, the type theory can be extended to support the so-called \( \eta \)-laws for our \( \Pi \)-type, \( \Sigma \)-type, and unit [Abel et al., 2008, 2009]:

\[
\Gamma \vdash S : \text{SET} \quad \Gamma \vdash f : (x : S) \to T \quad \text{ETA-LAM}
\]

\[
\Gamma \vdash S : \text{SET} \quad \Gamma \vdash p : (x : S) \times T \quad \text{ETA-PAIR}
\]

\[
\Gamma \vdash \text{t : } \mathbb{1} \quad \Gamma \vdash \text{t} \equiv \mathbb{1} : \text{ETA-VOID}
\]

The rule ETA-PAIR is also called surjective pairing. Again, these rules are not a requirement for the constructions presented in this thesis. They are merely a convenience, hence our choice for a negative presentation of types. In particular, as we shall see in Part II, our datatype constructors are ultimately nothing but nested pairs terminated by a unit type. Having \( \eta \)-laws for \( \Sigma \)-types and unit allows us to transparently – i.e. without any proving – unpack and repack these tuples.

2.17 Example. In a system supporting ETA-LAM, ETA-PAIR, and ETA-VOID, we have that the function

\[
\text{swap} \ (p : A \times B) : B \times A
\]

\[
\text{swap} \ p \ \mapsto \ (\pi_1 p, \pi_0 p)
\]

is definitionally equal to the identity function when \( A \) and \( B \) are taken to be the unit type, i.e. we have:

\[
\Gamma \vdash \text{swap} \equiv \text{id} : \mathbb{1} \times \mathbb{1} \to \mathbb{1} \times \mathbb{1}
\]

2.1.4. Propositional equality

Propositional equality is this language of evidences in type theory. As such, it can be assumed in context, for hypothetical reasoning, and propositional equalities can be established using type-theoretic reasoning principles, such as induction.

Notions of propositional equality differ between systems. Recent work on homotopy type theory [Hofmann and Streicher, 1994, Awodey and Warren, 2009] seem to suggest that there is more to equality than we might have initially thought [Altenkirch et al., 2007]. From a programmer’s point of view, the dispute revolves around whether Unicity of Identity Proof (UIP, defined below ¶ 2.22) holds in general or only in restricted cases. However, despite these singularities, propositional equality is organised around some core functionalities which are relatively uncontroversial. In order to remain as
2.1. A Minimal Calculus

\[ \Gamma \vdash A : \text{SET}_\ell \]
\[ \Gamma \vdash x : A \quad \Gamma \vdash y : A \]
\[ \Gamma \vdash x = y : \text{SET}_\ell \] \hspace{1cm} \text{PROP-EQUAL}

\[ \Gamma \vdash A : \text{SET}_\ell \]
\[ \Gamma \vdash x : A \]
\[ \Gamma \vdash \text{refl} : x = x \] \hspace{1cm} \text{REFL}

\[ \Gamma \vdash P : A \rightarrow \text{SET}_\ell \]
\[ \Gamma \vdash x : A \]
\[ \Gamma \vdash y : A \]
\[ \Gamma \vdash \text{subst} : x = y \rightarrow P x \rightarrow P y \] \hspace{1cm} \text{SUBST}

\[ \Gamma \vdash x : A \]
\[ \Gamma \vdash \text{absorb} : \text{subst} \text{ refl} x = x \] \hspace{1cm} \text{ABS}

Figure 2.4.: Axiomatic propositional equality

equality-agnostic as possible, we shall again adopt an axiomatic approach, following Coquand [2011]'s recent work. The axioms we choose are quite natural and are satisfied by all flavours of propositional equality to date.

2.19 Definition (Propositional equality). Let \( A : \text{SET}_\ell \) be a type, let \( x : A \) and \( y : A \) be two inhabitants of \( A \).

We extend our type theory with a new set former \( x = y \), which inhabits the type \( \text{SET}_\ell \). The terms \( x \) and \( y \) are said to be propositionally equal if the set \( x = y \) is inhabited.

The equality type is subject to the axioms given in Figure 2.4:

- **Identity**: for any \( x : A \), we have a proof that \( x = x \) (axiom REFL);
- **Substitutivity**: given a proof that \( x \) equals \( y \), we have an operator \( \text{subst} \) that substitutes \( x \) for \( y \) in a proposition (axiom SUBST);
- **Absorption**: substitution by reflexivity is, essentially, an identity (axiom ABS).

2.20 Remark. Because of the Identity axiom, propositional equality is reflexive and thus embeds the judgmental equality. In other words, the following rule is admissible:

\[ \Gamma \vdash a \equiv b : A \]
\[ \Gamma \vdash \text{refl} : a = b \]

The converse is not true, since propositional equalities can be established under inconsistent hypothesis. For example, the following judgment is derivable:

\[ b : 0 \vdash 1 = 0 : \text{SET} \]

We might be reluctant to have the corresponding equation to hold definitionally. Ex-
tensional type theories have this power. We come back to this point in Remark 5.51.

2.21 Remark (Models). Whilst we are careful to loosely specify our propositional equality, it is reassuring to check that our definition admits several, distinct models:

- A Martin-Löf type theory equipped with the identity type [Martin-Löf, 1984] provides an intensional model for which type checking is decidable;
- At the other end of the spectrum, a type theory à la NuPRL [Constable, 1986] offers an extensional model, at the cost of the decidability of definitional equality;
- As an intensional middle ground, an observational type theory [Altenkirch et al., 2007] offers a model that admits function extensionality without sacrificing decidability of type checking.

2.22 Definition (Unicity of Identity Proofs (UIP)). UIP states that there is only one proof of equality, i.e. we add an extra axiom identifying all proofs of equality:

\[
\text{UIP}: (\alpha : a = b)(\beta : a = b) \rightarrow \alpha = \beta
\]

Not every model of type theory satisfies UIP [Hofmann and Streicher, 1994]. Nonetheless, every type enjoying a decidable equality also satisfies UIP, by Hedberg’s theorem [Hedberg, 1998].

2.23 The reduction of pattern-matching to elimination principles [McBride, 1999] relies crucially on UIP. We shall conservatively assume that all the definitions by pattern-matching given in this thesis rely on UIP. Arguably, even a homotopic type theory is likely to offer UIP in a “programming” subuniverse [Licata and Harper, 2011]: our pattern-matching definitions would comfortably live in such a fragment. Aside from the programming fragment, our presentation is not tied to a particular equality. In particular, this system has been modelled in Agda, which features an intensional equality extended with UIP. On the other hand, it has been implemented in an experimental version of Epigram, whose equality has a slightly extensional flavor [Altenkirch et al., 2007]. We expect users of fully extensional systems to also find their way through this presentation.

2.24 Remark (Rewriting vs. computing). Because our presentation is purely axiomatic, it does not compute. The \textsc{abs} rule only tells us that (propositionally) a substitution on a reflexivity proof is an identity. By contrast, systems like Coq and Agda enforce this equality definitionally: substituting through a reflexivity proof computes to an identity. A type theory like Observational Type Theory (OTT) [Altenkirch et al., 2007] offers an even more computational equality, for the user’s convenience. To mimic these computational behaviors, we must explicitly rewrite our types using \textit{subst}. However, these substitutions would needlessly clutter our programs: we shall therefore keep them implicit. A formal treatment of such a translation has been given by Oury [2006], in which he translated the computational equality provided by an extensional type theory to rewrites in an intensional system, the Calculus of Constructions extended with UIP.

2.25 Remark (Design choice (1)). We chose to present an intensional type theory: we have designed decidability into definitional equality. Alternatively, some systems, such as NuPRL [Constable, 1986], lift that restriction and collapse propositional equality into
2.2. Enumerations

the definitional one, at the expense of the decidability of type checking. The means to achieve this is discussed in Remark 5.51. By keeping both notions apart, we can rely on definitional equality to transparently dispose of tedious, decidable equalities, while we have access to propositional equality for establishing non-trivial identities.

2.26 Remark (Design choice (2)). In systems such as Agda and Coq, the notion of inductive family allows the definition of the least reflexive relation, the so-called identity type

```haskell
data _=_ {A : Set} (x : A) : A → Set where
  refl : x = x
```

whose elimination principle justifies the following (computational, but not excitingly so) definition of substitution:

```haskell
subst : {A : Set}{P : A → Set}{x y : A} (x = y) → P x → P y
subst refl p = p
```

Our approach differs in that we strive to maintain equality orthogonal from the notion of datatypes. This leaves us free to merely specify our requirements on equality, without committing ourselves on how propositional equality must be defined. In particular, in our system, we cannot define equality inductively. We shall come back to this point as we present inductive families (Remark 3.48).

(2.27) In this section, we have presented our minimal calculus. We took this opportunity to recall the key concepts of Martin-Löf type theory and familiarise ourselves with the syntax, notation, and colour conventions. From this common ground, we now extend our minimal calculus in various directions. In the next section, we introduce a universe of enumerations. In Part II, we introduce a universe of inductive types followed by a universe of inductive families. At every stage, we extend the term language, the typing and equality judgments.

2.2. Enumerations

(2.28) This section extends our minimal calculus with a notion of enumeration, i.e. finite collection of tags. When coding inductive objects (i.e. datatypes) within type theory, say using W-types [Martin-Löf, 1984], we need to conveniently represent the (finite) choice of operations (i.e. the datatype constructors). While such finite choice can be encoded with the unit type and binary sums, it is a rather primitive interface. In particular, the inability to name the elements of the collection hinders readability.

Informally, we want to be able to declare a collection \{ ‘a’ , ‘b’ , ‘c’ \} of tags ‘a’, ‘b’, and ‘c’. Then, we want to be able to index into such collection by name, for example pointing to the element ‘b’ – second element – of the collection. Finally, we want to be able to eliminate such collection by mapping each tag to a return value, e.g.

\{ ‘a’ ↦ e_a ; ‘b’ ↦ e_b ; ‘c’ ↦ e_c \}
2. The Type Theory

We shall make these notations formal in Section 3.1.

2.29 Remark. Our definition of enumerations as lists of tags is motivated by practical considerations. From a logical standpoint, an enumeration is nothing but an initial sequence of natural numbers: the cardinality of a finite set. However, from a programming perspective, an enumeration also plays a presentation rôle: tags allow us to meaningfully interact with enumerations, by rationalising the typing information obtained through bidirectional type checking (¶3.16, ¶3.18, ¶3.21).

This follows a key design principle of this thesis. To support low-level representations of data, such as enumerations, or inductive types, we extend the type theory with a few fundamental components. However, we engineer these extensions in such a way that they provide enough presentation information in their types and therefore support readability through elaboration.

Enumerations are introduced into type theory through a universe. This universe is specified by the signature:

\[
\text{EnumU} : \text{SET}_\ell \quad \text{EnumT} (E : \text{EnumU}) : \text{SET}_\ell
\]

Inhabitants of \text{EnumU} are finite collections of tags, the so-called enumerations. For an enumeration \(E : \text{EnumU}\), \(\text{EnumT} E\) is the set of (valid) indices into that enumeration.

2.2.1. Tags

2.31 Definition (Tags). To represent named entities, we extend the type theory with tags, i.e. identifiers, quoted to indicate that they are pure data.

\[
\langle t \rangle, \langle T \rangle ::= \ldots \mid \text{UId} \mid \text{ident}
\]

In particular, tags are not variables and are therefore not substitutable.

2.32 Example. A tag is any valid identifier. The following are therefore valid tags: ‘a, ‘true, ‘false, ’0, ’suc, ’Σ.

Tag equality is nominal: two tags are equal if and only if their identifiers are the
2.2. Enumerations

same. We extend judgmental equality accordingly:

\[
\begin{align*}
\Gamma \vdash \text{VALID} \\
\Gamma \vdash s \equiv s : \text{UId}
\end{align*}
\]

2.34 Example. Hence, for instance, we have that \( \not\vdash \text{true} \equiv \text{false} : \text{UId} \).

2.2.2. Enumerations

2.35 Definition (EnumU). To specify an enumeration, all we need is to list its defining tags. This naturally leads to the following formation and introduction rule:

\[
\langle t \rangle, \langle T \rangle ::= \ldots \mid \text{EnumU} \mid \text{nilE} \mid \text{consE} \langle t_1 \rangle \langle t_2 \rangle
\]

\[
\begin{align*}
\Gamma \vdash \text{VALID} \\
\Gamma \vdash \text{EnumU} : \text{SET}_\ell
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{VALID} & \quad \Gamma \vdash t : \text{UId} & \quad \Gamma \vdash E : \text{EnumU} \\
\Gamma \vdash \text{nilE} : \text{EnumU} & & \Gamma \vdash \text{consE} t E : \text{EnumU}
\end{align*}
\]

Intuitively, an \text{EnumU} is a list of tags. By construction, this inductive object describes a finite collection of tags.

2.36 Remark (Specification). Here, we only specify the type of enumerations: in one way or another, we intend to provide a set former \text{EnumU} and the introduction rule \text{nilE} and \text{consE}. However, at this stage, we refrain from extending the type theory with these rules. Instead, we treat this specification as an abstract interface. We rely on this interface to further describe the type theory, but we do not make any assumption on its realisation. We shall fulfil this specification in Chapter 6, where we show that, being an inductive type, \text{EnumU} can be introduced as an inductive type.

2.37 Example (Coding Booleans). To describe the Booleans, we code a collection with two elements:

\[
\{ \text{‘true’ ‘false} \} \triangleq \text{consE ‘true} (\text{consE ‘false} \text{nilE})
\]

(2.38) Again, equality of enumerations is nominal and is also ordered. In particular, it is not solely based on cardinality. Two enumerations are equal if and only if the same
2. The Type Theory

names appear in the same order:

\[
\begin{align*}
\Gamma \vdash \text{VALID} & \\
\Gamma \vdash \text{nilE} \equiv \text{nilE}: \text{EnumU} & \\
\Gamma \vdash \text{consE} t_1 E_1 \equiv \text{consE} t_2 E_2 : \text{EnumU} & \\
\end{align*}
\]

2.39 Example. The following collections are therefore not equal:

\[
\begin{align*}
\not\vdash \{a \text{'b}\} \equiv \{a \text{'b'}c\} : \text{EnumU} & \\
\not\vdash \{a \text{'b}\} \equiv \{c \text{'d}\} : \text{EnumU} & \\
\not\vdash \{a \text{'b}\} \equiv \{b \text{'a}\} : \text{EnumU} & \\
\end{align*}
\]

2.2.3. Indexing into enumerations

2.40 Definition (EnumT). While we declare collections with EnumU, the set of indices into a collection \(E : \text{EnumU}\) – i.e. its elements – is described by EnumT \(E\). We represent the choice of a tag as a numerical index into \(E\):

\[
\langle t \rangle, \langle T \rangle ::= \ldots \mid \text{EnumT} \langle t \rangle \mid 0 \mid 1+ \langle t \rangle
\]

2.41 Remark. The reader familiar with the datatype Fin \(n\) of finite sets of size \(n : \text{Nat}\) will notice a striking similarity. Indeed, inhabitants of Fin \(n\) can be understood as indexing into \(n\), in the same way that inhabitants of EnumT \(E\) index into a collection \(E\). The only difference being that collections have an extra structure – the tags – whereas, for natural numbers, only cardinality matters.
2.2. Enumerations

2.42 Example. The set $\text{Bool} \triangleq \text{EnumT} \{ \text{true, false} \}$ is therefore the set of Booleans:

$$\text{true} \triangleq 0 : \text{Bool}$$
$$\text{false} \triangleq 1+ 0 : \text{Bool}$$

We check that for any $k : \text{EnumT} E$, $\not\vdash 1+ (1+ k) : \text{Bool}$. That is, $\text{Bool}$ has indeed only two inhabitants.

Equality of indices is the expected congruence:

<table>
<thead>
<tr>
<th>Meta-theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \text{VALID}$</td>
</tr>
<tr>
<td>$\Gamma \vdash 0 \equiv 0 : \text{consE} t E$</td>
</tr>
<tr>
<td>$\Gamma \vdash n_1 \equiv n_2 : E$</td>
</tr>
<tr>
<td>$\Gamma \vdash 1+ n_1 \equiv 1+ n_2 : \text{consE} t E$</td>
</tr>
</tbody>
</table>

2.43 Equality of indices is the expected congruence:

2.44 Example. We thus (reassuringly) cannot identify $\text{true}$ and $\text{false}$:

$\not\vdash \text{true} \triangleq 0 \equiv 1+ 0 \triangleq \text{false} : \text{EnumT} \{ \text{true, false} \}$

2.45 Remark (Notation). In practice, we would rather refer to inhabitants of collections by their name. So far, we refer to them by their position in the collection. In Chapter 3, we shall see that the type information can be used to achieve this. In the meantime, we shall informally write, say, $\text{false}$ when referring to an inhabitant of the type $\text{Bool}$, with the confidence that it can be automatically desugared to $1+ 0$. Intuitively, desugaring a name simply consists in finding the position of the corresponding tag in the enumeration.

2.46 Remark. When defining an enumeration, we do not enforce that its defining tags are distinct. Indeed, indexing in an enumeration is purely positional: whether tags are distinct or not makes no difference in the low-level type theory. For instance, it is perfectly valid to define the enumeration:

$$\{ \text{‘this}, \text{‘this} \} \triangleq \text{consE} \text{‘this} (\text{consE} \text{‘this} \text{nilE})$$

It is in fact isomorphic (but not equal, equality being nominal) to the enumeration defining $\text{Bool}$. Indeed, it is inhabited by two terms: $0$ that indexes the first $\text{‘this}$ in the enumeration and $1+ 0$ that indexes the second one. For obvious usability reasons, this kind of definition is best avoided: the high-level notation we present in Chapter 3 will not work on such a definition. This forces the programmer to use the low-level, hence less readable, positional interface.

2.2.4. Eliminating indices

2.47 The function space $(e : \text{EnumT} \{ \text{‘a_1} \ldots \text{‘a_n} \}) \to T$ is an interesting object: it is finite, isomorphic to the (right-nested) tuple $T a_1 \times \ldots \times T a_n \times 1$. Being first-order, we shall
2. The Type Theory

favour this representation. We thus introduce a first-order function space for enumerations. We then eliminate indices into an enumeration by looking up the corresponding value in the tuple, in effect using it as a lookup table.

2.48 Remark (Notation). So far, we have specified the computational behavior of our type theory through judgmental equality. However, this definition style is rather tedious. For compactness and readability, we shall now specify our operators as functional programs – much as we model them in Agda. These meta-definitions will be distinguished from standard definitions by the use of the META-THEORY frame, as for π below.

2.49 Definition (Small Π-type). We first define the small function space using the π operator, which computes the (first-order) tuple isomorphic to \((e : \text{EnumT} E) \rightarrow P e:\)

```
\begin{align*}
\pi (E : \text{EnumU}) \quad (P : \text{EnumT} E \rightarrow \text{SET}_\ell) : \text{SET}_\ell \\
\pi \quad \text{nilE} \quad P & \mapsto \top \\
\pi \quad (\text{consE} t E) \quad P & \mapsto P 0 \times \pi e \lambda e. P (1 + e)
\end{align*}
```

This builds a right-nested Σ-type, packing a witness \(P i\) for each \(i\) in the domain \(E\). These tuples are lookup tables, tabulating dependently-typed functions.

2.50 Remark (Notation). The step case of π exposes our notational convention that binders scope rightwards as far as possible.

2.51 Example. Let \(P : \text{Bool} \rightarrow \text{SET}\). The finite function space from \(b : \text{Bool}\) to \(P b\) unfolds to:

\[
\pi \{ \text{true} \, \text{false} \} P \sim P \text{true} \times P \text{false} \times \top
\]

That is, it is a pair of an inhabitant of \(P \text{true}\) and an inhabitant of \(P \text{false}\).

2.52 Definition (Elimination of enumeration). The elimination principle switch takes us from the small function space \(\pi E P\) – a first-order lookup table – to its functional interpretation \((e : \text{EnumT} E) \rightarrow P e\). In effect, switch simply peels off the lookup table based on the index \(e\):

```
\begin{align*}
\text{switch} \ (ps : \pi E P) \ (e : \text{EnumT} E) : P e \\
\text{switch} \ b \ 0 & \mapsto \pi b \\
\text{switch} \ b \ (1 + e) & \mapsto \text{switch} \ (\pi b) e
\end{align*}
```
2.2.5. Sum type

Using a finite set of cardinality $n$ in the domain of a $\Sigma$-type, we obtain an encoding of $n$-ary sums. In the binary case, using $\text{Bool}$ in the domain, we obtain an encoding of the usual intuitionistic disjunction:

$$A + B \triangleq (b: \text{Bool}) \times \left\{ \begin{array}{l} \text{true} \mapsto A \\ \text{false} \mapsto B \end{array} \right\}$$

We can then derive the usual introduction and elimination forms of disjunction by falling back to the introduction and elimination forms of $\Sigma$-types and enumerations. Alternatively, we could have introduced binary sums as part of our type theory. Either way, the resulting sum type satisfies the following axiomatisation:

\[
\begin{array}{c}
\Gamma \vdash A : \text{SET}_\ell \\
\Gamma \vdash B : \text{SET}_\ell \\
\sum
\end{array}
\]

\[
\begin{array}{c}
\langle (u:A \to C), (v:B \to C) \rangle (s : A + B) : C \\
\langle u, v \rangle (\text{inj}_l a) \mapsto u a \\
\langle u, v \rangle (\text{inj}_r b) \mapsto v b \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash a : A \\
\Gamma \vdash \text{inj}_l a : A + B \quad \text{INJ-LEFT} \\
\Gamma \vdash b : B \\
\Gamma \vdash \text{inj}_r b : A + B \quad \text{INJ-RIGHT}
\end{array}
\]

2.3. Meta-Theoretical Properties

In this section, we give the key meta-theoretic results that underpin our type theory. Following a Swedish tradition [Nordström et al., 1990], we gave a judgmental presentation of conversion. This has the benefit of conceptual simplicity [Goguen, 1994]. It also affords us the freedom of choosing, in an implementation, by which operational means we decide conversion.

However, most of the meta-theoretic results on type theory have been developed on Pure Type Systems (PTS), where conversion is untyped. Using an untyped conversion follows a more syntactic approach initiated by Tait [1967], and pursued by Girard [1972] and Martin-Löf [1984]. It also generalises to inductive types [Luo, 1994, Werner, 1994].

Trying to relate the two, Adams [2006] and Siles [2010] have demonstrated that some PTS can be equivalently presented with a typed or untyped conversion. The PTS they consider are not necessarily normalising. Consequently, they do not support conversion up-to $\eta$-laws, or $\Sigma$-types. Nonetheless, by making a normalisation assumption, Werner [1994] has shown how to regain these $\eta$-laws with an untyped conversion. Conversely, Goguen [1994] has shown that a typed conversion can be faithfully modelled by an underlying, untyped conversion.
2. The Type Theory

Confident that these techniques apply to our type theory, we now state the fundamental properties of this logic. They fit into three groups. First, the structural properties let us juggle with the context. Second, the correctness properties are a first sanity-check that our logic does indeed behave like one. Third, the strong normalisation theorem establishes that our system is a viable logic: it supports cut elimination. This theorem is also crucial in establishing our system as a programming language: this guarantees decidability of type checking. This section is largely inspired by the work of Siles [2010] and Barras [2013].

2.3.1. Structural properties

2.56 Lemma (Weakening [Siles, 2010]). Weakening allows us to introduce more hypotheses in the context, making terms available to larger contexts:

\[
\begin{align*}
1. & \quad \Gamma; \Delta \vdash \text{VALID} \\
& \quad \Gamma \vdash A : \text{SET} \quad \text{then} \quad \Gamma; x : A ; \Delta \vdash \text{VALID} \\
& \quad \Gamma; \Delta \vdash t : T \\
2. & \quad \Gamma \vdash A : \text{SET}, \quad \Gamma; x : A ; \Delta \vdash t : T \\
& \quad \Gamma; \Delta \vdash t \quad \text{then} \quad \Gamma; x : A ; \Delta \vdash \text{VALID} \\
& \quad x \not\in \Gamma; \Delta \\
3. & \quad \Gamma \vdash A : \text{SET} \\
& \quad \Gamma; x : A ; \Delta \vdash s \equiv t : T \\
& \quad x \not\in \Gamma; \Delta
\end{align*}
\]

Proof. By simultaneous induction on derivations of context validity, typing judgment, and the equality judgment. 

2.57 Lemma (Substitution [Siles, 2010]). Substitution lets us cut a judgment making an assumption \( A \) against a proof of \( A \):

\[
\begin{align*}
1. & \quad \Gamma; x : A ; \Delta \vdash \text{VALID} \\
& \quad \Gamma \vdash a : A \\
& \quad \Gamma; \Delta \vdash a / x \quad \text{then} \quad \Gamma; \Delta \vdash a / x \quad \text{VALID} \\
2. & \quad \Gamma \vdash a : A \\
& \quad \Gamma; x : A ; \Delta \vdash t : T \\
& \quad \Gamma; \Delta \vdash a / x \vdash t[a/x] : T[a/x] \\
3. & \quad \Gamma \vdash a : A \\
& \quad \Gamma; x : A ; \Delta \vdash s \equiv t : T \\
& \quad \Gamma; \Delta \vdash s[a/x] \equiv t[a/x] : T[a/x]
\end{align*}
\]

Proof. By simultaneous induction on derivations of context validity, typing judgment, and the equality judgment.

2.58 Definition (Context conversion [Siles, 2010]). Conversion naturally extends from types to contexts: a context is seen as a list of types, themselves subject to conversion. Two contexts are convertible if their types are convertible. Put otherwise, the empty
context are convertible, and, inductively, if two types are convertible in convertible contexts, they form a convertible context:

\[ e \equiv e \vdash \text{VALID} \quad \Gamma \equiv \Delta \vdash \text{VALID} \quad \Gamma \vdash S \equiv T : \text{SET}_\ell \]

2.59 Lemma (Substitution of context). We can naturally transport judgments along convertible contexts:

1. If \( \Gamma \vdash t : T \)
   - If \( \Gamma \equiv \Delta \vdash \text{VALID} \), then \( \Delta \vdash t : T \)
   - \( \Delta \vdash \text{VALID} \)
   - \( \Gamma \vdash s \equiv t : T \)
2. If \( \Gamma \equiv \Delta \vdash \text{VALID} \), then \( \Delta \vdash s \equiv t : T \)
   - \( \Delta \vdash \text{VALID} \)

Proof. By simultaneous induction on the typing judgment and the equality judgment. On the variable case, we do an extra conversion – justified by the context conversion between \( \Gamma \) and \( \Delta \) – to match up the type in the new context \( \Delta \).

\[ \square \]

2.3.2. Correctness properties

(2.60) The following lemmas are sanity-checks that our type theory is not grossly defective. Proof theoretically, they offer little reassurance. However, these lemmas are key to establish more interesting properties.

2.61 Lemma (Validity of context). From a derivation that a term is well-typed in a context \( \Gamma \), we can deduce that its context itself is well-formed: if \( \Gamma \vdash t : T \), then \( \Gamma \vdash \text{VALID} \).

Proof. By induction of the typing judgment.

\[ \square \]

2.62 Lemma (Type correctness and reflexivities [Siles, 2010]). Similarly, from the derivation of a typing or equality judgment, we can deduce that the type is a sort. Also, from the derivation of an equality judgment, we can deduce that the terms are well-typed:

1. If \( \Gamma \vdash t : T \), then either \( T \) is a sort \( \text{SET}_\ell \), or \( \Gamma \vdash T : \text{SET}_\ell \)
2. If \( \Gamma \vdash s \equiv t : T \), then either \( T \) is a sort \( \text{SET}_\ell \), or \( \Gamma \vdash T : \text{SET}_\ell \)
3. If \( \Gamma \vdash s \equiv t : T \), then \( \Gamma \vdash s : T \) and \( \Gamma \vdash t : T \)

Proof. By simultaneous induction on typing judgment and equality judgment.

\[ \square \]

2.3.3. Proof-theoretic properties

(2.63) In a first approximation, a formal system deserves the name of logic if it satisfies a cut-elimination theorem: from a proof making some assumptions, and proofs that these
assumptions hold, we can derive a proof making no assumption at all. Computationally, this corresponds to normalising a function – a proof relying on some assumptions – applied to some terms – the proofs of these assumptions. The result is a self-contained proof, i.e. a term in normal form.

2.64 Conjecture (Strong normalisation [Barras, 2013, Lemma 6.17]). Our type theory admits a strong normalisation model in which reduction is definable and for which every well-typed term reduces to a unique normal form.

(2.65) We refer the reader to Barras [2013] for an elegant (and machine checked!) proof of strong normalisation for the Calculus of Constructions. We are confident that Barras’ proof could be adapted to our setting. Indeed, our type theory is a strict subset of the Calculus of Constructions: we restrict ourselves to predicative sorts equipped with a few standard quantifiers. Thus, if our calculus was not strongly normalising, then a fortiori would the Calculus of Constructions fail to be strongly normalising.

2.66 Corollary (Decidability of conversion). It can be mechanically decided whether or not two terms are definitionally equal: for any $\Gamma, s, t,$ and $T,$ we can decide whether the judgment $\Gamma \vdash s \equiv t : T$ holds or not.

Proof. By application of the strong normalisation theorem: we can simply normalise the two terms, and compare their (unique) normal form.

2.67 Corollary (Decidability of type checking). It can be mechanically decided whether or not a given term is well-typed in a given context: for any $\Gamma, t,$ and $T,$ we can decide whether the judgment $\Gamma \vdash t : T$ holds or not.

Proof. The typing judgment is entirely syntax directed, except for conversion. By Corollary 2.66, we have that convertibility is decidable. Hence, type checking itself is decidable.
2.3. Meta-Theoretical Properties

Conclusion

(2.68) In this introductory chapter, we have set up the type theory in which this thesis is developed. We started with a minimal calculus based on a standard Martin-Löf intuitionistic type theory. Seen as a logical system, this calculus corresponds to a constructive predicate logic. Seen as a programming language, it offers a uniform language of terms to denote programs, their specifications, and their proofs.

(2.69) We have extended this basic calculus with enumerations, which allow us to represent finite sets. From a programming perspective, this lets us conveniently manipulate finite collections through an intelligible interface, namely sets of names. It is also our first example of a universe, even if very rudimentary.

(2.70) We also recalled the meta-theory of such a type theory, which we divide in 3 classes: the structural properties of the typing judgments, some key administrative lemmas, and the strong normalisation theorem. Our objective in doing this is twofold: first, reassuring the reader that our type theory is indeed a proper logical system; and second, reassuring the reader that our type theory is also a programming language for which type checking is decidable. Besides, we have been careful to remain agnostic in the underlying notions of equality, both definitional and propositional. This shall make the following developments more widely applicable.

(2.71) At this stage of development, our calculus is able to represent finite collections of objects using enumerations, record-like structures by combining enumerations with $\Sigma$-types, and functions using $\Pi$-types. Whilst we cannot yet describe datatypes, we can nonetheless describe their signature: all we need is to be able to “tie the knot”, which we shall do in Part II.

Related work

(2.72) Our type theory is a direct descendant of Martin-Löf type theory [Martin-Löf, 1984, Nordström et al., 1990], with influences from Coquand’s Calculus of Constructions [Coquand and Huet, 1988] and Luo’s Unified Type Theory [Luo, 1994]. Technically, our presentation is unsurprisingly conventional. For example, enumerations are but a very weak extension of the type theory, largely subsumed by W-types for examples. We only introduce them for practical reasons.

(2.73) Similarly, the meta-theoretic results are standard. Having adopted a judgmental approach to equality, we rely on the work of Siles [2010] to establish the structural properties and the administrative lemmas that safe-guard our type theory against some operational aberrations. We conjectured strong normalisation, based on an abstract formulation due to Barras [2013], from which we easily derive the decidability of type checking. Although our presentation is strongly based on Siles and Barras presentations, the work of Adams [2006] and Goguen [1994] have also been instrumental in the development of meta-theories for type theories based on judgmental equality.
3. A Notation for Programs

(3.1) From the early days of Martin-Löf type theory, it was clear that dependent types gave more than a framework for constructive mathematics. Martin-Löf [1985] explicitly drew the connection with programming languages. In practice, this analogy has been the driving force behind the Swedish programming languages Alf [Magnusson and Nordström, 1993], Alfa [Hallgren and Ranta, 2000], Agda1 [Coquand and Coquand, 1999], Cayenne [Augustsson, 1998], and more recently Agda [Norell, 2007]. This thesis builds upon the foundation laid by the Epigram school [McBride and McKinna, 2004, Brady et al.]. The key precept of this school is that a dependently-typed programming language should elaborate to a minimal, trusted type theory. Or, seen from the ground-up, McBride and McKinna advocate that we should grow our programming language [Steele, 1999] upon a minimal calculus, such as the one presented in Chapter 2. We recapitulate this approach in this chapter.

Another objective of this chapter is to present the notations used in this thesis. Our programming syntax being close to the functional programming canon, we expect the readers familiar with Haskell or Agda to make their way through this chapter without surprise. However, we shall strive to formally relate the high-level constructs to their representation in the base calculus, thus giving their semantics by translation.

3.1. Bidirectional Type Checking

(3.2) The idea of bidirectional type checking [Pierce and Turner, 2000] is to capture, in the specification of the type checker, the local flow of typing information. On the one hand, we synthesise types from variables and function applications while, on the other hand, we check terms against these synthesised types. By checking the terms against their types, we can use types to structure the term language: for example, to deduce the type annotation in an abstraction, or to deduce the type of the second element of a pair. Doing so, we obtain a calculus in Curry-style, i.e. freed from typing annotation.

3.3 Example (Type synthesis). A typical example of type synthesis is a variable in context. Because the context stores the type of variables, we can deduce its type whenever a variable is used. We thus orient the original VAR rule (on the left) as a synthesis rule (on the right):

\[ \frac{\Gamma; x : S; \Delta \vdash \text{VALID}}{\Gamma; x : S; \Delta \vdash x : S} \quad \Rightarrow \quad \frac{\Gamma; x : S; \Delta \vdash \text{VALID}}{\Gamma; x : S; \Delta \vdash \text{Syn} \quad \Rightarrow \quad \forall x : S} \]
3. A Notation for Programs

3.4 Example (Type checking). The typing rule of λ-abstraction requires an annotation specifying the domain. However, if we orient that rule as a checking rule, we can read the domain – and, hence, deduce the annotation – off the Π-type that the term is checked against. We transform the original inference rule (on the left) into a checking rule (on the right):

\[
\begin{align*}
\Gamma \vdash S : \text{SET} \quad \Gamma;x:S \vdash t : T \\
\Gamma \vdash \lambda x:S.t : (x:S) \to T
\end{align*}
\]

\[
\Rightarrow \quad \frac{\Gamma;x:S \vdash T \ni t \Chk \quad t' \Chk}{\Gamma \vdash (x:S) \to T \ni \lambda x:S.t'}
\]

3.5 Remark (Terminology). To avoid confusion, we settle on the following terminology:

- The judgment \( \langle \Gamma \rangle \vdash \langle t \rangle : \langle T \rangle \) specifies typing (Figure 2.2);
- The judgment \( \langle \Gamma \rangle \vdash \langle T \rangle \ni \langle \text{exprCheck} \langle t \rangle \Chk \) specifies type checking;
- The judgment \( \langle \Gamma \rangle \vdash \langle \text{exprSynth} \langle t \rangle \Syn \in \langle T \rangle \) specifies type synthesis.

During type synthesis, we actually deduce a type \( T \) from an expression \( \text{exprSynth} \). However, unlike Hindley-Milner systems, no constraint solving takes place: we merely take advantage of the local flow of type information.

3.1.1. Type synthesis and type checking

(3.6) Following Harper and Stone [2000], we distinguish two syntactic categories. First, the core type theory defines the language of terms (Definition 2.2). Then, the bidirectional approach lets us extend this core language into a more convenient language of expressions. As we saw in Example 3.3 and Example 3.4, we expect expressions to translate to terms: this is the so-called elaboration of the high-level expressions into well-typed terms in the low-level type theory. This way, we obtain a semantics for our expression language by translation to the minimal calculus.

3.7 Definition (Expression language). We further divide the expression language into two syntactic categories: exprCheck expressions – into which types are pushed – and exprSynth expressions – from which types are extracted (Figure 3.1). In the bidirectional spirit, the exprCheck expressions are subject to type checking. We can afford to enrich it with new term formers: we rely on the typing information to elaborate them. On the other hand, the exprSynth expressions – variables and elimination forms – admit type synthesis. We embed exprSynth into exprCheck, where we shall demand that the synthesised type coincides with the type proposed. The other direction – introducing cuts – takes the form of a type annotation.

3.8 Remark. The non-terminal \( \langle t \rangle \) must be understood as a term (Definition 2.2) whose subterms can be expressions as well as terms. In effect, this includes the language of terms into the language of expressions. The reader will also notice the absence of the elimination principles associated with enumerations (\( \pi \) and \( \text{switch} \)). We expose these operators as (defined) variables. Doing so, we avoid cluttering our definition of the expression language.
3.1. Bidirectional Type Checking

![ELABORATION](image)

Figure 3.1.: Expression language

3.9 Definition (Type synthesis). Type synthesis (Figure 3.2a) is the source of type information. It is directed by the syntax of exprSynth expressions, delivering both a term and its type. Terms and expressions never mix: e.g., for application, we instantiate the domain with the term delivered by checking the argument expression.

3.10 Definition (Type checking). Dually, type checking (Figure 3.2b and Figure 3.2c) receives the flow of types and puts it at work. From an exprCheck expression and a type checked against it, the type checking judgment elaborates a term, using whatever information is available from the type. Canonical set formers are checked. Note that abstraction and pairing are free of annotation, as promised. Most of the propagation rules are unremarkably structural: for the sake of illustration, some of them are given in Figure 3.2c.

3.11 Remark (Normalising substitution). Because type checking is governed by the structure of types, we must ensure that terms are checked against types that are – at the very least – in head-normal form. During type checking and type synthesis, the only instances were this invariant might not be preserved is when we apply a substitution, which may introduce a redex. We rely on substitution when:

- synthesising the application;
- synthesising the second projection;
- checking the second component of a pair.

Hence, instead of applying a mere substitution (Definition 2.6), we rely on a nor-
3. A Notation for Programs

\[ \langle \Gamma \rangle \vdash \langle \text{exprSynth} \rangle \ Syn \quad \langle t \rangle \in \langle T \rangle \]

\[ \Gamma ; x : S ; \Delta \vdash \text{VALID} \quad \Gamma ; \Delta \vdash \text{VALID} \]

\[ \Gamma \vdash \text{SET}_{\ell} \ni t \equiv t' \quad \Gamma \vdash (x : S) \rightarrow T \ni \lambda x . t \equiv \lambda x : S . t' \]

\[ \Gamma \vdash s \equiv s' \quad \Gamma \vdash t \ni \lambda \ni \lambda x : S . t' \]

\[ \Gamma \vdash \text{SET}_{\ell} \ni (s, t) \equiv (x = s', t' : T) \]

(b) Type checking

(c) Type checking (structural rules)
malising substitution, denoted \(\{\sim\}\). A typical example of normalising substitution is hereditary substitution [Watkins et al., 2004], which substitutes and (fully) normalises at the same time.

### 3.1.2. Extending type checking

While the type checker we have specified so far only allows untyped pairs and abstractions, we extend it with some convenient features.

(3.12) \textbf{Elaboration of tuples.} By design of our universe of enumerations, we are often going to build inhabitants of \(\Sigma\)-telescope of the form \((a : A) \times (b : B) \times \ldots \times (z : Z) \times \bot\). To reduce the syntactic burden of these nested pairs, we elaborate a tuple notation inspired by LISP:

\[
\Gamma \vdash A \ni x \text{ Chk} ; x \\Gamma \vdash B \{x' / a\} \ni [xs] \text{ Chk} ; xs \\Gamma \vdash (a : A) \times B \ni [x xs] \text{ Chk} ; (a = x', xs' : B)
\]

### 3.13 Example.

First, we remark that we have introduced an alternative notation for the inhabitant of the unit type: we can either write the term \(*\), or the expression \(\{\}\). Both elaborates to the term \(*\):

\[
\Gamma \vdash 1 \ni \begin{array}{c} * \text{ Chk} \\
\end{array} \quad \Gamma \vdash 1 \ni \begin{array}{c} \{\} \text{ Chk} \\
\end{array}\
\]

### 3.14 Example.

Let \(A : \text{SET}\), \(B : A \rightarrow \text{SET}\), and \(C : (x : A) \rightarrow B x \rightarrow \text{SET}\).

The tuple \([a b c]\) checked against the \(\Sigma\)-telescope \((x : A)(y : B x)(z : C x y) \times \bot\) elaborates to the expected right-nested tuple:

\[
\Gamma \vdash A \ni a \text{ Chk} \quad \Gamma \vdash (y : B t_a) (z : C t_a y) \times \bot \ni [a b c] \text{ Chk} (z = t_{c'} : \bot) \\
\Gamma \vdash (x : A) (y : B x) (z : C x y) \times \bot \ni [a b c] \text{ Chk} (x = t_{a}, y = t_{b}, z = t_{c'} : \bot) : B x)
\]

### 3.15 Example.

In Part II, we shall set-up our datatypes in such a way that datatype constructors are nothing but right-nested tuples starting with a tag, which marks the constructor name. For instance, given \(l : \text{Tree} A\), \(a : A\), and \(r : \text{Tree} A\), the node constructor of the binary tree \text{Tree} A is essentially encoded by the tuple \([\text{node} \ l \ a \ r]\).
3. A Notation for Programs

(3.16) **Elaboration of enumerations.** To define enumerations, we have used the informal notation \{‘a’b’c\}. We support this notation by extending elaboration with the judgments:

\[
\begin{align*}
\Gamma \vdash \text{Enum} & \ni \{\} \rightsimeq \text{nil}E \\
\Gamma \vdash \text{Enum} & \ni \{t \rightsimeq t’ \} \rightsimeq \text{cons}E \ t’E
\end{align*}
\]

**3.17 Example.** Our example thus elaborates to the desired list of tags:

\[
\begin{align*}
\Gamma \vdash \text{UId} & \ni ‘a’ \rightsimeq ‘a’E \\
\Gamma \vdash \text{Enum} & \ni \{’b’ \rightsimeq ‘b’\} \rightsimeq \text{cons}E ‘c’ \ni \text{nil}E
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{Enum} & \ni \{’c’ \rightsimeq \text{cons}E ‘c’ \ni \text{nil}E\} \\
\Gamma \vdash \text{Enum} & \ni \{‘a’ ‘b’ ‘c’\} \rightsimeq \text{cons}E ‘a’ (\text{cons}E ‘b’ (\text{cons}E ‘c’ \ni \text{nil}E))
\end{align*}
\]

(3.18) **Elaboration of enumeration elimination.** Similarly, to eliminate an enumeration, we have used a more natural case analysis syntax:

\[
\{‘a’ \mapsto e_a ; ‘b’ \mapsto e_b ; ‘c’ \mapsto e_c\} : (e : \text{Enum} \{‘a’ ‘b’ ‘c’\}) \rightarrow P \ e
\]

Elaborating this syntax is simply a matter of collecting the cases \(e_i\) in a tuple and elaborating that tuple against the small function space:

\[
\begin{align*}
\Gamma \vdash \text{Enum} & \ni \{’l_0 \ldots ’l_k\} \rightsimeq E \\
\Gamma \vdash (e : \text{Enum} \ {‘l_0 \mapsto e_0 ; \ldots ; ’l_k \mapsto e_k}) \rightsimeq \text{switch} \ ts
\end{align*}
\]

Note that to ensure that the case analysis is meaningful, the collection of matched labels \{‘l_0 \ldots ’l_k\} must elaborate to \(E\), the enumeration we are checked against.

**3.19 Example.** On our running example, for \(E \triangleq \text{cons}E ‘a’ (\text{cons}E ‘b’ (\text{cons}E ‘c’ \ni \text{nil}E))\), this definition unfolds to:

\[
\begin{align*}
\Gamma \vdash \text{Enum} & \ni \{‘a’ ‘b’ ‘c’\} \rightsimeq E \\
\Gamma \vdash (e : \text{Enum} \ {‘a’ \mapsto e_a ; ‘b’ \mapsto e_b ; ‘c’ \mapsto e_c}) \rightsimeq \text{switch} \ ts
\end{align*}
\]
3.1. Bidirectional Type Checking

Note that, formally, the elaborated pairs ought to be annotated. For readability, we have dropped these annotations.

3.20 Remark (Notation). We sometimes vertically break the eliminator of an enumeration. In this case, we do not write the separating semicolon. For example, the above example could (equivalently) be written:

\[
\vdash (e : \text{EnumT} E) \to P e \equiv \left\{ \begin{array}{l}
'a \mapsto e_a \\
'b \mapsto e_b \\
'c \mapsto e_c \\
\end{array} \right. \quad \text{Chk} \quad \text{switch} (t_a, (t_b, (t_c, *)))
\]

(3.21) Elaboration of indices into enumerations. Also, rather than indexing into enumerations through the \text{EnumT} codes, we directly write the corresponding tag. The following rule shows how to elaborate tags to their index:

\[
\begin{array}{c}
\Gamma \vdash \text{consE}'u \ E \ni 'u \text{Chk} \\
\Gamma \vdash E \ni 't \text{Chk} \ni n \\
\Gamma \vdash \text{consE}'u \ E \ni 'u \text{Chk} \\
\Gamma \vdash \text{consE}'u \ E \ni 't \text{Chk} \ni n \\
\end{array}
\]

The idea is to lookup the tag in the enumeration, which is provided by the type checker.

3.22 Example. For example, we can index the tag ‘a – the 0\textsuperscript{th} element – in the enumeration {‘a ‘b ‘c}:

\[
\vdash \text{consE}'a \ (\text{consE}'b \ (\text{consE}'c \text{nilE})) \ni 'a \text{Chk} \ni 0
\]

3.23 Example. Similarly, we can index the tag ‘b – the 1\textsuperscript{st} element – in the enumeration {‘a ‘b ‘c}:

\[
\vdash \text{consE}'b \ (\text{consE}'c \text{nilE}) \ni 'b \text{Chk} \ni 0
\]

\[
\vdash \text{consE}'a \ (\text{consE}'b \ (\text{consE}'c \text{nilE})) \ni 'b \text{Chk} \ni 1+0
\]

3.1.3. Soundness

(3.24) Our elaboration procedure is sound if:

- The term elaborated by checking an expression \text{exprCheck} against a type \( T \) is indeed of type \( T \);
- The pair of a term and its type elaborated from the synthesis of an expression \text{exprSynth} is indeed well-typed.

That is, elaboration of expressions produces well-typed terms in the minimal calculus.

3.25 Theorem (Soundness of bidirectional type checking).

53
3. A Notation for Programs

- If $\Gamma \vdash \text{exprSynth} \; \text{Syn} \; \ell' \in T$, then there exists a level $\ell$ such that $\Gamma \vdash T : \text{SET}_\ell$ and $\Gamma \vdash t' : T$.
- If $\Gamma \vdash T \ni \text{exprCheck} \; \text{Chk} \; t'$, then $\Gamma \vdash t' : T$.

Proof. By mutual induction on the type synthesis and type checking judgments. 

3.26 Remark. This soundness result might seem limited: we are merely enforcing well-typedness. For instance, when elaborating a tag in an enumeration (3.21), we could have (erroneously) defined

$$
\Gamma \vdash \text{consE} \; t \ni \text{tChk} \; 0 \\
\Gamma \vdash \text{consE} \; u \ni \text{tChk} \; 0
$$

without compromising the soundness theorem.

However, we must bear in mind that elaboration defines the semantics of expressions: the above definition is erroneous in as much as it does not convey our intentions. Elaboration is the translation of our intentions into a formal system. We must therefore be cautious that our definitions make sense. In that respect, the soundness theorem is a welcome safeguard against an absolutely non-sensical definition.

3.27 Remark (Functional judgments). We recall that a relation $R \subseteq X \times Y$ is functional if for all $x \in X$, and $y, z \in Y$, if $x R y$ and $x R z$ then $y = z$. Put otherwise, it is the graph of a partial function.

The type synthesis and type checking judgments are functional: they are entirely syntax directed. From our judgmental presentation, we can therefore extract an elaboration algorithm: provided a decision procedure for equality of terms (Corollary 2.66), we have a decision procedure for elaboration.

3.2. Elaborating Programs

3.28 In the previous section, we have enriched our minimal type theory with an expression language, supporting a more convenient, type-directed syntax for terms. In this section, we present some notational convenience for writing programs, i.e. function definitions and datatype definitions. The programming language we describe is similar to those in functional programming canon, such as Haskell or Agda.

3.2.1. Function definitions

3.29 Example (Motivation). Let us consider the implementation of addition by induction over $\text{Nat}$. Recall that the induction principle over natural numbers has type:

$$
\text{Nat-elim} : (P : \text{Nat} \rightarrow \text{SET}) (n : \text{Nat}) \rightarrow P 0 \rightarrow ((m : \text{Nat}) \rightarrow P m \rightarrow P (\text{suc} m)) \rightarrow P n
$$
3.2. Elaborating Programs

After setting up the induction, we have the following code:

\[ + : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \equiv \lambda m \ n. \ \text{Nat-elim} \ (\lambda m'. \ \text{Nat}) \ m \]

\[ \{? : \text{Nat}\} \]

\[ \{? : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}\} \]

3.30 Remark. To represent a development in progress, we use a “hole”:

\[ f (n : \text{Nat}) : \text{Nat} \]

\[ f \ n \mapsto \{?\} \]

Users of Epigram2, Agda, and, lately, Haskell are already familiar with this object, which was initially introduced in the Alf programming language [Magnusson and Nordström, 1993]. Formally, it corresponds to a meta-variable [McBride, 1999, Jojgov, 2004]. We sometimes use an annotated version that displays the type of the hole:

\[ \text{vs} : \text{Vec Nat (fib 14)} \]

\[ \text{vs} \mapsto \{? : \text{Vec Nat 377}\} \]

(3.31) Looking at these proof-goals, it is hard to relate their type with our high-level intent, which is to implement addition and not any function of type \( \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \). This is a fundamental difference between proving and programming: when programming, our intents matter. We therefore extend our type theory with labelled types [McBride and McKinna, 2004]. Their purpose is to label a low-level term with its high-level meaning. We shall employ a labelled type \( \{f \ t_0 \ldots t_k : T\} \) to indicate that a term participates in “defining the function \( f \) applied to a spine of arguments \( t_0 \ldots t_k \)”. In essence, a labelled type is a phantom type [Cheney and Hinze, 2003].

3.32 Definition (Labelled type). We extend the term syntax with labels:

\[
\begin{align*}
\langle t \rangle &::= \ldots \quad | \quad f\langle t\rangle
\end{align*}
\]

Here, \( f \) is any (valid) identifier. A label is simply a name – the name of a function – applied to a spine of arguments.

The formation, introduction, and elimination of programming labels is presented in Figure 3.3. A labelled type \( \{l : T\} \) is just a phantom type around \( T \): we pack a term \( t : T \) with return, and unpack it with call\( [l : T] \).

3.33 Example. Pursuing Example 3.29, we can express our intention of defining addi-
3. A Notation for Programs

\[ \Gamma \vdash T : \text{SET}_\ell \]
\[ \Gamma \vdash \{l : T\} : \text{SET}_\ell \]
\[ \Gamma \vdash t : T \]
\[ \Gamma \vdash \text{return} t : \text{SET}_\ell \]
\[ \Gamma \vdash \text{call} \{l : T\} \left( \text{return} l \right) \rightarrow t \]

\[ \text{M\textsc{eta-Theory}} \]

Figure 3.3.: Programming labels

\[
\begin{align*}
+ : (m : \text{Nat})(n : \text{Nat}) &\rightarrow \{ + m n : \text{Nat} \} \\
&\triangleq \lambda m n. \text{Nat-el} \ 	ext{el} \ 	ext{m} \ 	ext{ Nat-elim} \ (\lambda m'. \{ + m n : \text{Nat} \}) \ m \\
&\{? : \{ + 0 n : \text{Nat} \}\} \\
&\{? : (m : \text{Nat}) \rightarrow \{ + m n : \text{Nat} \} \rightarrow \{ + (\text{suc} m n) : \text{Nat} \}\}
\end{align*}
\]

Thus, from the type of the proof-goals, we know that we have to give the result of \(0 + n\) in the first hole, and, in the second hole, the result of \(\text{suc} m + n\) provided the result of \(m + n\) (induction hypothesis).

In an implementation, the label \(\{ + m n : \text{Nat}\}\) could easily be rendered as \(m + n\). Doing so, we hide the low-level type-theoretic plumbing to the programmer whilst making eliminator-based programming an affordable alternative to pattern-matching.

(3.34) We adopt the following syntax for defining functions:

\[
f (x:X) (y:Y) \ldots : Z
\]

That is, we specify the type of the function \(f\) on the first line, through a telescope of arguments \(x, y, \text{etc.}\) terminated by the result type \(Z\). On the second line, we proceed with the actual definition. When convenient, we use a mix-fix notation, as we do in Example 3.35 below.

3.35 Example. In the programming fragment, the definition of addition is initiated by:

\[
\begin{align*}
(m : \text{Nat}) + (n : \text{Nat}) &\rightarrow \text{Nat} \\
\text{m} &\rightarrow \text{n} \ldots
\end{align*}
\]

(3.36) Following McBride and McKinna [2004], the construction of a function is captured by a specific grammar. At any stage of a development, we can either:

- \text{Return} a value, using the “return” \(\rightarrow\) gadget ; or
- \text{Eliminate} an argument, using the “by” \(\leftarrow\) gadget ; or
- \text{Introduce} an argument under the function’s scrutiny, using the “with” \(\mid\) gadget.
3.2. Elaborating Programs

The “return” gadget corresponds to the usual right-hand side of a pattern definition. The “by” gadget applies an elimination principle. Doing so, with a bit of syntactic sugar, we regain a pattern-matching notation while remaining explicit about the recursive structure of definitions. The “with” gadget allows to (transparently) add the result of a computation to the scrutiny of the function being defined. It merely desugars to an helper function extended with an argument, which is immediately applied to the desired computation.

3.37 Example. To define addition (Example 3.35), we proceed by induction over \( m \) and return \( n \) in the base case, i.e. for \( m = 0 \):

\[
\begin{align*}
(m : \text{Nat}) + (n : \text{Nat}) & : \text{Nat} \\
m + n & \iff \text{Nat-elim} m \\
0 + n & \mapsto n \\
suc m + n & \{	ext{?}\}
\end{align*}
\]

(3.38) When returning a value, recursive calls turn into context lookups: there must be a term in the context whose labelled type matches the recursive call. Put otherwise, there must be an induction hypothesis in the context – introduced by an elimination principle – that justifies the well-foundedness. Otherwise, the definition is rejected.

3.39 Example. We conclude our implementation of addition (Example 3.37) by recursively appealing to addition through the induction hypothesis:

\[
\begin{align*}
(m : \text{Nat}) + (n : \text{Nat}) & : \text{Nat} \\
m + n & \iff \text{Nat-elim} m \\
0 + n & \mapsto n \\
suc m + n & \mapsto \text{suc} (m + n)
\end{align*}
\]

Indeed, the inductive step of the induction principle introduces a hypothesis of type \( \forall \alpha \cdot H + m n : \text{Nat} \) in the context. The recursive call is therefore justified by this witness.

3.40 Remark. In this section, we have presented the syntax of function definitions and informally motivated its semantics. We refer the reader to McBride and McKinna [2004] for the details. In particular, we did not dwell upon the elaboration of these high-level definitions down to low-level terms in the core calculus. We shall assume the existence of such a mechanism.

The idea to take away from this section is the use of labelled type to reflect the high-level intents of the programmer back into the low-level type theory. We shall adapt this technique in Chapter 7 to give a translation semantics to inductive definitions.

3.2.2. Datatype definitions

(3.41) Our notation for declaring datatypes is strongly inspired by Agda’s declarations. In this section, we informally motivate our syntax. In Chapter 7, we shall give a formal semantics to these objects.

3.42 Remark (Notation). Conventionally, for a datatype \( T \), we denote:
3. A Notation for Programs

- \textit{T}-\text{elim} the elimination principle of \( T \);
- \textit{T}-\text{case} the case analysis over \( T \); and
- \textit{T}-\text{rec} the strong induction principle over \( T \).

3.43 Example. For Peano numbers (\textit{i.e.} the natural numbers \( \text{Nat} \)), these induction principles correspond to the propositions:

\begin{align*}
\text{Nat-elim}: (P: \text{Nat} \to \text{SET})(n: \text{Nat}) \to P \ 0 \to ((m: \text{Nat}) \to P \ m \to P \ (\text{suc} \ m)) \to P \ n \\
\text{Nat-case}: (P: \text{Nat} \to \text{SET})(n: \text{Nat}) \to P \ 0 \to ((m: \text{Nat}) \to P \ (\text{suc} \ m)) \to P \ n \\
\text{Nat-rec}: (P: \text{Nat} \to \text{SET})(n: \text{Nat}) \to ((m: \text{Nat}) \to ((k: \text{Nat}) \to k \leq m \to P \ k) \to P \ (\text{suc} \ m)) \to P \ n
\end{align*}

3.44 Our syntax for inductive definitions is intuitive enough to be understood with a few examples. Three key ideas are at play, one of which is novel:

- Constructors are presented as sums of products, à la ML (Example 3.45)
- Indices can be constrained by equality, à la Agda and Coq (Example 3.47)
- Indices can be matched upon (Examples 3.49)

3.45 Example (Sums of products, following the ML tradition). We name the datatype and then comes a choice of constructors. Each constructor is then defined by a \( \Sigma \)-telescope of arguments. For example, \textit{List} is defined by:

\begin{verbatim}
data List [A: SET]: SET where
  List A \ni
  nil
  cons (a: A)(as: List A)
\end{verbatim}

3.46 Remark. We declare datatype parameters in brackets – \textit{e.g.} \([A: \text{SET}]\) in the definition of \textit{List} above (¶ 3.45) – and datatype indices in parentheses – \textit{e.g.} \((n: \text{Nat})\) in the definition of \textit{Fin} below (¶ 3.47). We make equational constraints on the latter only when needed, and explicitly – \textit{e.g.} \((n = \text{suc} \ n')\). The use of explicit equality constraints is reminiscent of the original syntax for GADTs\footnote{GADTs are a subset of inductive families for which the principal-types property is preserved [Cheney and Hinze, 2003, Schrijvers et al., 2009], thus enabling modular (local) type inference.} [Cheney and Hinze, 2003] and aim at conveying the fact that these constraints turn into propositional equalities (Remark 3.48). This difference of treatment between parameters and indices will be justified in Chapter 5, where we extend our type theory with indexed datatypes.

3.47 Example (Indexing, following the Agda convention). Indices can be \textit{constrained} to some particular value. For example, vectors can be defined by constraining the index parameters.
3.2. Elaborating Programs

to be 0 in the nil case and suc\ n’ for some\ n’ : Nat in the cons case:

```
data\ Vec [A : Set](n : Nat) : Set where
  Vec A (n = 0) \∋\ nil
  Vec A (n = suc\ n’) \∋\ cons\ (n’ : Nat)(a : A)(vs : Vec\ A\ n’)
```

In the same vein, finite sets can be defined by constraining the upper-bound n to be always strictly positive, and indexing the argument of \(\text{fsuc}\) by the predecessor:

```
data\ Fin (n : Nat) : Set where
  Fin (n = suc\ n’) \∋\ f0 (n’ : Nat)
  \|\ fsuc (n’ : Nat)(k : Fin\ n’)
```

3.48 Remark (Constraints and equality). In our semantics of inductive definitions, we intend to capture these constraints on indices by falling back to propositional equality (Chapter 7). This is in line with our discipline of remaining equality agnostic. In particular, we do not introduce propositional equality through the inductive fragment.

This is unlike systems such as Coq or Agda, where propositional equality is introduced by the identity type

```
data\ Id [a_1 : A](a_2 : A) : Set where
  _\Id\ a_1\ a_2 (a_2 = a_1) \∋\ refl
```


3.49 Example (Computing over indices). We can also use the crucial property that a datatype definition is, in effect, a function from its indices to a choice of datatype constructors. Our notation should reflect this ability. For instance, inspired by Brady et al. [2003], we give an alternative presentation of vectors that matches on the index to determine the constructor to be presented, hence removing the need for constraints:

```
data\ Vec [A : Set](n : Nat) : Set where
  Vec A n \⇐\ Nat-case\ n
  Vec A 0 \∋\ nil
  Vec A (suc\ m) \∋\ cons\ (a : A)(vs : Vec\ A\ m)
```

In order to be fully explicit about computations, we use here the “by” (\(\Leftarrow\)) gadget, which lets us appeal to any elimination principle. For simplicity, we shall use a pattern-matching style when the recursion pattern is unremarkable. Using pattern-matching, we define the computational counterpart of finite sets by matching on \(n\), offering no
3. A Notation for Programs

constructor in the 0 case, and the two expected constructors in the suc n case:

```
data Fin (n : Nat) : SET where
    Fin 0 ⊇
    Fin (suc n) ⊇ f0
    | fsuc (k : Fin n)
```

(3.50) This last definition style departs radically from the one adopted by Coq, Agda, or generalised algebraic datatypes (GADTs). It is crucial to understand that this is but reflecting the actual semantics of inductive families (as presented in Chapter 5): we can compute over indices, not merely constrain them. With our syntax, we give the user the ability to write these functions: the reader should now understand a datatype definition as a special kind of function definition, taking indices as arguments, potentially computing over them, and eventually emitting a choice of constructors.

3.51 Example. We can sensibly mix these definition styles. An example that benefits from this approach is the presentation of minimal logic – i.e., from the other side of Curry-Howard, the simply-typed lambda calculus [Benton et al., 2012] – given as an inductively-defined inference system. We express the judgment \( \Gamma \vdash T \) through an inductive family indexed by a context \( \Gamma \) of typed variables and a type \( T \):

```
data (\Gamma : Context) \vdash (T : Type) : SET where
    \Gamma \vdash T \ni \text{var} (v : T \in \Gamma)
    | \text{app} (S : Type)(f : \Gamma \vdash S \Rightarrow T)(s : \Gamma \vdash S)
    \Gamma \vdash T \iff \text{Type-case} T
    \Gamma \vdash \text{unit} \ni *
    \Gamma \vdash A \Rightarrow B \ni \text{lam} (b : \Gamma ; A \vdash B)
```

where, for simplicity, we have restricted the language of types to the unit and the exponential:

```
data Type : SET where
    Type \ni \text{unit}
    | (A : Type) \Rightarrow (B : Type)
```

and for which we can define (inductively, in fact) a predicate \( \Gamma \in T \) that indexes a variable of type \( T \) in context \( \Gamma \).

Crucially, the variable and application rules take the index as is, without constraint or computation. The remaining rules depends on the index: if it is an exponential, we give the abstraction rule; if it is the unit type, we give the (only) inhabitant of that type.

3.52 Remark (Order of constructors). By convention, we always first define the constructors that are index-independent, and then define the constructors whose availability depends on the index. This definition style suits our low-level encoding of indexed datatypes (Definition 5.44).
3.3. Abuse of (Programming) Language

Throughout this thesis, we shall write many “programs”. In this chapter, we have grown our programming language from a minimal calculus. We aimed at defining – as rigorously as possible – the meaning of the programs defined in this system. While rigour is primordial, our primary objective is to convey ideas to the reader. We shall therefore indulge in a “fundamental and long established mathematical practice – abuse of language. [Since] without it, we would be stuck in a mass of unnecessary precision and a superfluity of significance.” [Wraith, 1975]

Hence, we shall rely on the reader’s ability to cope with ambiguity, a task that computers are very bad at. Below, we describe the notational conventions we have followed throughout the thesis. We do not expect these notations to be intelligible for a computer: ambiguity is a luxury that only humans can afford.

As in ML, unbound variables in type definitions are implicitly universally quantified. For example, we do not explicitly quantify over n: Nat or A: SET in the definition of vector lookup:

\[
\text{vlookup} \ (m: \text{Fin} \ n) \ (vs: \text{Vec} \ A \ n) : A
\]

While this dramatically reduces the burden of quantifiers, this does not cover all the cases where one would want to declare an argument as implicit. Case in point are higher-order type signatures. To indicate that an argument is implicit, we use the quantifier \(\forall x.\) – or its annotated variant \(\forall x:T.\) – as follows:

\[
\text{example} \ (op: \forall n. \text{Vec} \ A \ n \to \emptyset) \ (xs: \text{Vec} \ A \ k) \ (ys: \text{Vec} \ A \ l) : op \ xs = op \ ys
\]

Following mathematical usage, we shall extensively use mixfix operators, i.e. operators in prefix, infix, postfix, or closed form. We shall not be concerned with the practicality of parsing such definitions: we rely on the reader’s eye for mathematical definitions. For instance, we write:

\[
(m: \text{Nat}) + (n: \text{Nat}) : \text{Nat}
\]

Similarly, we will not concern ourselves with operator precedence: it should always be clear from the context.

Formally, instead of pattern-matching, we use the “by” gadget that lets us apply any elimination principle. However, when the recursive structure of a definition is unimportant and the elimination principle itself is unsurprising, we use a standard pattern-matching notation [Coquand, 1992, Norell, 2007, Sozeau, 2010]. We shall there-
fore define addition (Example 3.39) more concisely by:

\[(m : \text{Nat}) + (n : \text{Nat}) : \text{Nat} \]
\[0 + n \mapsto n \]
\[\text{suc} \, m + n \mapsto \text{suc} \, (m + n) \]

This also applies to computations in datatypes definitions, where Example 3.51 becomes:

\[
data \; (\Gamma : \text{Context}) \vdash (T : \text{Type}) : \text{SET} \text{ where}
\]
\[
\Gamma \vdash T \ni \text{var} \, (v : T \in \Gamma) \]
\[
\Gamma \vdash \text{unit} \ni * \]
\[
\Gamma \vdash A \Rightarrow B \ni \text{lam} \, (b : \Gamma ; A \vdash B) \]

Additionally, we shall sometimes use pattern-matching to define anonymous functions by case analysis, such as for instance:

\[
\lambda \begin{cases} 
\text{true} & \mapsto \text{false} \\
\text{false} & \mapsto \text{true} 
\end{cases} : \text{Bool} \rightarrow \text{Bool} 
\]

With dependent types, a branch of a pattern-matching tree might simply be unreachable. However, whether a branch is reachable or not is undecidable. We shall simply ignore those branches, leaving to the reader the verification that the missing branches are indeed impossible. For example, we can ignore the \text{nil} case when taking the tail of a non-empty vector:

\[
\text{tl} \; (\text{vs} : \text{Vec} \, A \, (\text{suc} \, n)) : \text{Vec} \, A \, n \\
\text{tl} \; (\text{cons} \, a \, \text{vs}) \mapsto \text{vs} 
\]

Because we implicitly quantify over unbound type variables (¶ 3.54), these variables are not explicitly in scope. We shall rely on the convention that, unless the variable name is shadowed, these implicit arguments are automatically in scope of the definition, using the same variable name. For example, in the following definition, \(n\) is universally quantified in the type declaration and is in scope in the definition of \text{lengthVec}:

\[
\text{lengthVec} \, (\text{vs} : \text{Vec} \, A \, n) : \text{Nat} \\
\text{lengthVec} \; \text{vs} \mapsto n 
\]

By design, pattern variable must be \text{linear}, i.e. in a pattern, a variable appears only once. However, with dependent pattern-matching, some terms might be definitionally equal to each others: thus the \text{same} pattern variable may appear several time in a pattern. In Agda, linearity is regained by distinguishing these “forced arguments” with a dot preceding the variable. In our case, we write apparently non-linear patterns: we leave it to the reader to check that appearances are deceptive. For example, we write:

\[
patt \; (m : \text{Nat}) \; (n : \text{Nat}) \; (q : m = n) : \mathbb{1} \\
patt \; m \; m \; \text{refl} \mapsto * 
\]
3.3. Abuse of (Programming) Language

As mentioned in Remark 2.11, being a non-syntactic rule, cumulativity complicates type checking. Several systems exist that engineer decidable type checkers for cumulative theories, including some forms of typical ambiguity [Harper and Pollack, 1989, Luo, 1994, Courant, 2002]. Rather than burdening our presentation with such technicalities, we adopt a lightweight (but informal) convention. When giving a type signature, we shall assign the lowest possible levels to the sets involved but transparently shift its type level if needs arise. For example, we would define the powerset as

\[
\text{Pow} \ (X : \text{SET}) : \text{SET}_1 \\
\text{Pow} \ X \mapsto X \rightarrow \text{SET}
\]

and we would then be able to apply the powerset construction to an hypothetical \( T : \text{SET}_2 \), which shifts the resulting type uniformly:

\[
\text{Pow} \ T : \text{SET}_3
\]

Using this informal convention, we thus write a single definition while supporting its (uniform) use at higher universe levels. We leave it to the reader to check that the definitions obtained following this convention are well stratified.
3. A Notation for Programs

Conclusion

In this chapter, we have grown our earlier calculus closer to an actual programming language. The first step was to declutter the term language, using bidirectional type checking. Doing so, we benefit from a Curry-style type system, in which terms are freed from type annotations. Besides, by putting the flow of typing information at work, we can use types to guide the presentation of terms: we obtain a slightly more high-level, and yet unambiguous (even for a type checker) expression language. This expression language subsumes the minimal calculus of the first Chapter, and forms the basis of the programming language used throughout this thesis.

We have introduced this larger programming language by way of examples. Two key components are at play: the definition of functions, and the definition of datatypes. In both cases, we expect an actual system to translate these high-level definitions to elaborate to the our earlier minimal calculus. The former has been treated by McBride and McKinna [2004], while the latter is the topic of Chapter 7. This section was also an opportunity to introduce the syntax used in this thesis.

Our extensive use of programs to describe mathematical objects creates a tension: our notation must be sufficiently formal to be meaningful, yet not too cluttered with syntactic details so as to be intelligible by the reader. We therefore took the liberty of introducing ambiguity in our notations, thus making the type-theoretic definitions less bureaucratic. We have listed our abuses and, again, illustrated these with examples.

Related work

Using a bidirectional approach in a dependently-typed setting is not a novel idea: for instance, it is the basis of Epigram’s type checker [Chapman et al., 2005] and Matita’s refinement system [Asperti et al., 2012]. The rationale behind a particular orientation of rules remain mostly unexplained. Bidirectional type checking originated from an experimental analysis of Standard ML code. This might explain the perhaps dogmatic tone of this chapter: we observe the rule of thumb that canonical objects are checked against their types, while types are synthesised from elimination forms. Besides, cuts must be annotated by their type, so as to participate to type synthesis. Nonetheless, Puech [2013] recently gave a systematic construction of a bidirectional type theory from a presentation in natural deduction. The future will tell whether his findings adapt to our richer setting, and whether they provide a satisfactory explanation as to the specific orientation of rules.
Part II.

The Inductive Fragment
of Type Theory

This second part is organised as follows. In Chapter 4, we internalise a presentation of inductive types in type theory. Relying on our intuition of ML datatypes, we develop a general methodology for extending a type theory with inductive types and their elimination principle.

This lays the foundation for Chapter 5 where we extend our type theory with inductive families. Inductive families subsume inductive types by supporting a notion of indexing. Indices let us enforce logical invariants on top of our data-structures. We also recall a categorical semantics of inductive families based on the theory of containers. Doing so, we establish a back-and-forth between type-theoretic objects and categorical concepts.
4. A Universe of Inductive Types

(4.1) In this chapter, we extend our type theory to support inductive types as we know them from ML. In particular, we are not concerned with indexing, i.e. inductive families, at this stage. This is for purely pedagogical reasons. First, this lets us build our presentation on our intuition of ML datatypes. Second, a simpler framework makes for a better experimental platform, in which we can easily try new ideas. Finally, as we shall see in the next chapter, our constructions on non-indexed structures lift almost immediately to indexed ones, at the cost of some indexing noise. This chapter is thus a first step toward understanding inductive families and constructions on them.

(4.2) In dependently-typed languages, Σ-types can be interpreted as two different generalisations of propositional connectives. This duality is reflected in the notation we can find in the literature. On the one hand, the notation \( \sum_{x:A} B x \) highlights the view that Σ-types are “dependent sums”, generalising sums over arbitrary arities, where simply-typed languages have only finite sums. On the other hand, our choice, \((x : A) \times B x\), puts the emphasis on Σ-types as generalised products, with the type of the second component depending on the value of the first.

(4.3) In the ML family, datatypes are presented as a sum-of-products. A datatype is defined by a finite sum of constructors, each carrying a product of arguments. Formally, this structure is reflected by their signature functors, such as

\[
\begin{align*}
\text{Nat} X &= 1 + X \\
\text{List}_A X &= 1 + A \times X \\
\text{Tree}_A X &= 1 + A \times X \times X \\
\text{Ordinal} X &= 1 + X + X^{11}
\end{align*}
\]

where the sums encode the choice of constructors and the products encode the constructors’ arguments. We have used an exponential notation \( X^{11} \) to denote the function space \( \mathbb{N} \to X \).

(4.4) With dependent types, the notion of sum-of-products translates to sigmas-of-sigmas. The sum of constructors is but a dependent sum over a finite enumeration of constructors. The product of arguments is then a telescope of dependent products. Consequently, we obtain a simple model of inductive types within type theory: we give their semantics by means of endofunctor on \textbf{SET}. For instance, the earlier examples are mod-
4. A Universe of Inductive Types

elled by the following functions on $\text{SET}$

$$\text{Nat } X \triangleq \sum_{\text{Bool}} \left\{ \begin{array}{c}
\text{'true } \mapsto \mathbb{1} \\
\text{'false } \mapsto X
\end{array} \right\}$$

$$\text{List}_A X \triangleq \sum_{\text{Bool}} \left\{ \begin{array}{c}
\text{'true } \mapsto \mathbb{1} \\
\text{'false } \mapsto (a:A) \times X
\end{array} \right\}$$

$$\text{Tree}_A X \triangleq \sum_{\text{Bool}} \left\{ \begin{array}{c}
\text{'true } \mapsto \mathbb{1} \\
\text{'false } \mapsto (a:A) \times (x:X) \times X
\end{array} \right\}$$

$$\text{Ordinal } X \triangleq \sum_{\text{EnumT}} \left\{ \begin{array}{c}
\text{'0 } \mapsto \mathbb{1} \\
\text{'cons } \mapsto X \\
\text{'lim } \mapsto \mathbb{N} \rightarrow X
\end{array} \right\}$$

where we have artificially used the two notations for $\Sigma$-types to highlight the distinct role of the (unique!) $\Sigma$-type.

4.1. The Universe of Descriptions

MODEL: Chapter4.Desc\(^{1}\)

Sigmas-of-sigmas provide a type-theoretic semantics for inductive types. This characterisation by signature functors is extensional, akin to W-types in extensional type theory. However, in our (intensional) type theory, we must adopt a more intensional approach for our definition to be practical. We obtain such an intensional definition through a universe construction [Martin-Löf, 1984]. A universe is composed of a code – an intensional object – and an interpretation function that computes the extension of the codes. Universes are ubiquitous in dependently-typed programming [Benke et al., 2003, Oury and Swierstra, 2008].

In this thesis, we inductively capture the grammar of (strictly-positive) signature functors. The resulting codes are then interpreted to the desired extension: endofunctors on $\text{SET}$. Consequently, we extend our type theory with a universe of descriptions. The codes describe the signature functors we are interested in. Such an object is purely syntactic, and thus intensional. We obtain the extension of a description, i.e. its functorial semantics, by interpretation.

4.6 Definition (Universe of descriptions). The universe of descriptions is defined in Figure 4.1. The meaning of codes is given by their interpretations on $\text{SET}$:

- $\Sigma$ codes $\Sigma$-types – to build sigmas-of-sigmas;
- $\times$ codes products – the first-order, finitary counterpart of $\Pi$;
- $\Pi$ codes $\Pi$-types – to capture higher-order arguments;
- $1$ codes the unit type – to terminate codes;

\(^{1}\)Following Remark 1.34, this marker indicates that the content of this section has been modelled in the corresponding Agda file, which is available on the author’s website.
4.1. The Universe of Descriptions

- \textbf{var} codes the identity functor – to introduce the recursive argument (X in our examples).

We call an object \(D : \text{Desc}\) a description.

\textbf{4.7 Remark} (First-order representations). Up to isomorphism, \(\times\) is subsumed by \(\Pi\), \textit{i.e.} we extensionally have that:

\[
\begin{align*}
[A \times B] X & \equiv [A] X \times [B] X \\
& \cong (b: \text{Bool}) \mapsto \begin{cases} 
\text{true} & \mapsto [A] X \\
\text{false} & \mapsto [B] X 
\end{cases} b \\
& \cong [\Pi \text{Bool}] \begin{cases} 
\text{true} & \mapsto A \\
\text{false} & \mapsto B 
\end{cases} X
\end{align*}
\]

However, this apparent duplication has some value. Unlike its counterpart, \(\times\) is first-order. First-order representations are finitary by construction, and thus admit a richer decidable equality than higher-order representations may in general possess. For example, extensionally, there is a unique function in \(\emptyset \to \text{Nat}\). However, intensionally, there is a countable infinitude of such functions that cannot be identified definitionally. In this case, we can expect a stronger definition equality for pairs (\textit{i.e.} surjective pairing) than we can hope for functions (Remark 2.16).

\textbf{4.8 Remark}. Our original presentation of descriptions \cite{Chapman et al., 2010} was based on a stripped-down version of induction-recursion \cite{Dybjer and Setzer, 1999}, where the
4. A Universe of Inductive Types

code $\delta$ was restricted to a constant set-domain $H$. The universe was the following:

\[
\begin{align*}
\Gamma &\vdash \text{VALID} \\
\Gamma &\vdash \iota : \text{Desc} \\
\Gamma &\vdash S : \text{SET} \\
\Gamma &\vdash T : S \to \text{Desc} \\
\Gamma &\vdash \sigma ST : \text{Desc} \\
\Gamma &\vdash H : \text{SET} \\
\Gamma &\vdash D : \text{Desc} \\
\Gamma &\vdash \delta H D : \text{Desc}
\end{align*}
\]

While in essence equivalent, the presentation we adopt in this thesis is more algebraic. It reflects more faithfully the underlying categorical structure of signature functors. Consequently, developing the algebra of descriptions within type theory is more natural. In Section 5.3.2, we shall present a few such constructions, namely composition of functors, but also the reindexing functor and its adjoints (these operations are defined in Remark 5.66).

4.9 Definition (Strictly-positive functor). A functor $F$ is said to be strictly positive if the variable $X$ does not appear in the domain of an exponential in $F X$.

4.10 Lemma. Descriptions define strictly-positive functors.

Proof. By induction on the codes and analysis of their interpretation. Intuitively, this is straightforward: the code for $\Pi$ – which is the only code interpreting to an arrow – takes a constant $\text{SET}$ in its domain. The resulting code has therefore only constant sets in the domains of the exponentials.

4.11 Remark. If we were to take $S : \text{Desc}$ in the domain of the $\Pi$ code, our functors would therefore fail to be strictly-positive by construction. We would not be able to take their initial algebras. With some (non-trivial) massaging, we could take the domain of the $\Sigma$ code to be a $S : \text{Desc}$: the resulting coding system would be at least as expressive as induction-recursion. We do not consider this generalisation here.

4.12 Remark (Limitations). In our definition of (indexed) descriptions, simplicity has primed over exhaustiveness. Indeed, we have chosen to focus our efforts on building infrastructure – such as generic programming (Part III), and ornaments (Part IV) – on top of a universe of datatypes. To this end, the simpler the universe, the better.

While any strictly-positive family is expressible through encodings, our universe does not explicitly support:

- Nested fixpoints, unlike the universe given by Morris et al. [2009];
- Parametric arguments, unlike the presentation given by Jansson and Jeuring [1997];
- Mutually-inductive definitions (Example 5.34); and
- Nested types [Bird and Meertens, 1998], unlike the presentation given by Matthes [2009].

70
4.1. The Universe of Descriptions

(4.13) Being monotone, strictly-positive functors admit a least fixpoint [Smyth and Plotkin, 1977]. Categorically, this translates into the existence of an initial algebra. We shall develop the initial algebra semantics of descriptions in Section 4.2.

4.14 Remark. As for enumerations (Remark 2.36), we only specify the code of descriptions. This specification must be read as a type signature. A viable approach would be to simply extend the theory with constants for these constructors and an operator to reason by induction over the codes of Desc. In Chapter 6, we present a more economical alternative: we implement Desc by bootstrapping it within the universe of datatypes itself.

(4.15) The benefits of a universe-based approach are manifold. First, we obtain a concrete – indeed, syntactic – representation of strictly-positive functors within type theory. Accordingly, our type theory can be extended with the associated functorial structure – such as its action on objects \([ \_ ] \) (Figure 4.1b) and morphisms \([ \_ ] \rightarrow\):

| \[ (D:\text{Desc}) \rightarrow (f:X \rightarrow Y) (xs:[][D] X) : [D] Y \] |
| \[ 1 \rightarrow \] |
| \[ f \rightarrow * \] |
| \[ \text{var} \rightarrow \] |
| \[ f \rightarrow x \] |
| \[ \rightarrow f x \] |
| \[ [A \times B] \rightarrow \] |
| \[ f \rightarrow (a,b) \rightarrow ([A] \rightarrow f a, [B] \rightarrow f b) \] |
| \[ \Sigma S T \rightarrow \] |
| \[ f \rightarrow (s,t) \rightarrow (s,[T s] \rightarrow f t) \] |
| \[ \Pi S T \rightarrow \] |
| \[ f \rightarrow t \rightarrow \lambda s. [T s] \rightarrow f (t s) \] |

In Section 4.2, we further extend our type theory with fixpoints and the associated initial algebra semantics.

Second, having internalised the grammar of datatypes, we can – in type theory – manipulate datatype definitions. We are able to inspect an earlier definition and derive a new one from it. Also, we are able to define programs over all (or a subclass of) datatypes in the system: since the description codes are type-theoretic objects, we ought to be able to reason and compute over them. Such definitions over the grammar of datatypes are an example of datatype-generic programming [Gibbons, 2007].

4.1.1. Examples

MODEL: Chapter4.Desc.Examples, Chapter4.Desc.Tagged

(4.16) In this section, we build our intuition for the universe of descriptions. To do so, we present a few examples of signature functors from the ML world and describe them in our universe.

4.17 Remark. We shall work in the high-level expression language of Section 3.1, exploiting type checking to declutter our term language. This convenience affords us a natural syntax for enumerations: we use enumerations to list datatype constructors and to map each constructor to its code.
4. A Universe of Inductive Types

4.18 Example (Description: natural numbers). We begin with the natural numbers, of signature functor $X \mapsto 1 + X$:

$$\text{NatD} : \text{Desc}$$

$$\text{NatD} \mapsto \Sigma \text{EnumT} \begin{cases}
'0' \mapsto 1 \\
'suc' \mapsto \text{var} \times 1
\end{cases}$$

This construction reads as follow. First, we use $\Sigma$ to give a choice between the '0 and 'suc operations. What follows depends on this choice, so we then match each case and give the corresponding code. In the '0 case, we simply terminate with the unit type. In the 'suc case, we attach one recursive argument and terminate the description. Translating the $\Sigma$-type to a binary sum, we have effectively described the desired functor:

$$\llbracket \text{NatD} \rrbracket X \cong 1 + X$$

4.19 Remark. The code describing the successor is terminated by a multiplication by unit, coded by $- \times 1$. Extensionally, this multiplication is superfluous. However, we will be particularly careful to always terminate the constructor’s code with this unit type. As hinted at in Example 3.15, this definition style allows us to rely on the tuple notation for right-nested tuples, which are, by design, terminated by unit ($\¶ 3.12$).

We shall come back to this point more concretely when define the datatypes’ constructors (Remark 4.42) and when we extend elaboration to support constructor expressions (Remark 4.69).

4.20 Example (Description: lists). The signature functor for lists is $X \mapsto 1 + A \times X$, thus its description is but slightly different from NatD:

$$\text{ListD} (A : \text{SET}) : \text{Desc}$$

$$\text{ListD} A \mapsto \Sigma \text{EnumT} \begin{cases}
'\text{nil}' \mapsto 1 \\
'\text{cons}' \mapsto \Sigma A \lambda \cdot \text{var} \times 1
\end{cases}$$

The suc constructor becomes cons, taking an $A$ followed by a recursive argument. By unfolding the definition of the interpretation $\llbracket \cdot \rrbracket$ on ListD $A$, we verify that this code indeed describes the intended functor:

$$\llbracket \text{ListD} A \rrbracket X \cong 1 + A \times X$$

4.21 Example (Description: trees). Of course, we are not limited to one recursive argument. For instance node-labelled binary trees are captured by the signature $X \mapsto 1 + X \times A \times X$. This is described by:

$$\text{TreeD} (A : \text{SET}) : \text{Desc}$$

$$\text{TreeD} A \mapsto \Sigma \text{EnumT} \begin{cases}
'\text{leaf}' \mapsto 1 \\
'\text{node}' \mapsto \text{var} \times \Sigma A \lambda \cdot \text{var} \times 1
\end{cases}$$

Again, we are only one evolutionary step away from ListD. However, instead of a
4.1. The Universe of Descriptions

single call to the induction code, we add another one. The interpretation of this code is isomorphic to the expected functor:

\[ \text{TreeD} A \] X \cong 1 + X \times A \times X \]

4.22 Example (Description: option type). Of course, descriptions also capture datatypes with no recursive arguments, such as the option type. The option type can be represented by the somewhat trivial signature functor \( X \mapsto A + 1 \). In our system, this translates to:

\[
\begin{align*}
\text{MaybeD} & (A : \text{SET}) : \text{Desc} \\
\text{MaybeD} A & \mapsto \Sigma \text{EnumT} \left\{ \begin{array}{l}
\text{‘just} \\
\text{‘nothing}
\end{array} \right\} \\
& \text{‘just} \mapsto \Sigma A \lambda \cdot 1 \\
& \text{‘nothing} \mapsto 1
\end{align*}
\]

As before, we let the reader verify that the interpretation of this code is isomorphic to the expected functor:

\[ \text{MaybeD} A \] X \cong A + 1 \]

4.23 Definition (Tagged description). From the examples above, we observe that a typical datatype definition consists of a \( \Sigma \) code whose first argument enumerates the constructors. Codes fitting this pattern are called tagged descriptions. Again, this is a clear heritage of the sum-of-products style. To capture this constructor-oriented form in type theory, we define tagged descriptions as pairs of an enumeration of constructors and, for each constructor, its code:

\[
\text{tagDesc} : \text{SET}_1 \\
\text{tagDesc} \mapsto (E : \text{EnumU}) \times \pi E \lambda \cdot \text{Desc}
\]

A tagged description naturally defines a description in sum-of-products form:

\[
\begin{align*}
\text{toDesc} (D : \text{tagDesc}) & : \text{Desc} \\
\text{toDesc} & (E, cs) \mapsto \Sigma (\text{EnumT} E) (\text{switch cs})
\end{align*}
\]

(4.24) We have seen that tagged descriptions are but descriptions with some more structure. Conversely, any description can be forced into this style by taking a single constructor:

\[
\begin{align*}
\text{toTagDesc} (D : \text{Desc}) & : \text{tagDesc} \\
\text{toTagDesc} & D \mapsto (\text{EnumT} \{\text{‘con}\}, (D, *))
\end{align*}
\]

Adopting a tagged approach does not reduce our expressive power. Beside, by enforcing a constructor-oriented form of datatypes, we ease the implementation of datatype transformations. The free monad construction (Section 6.2.2) is such an example.

4.25 Remark (Notation). Our definition of \( \text{tagDesc} \) forces a low-level definition style upon us: we must give a pair of an enumeration and the tuple of their codes. We informally adopt a more natural syntax that consists in listing each constructor tag paired
4. A Universe of Inductive Types

with its code. For natural numbers, we thus write:

\[
\text{NatD} : \text{tagDesc}
\]

\[
\text{NatD} \mapsto \begin{cases}
'0' & : 1 \\
'suc' & : \text{var} \times 1
\end{cases}
\]

We shall rely on the concatenative structure of tagged descriptions later. Given two tagged descriptions \(D_1 \triangleq (E_1, C_1)\) and \(D_2 \triangleq (E_2, C_2)\), we can define a tagged description \(D = D_1 + D_2\) by concatenating the constructor enumerations \(E_1 + E_2\) and gluing their codes \(C_1\) and \(C_2\) accordingly. Informally, we capture this concatenative structure by vertical juxtaposition, thus writing:

\[
D \triangleq \begin{cases}
E_1 & : C_1 \\
E_2 & : C_2
\end{cases}
\]

Finally, we shall transparently switch from tagged descriptions to descriptions, and back. The coercions \(\text{toDesc}\) and \(\text{toTagDesc}\) will therefore be kept silent, so as to remove such needless noise from our definitions. It will be clear, by the types, when to apply one or the other.

4.2. Initial Algebra Semantics

(4.26) The universe of description lets us build the signature functor of ML-like datatypes. However, there is a step missing between these signatures and the actual datatypes. A datatype is the least set closed under a signature, or put otherwise, it is the least fixpoint of the signature functor. To capture datatypes, we need to extend our type theory with a fixpoint operator. In categorical terms, we aim at giving an initial algebra semantics to our signatures. For a signature functor \(F\), the least fixpoint is given by the carrier of the initial \(F\)-algebra, denoted \(\mu F\). In this framework, the unique algebra homomorphism from the initial algebra to a given algebra \(\alpha : F X \rightarrow X\) corresponds to a function from \(\mu F\) to \(X\), which recursively applies \(\alpha\) bottom-up on the inhabitant of \(\mu F\).

However, not every signature functor admits an initial algebra. Typical examples include the powerset functor \(X \mapsto X \rightarrow \text{Bool}\) [Awodey, 2006], or, in a predicative setting, the functor \(X \mapsto (X \rightarrow \text{Bool}) \rightarrow \text{Bool}\) [Reynolds, 1984, Pitts, 1987]. This is why we restrict ourselves to descriptions, which define only strictly-positive functors (\(\S\) 4.13). Being strictly positive, these functors are monotone and always admit an initial algebra. We can legitimately extend our type theory with the least fixpoint of descriptions and the associated induction principle.

4.2.1. Least fixpoint

MODEL: Chapter4.Desc.Fixpoint, Chapter4.Desc.InitialAlgebra
Since we shall cast our constructions in a categorical framework, we first recall the
categorical concepts we rely on. We give the definition of the algebra of a functor,
organise algebras in a category, and focus on the initial objects of this category. We then
relate these initial algebras with inductive definitions.

4.27 Definition (F-algebra). Let \( F \) be an endofunctor on a category \( \mathcal{C} \).
An \( F \)-algebra is the pair of a carrier \( C \in |\mathcal{C}| \) and a morphism \( \alpha : F \circ C \to C \) in \( \mathcal{C} \).

4.28 Remark (Notation). When clear from the context, we shall leave out the signature
functor and simply talk about an algebra. Also, we shall talk about an algebra on \( C \),
meaning an algebra whose carrier is \( C \).

4.29 Example (List\(_A\)-algebra). Let \( \text{List}_A X \mapsto 1 + A \times X \) be the signature
functor of lists. An example of a \( \text{List}_A \)-algebra on \( \text{Nat} \) is:

\[
\begin{align*}
\alpha (xs : \text{List}_A \text{Nat}) & : \text{Nat} \\
\alpha (\text{inj}_r \ast) & \mapsto 0 \\
\alpha (\text{inj}_l (a, n)) & \mapsto \text{suc } n
\end{align*}
\]

4.30 Definition (Category of \( F \)-algebras). We organise the \( F \)-algebras in a category, de-
noted \( F \mathcal{-alg} \), whose objects are \( (C, \alpha) \), and for which a morphism from \( (C, \alpha) \) to \( (D, \beta) \) is a morphism \( f : C \to D \) in \( \mathcal{C} \) such that the following diagram commutes:

\[
\begin{array}{ccc}
F C & \xrightarrow{f} & F D \\
\alpha \downarrow & & \downarrow \beta \\
C & \xrightarrow{f} & D
\end{array}
\]

4.31 Definition (Initial \( F \)-algebra). The initial \( F \)-algebra is the initial object in the cate-
gory \( F \mathcal{-alg} \). Concretely, the initial algebra is the algebra \( (\mu F, in : F \mu F \to \mu F) \) such that,
for any algebra \( (X, \alpha : F X \to X) \), there exists a unique morphism \( \langle \alpha \rangle \) in \( \mathcal{C} \) making the
following diagram commute:

\[
\begin{array}{ccc}
F (\mu F) & \xrightarrow{\langle \alpha \rangle} & F X \\
\downarrow m & & \downarrow \alpha \\
\mu F & \xrightarrow{\langle \alpha \rangle} & X
\end{array}
\]

4.32 Remark (Terminology). In the functional programming community, the morphism
\( \langle \alpha \rangle \) is called the catamorphism, or fold.

4.33 Example. Applied to Example 4.29, initiality gives us a (unique) morphism \( \langle \alpha \rangle : \)
4. A Universe of Inductive Types

\[ \mu \text{List}_A \rightarrow \text{Nat} \] that satisfies:

\[ \text{List}_A \mu \text{List}_A \rightarrow \text{List}_A \text{Nat} \]

\[ \mu \text{List}_A \rightarrow \text{Nat} \]

That is, we have the equation:

\[ \forall xs : \text{List}_A \mu \text{List}_A. (\langle \alpha \rangle (\text{in} \, xs)) = \alpha (\text{List}_A \langle \alpha \rangle \, \text{xs}) \] (4.1)

By definition of List_A, xs is either inj_l * or inj_r (a, ys) with a : A and ys : \( \mu \text{List}_A \). The carrier of the initial algebra thus contains two kinds of objects, which correspond exactly to the constructors of the List datatype:

\[ \text{nil} \triangleq \text{in} (\text{inj}_l *) \]
\[ \text{cons} a \, \text{ys} \triangleq \text{in} (\text{inj}_r (a, \text{ys})) \]

for a : A and ys : \( \mu \text{List}_A \)

Further, we can split the equation (4.1) along these two cases:

\[ \langle \alpha \rangle \, \text{nil} = \langle \alpha \rangle (\text{in} (\text{inj}_l *)) \]
\[ \langle \alpha \rangle \, \text{cons} \, a \, \text{ys} = \langle \alpha \rangle (\text{in} (\text{inj}_r (a, \text{ys}))) \]
\[ = \alpha (\text{inj}_l *) \]
\[ = \alpha (\text{inj}_r (a, \langle \alpha \rangle \, \text{ys})) \]
\[ = 0 \]
\[ \text{suc} (\langle \alpha \rangle \, \text{ys}) \]

That is, we have defined the function that computes the length of a list:

\[ \langle \alpha \rangle \, \text{List}_A : \text{Nat} \]
\[ \langle \alpha \rangle \, \text{nil} \mapsto 0 \]
\[ \langle \alpha \rangle \, \text{cons} \, a \, \text{xs} \mapsto \text{suc} (\langle \alpha \rangle \, \text{xs}) \] (4.34)

**Computational interpretation.** As hinted at in the previous example, the existence of an initial \( F \)-algebra gives rise to the constructors of the \( \mu F \) datatype: they are the inhabitants of the carrier algebra given by \( \text{in} \) – such as \text{nil} and \text{cons} in Example 4.33.

Besides, the initiality of (\( \mu F, \text{in} \)) gives a catamorphism, of type

\[ \langle (\alpha : F \, X \rightarrow X) \rangle : \mu F \rightarrow X \]

satisfying the recursive equation:

\[ \forall xs : F \, \mu F. (\langle \alpha \rangle (\text{in} \, xs)) = \alpha (F (\langle \alpha \rangle \, \text{xs}) \]
4.2. Initial Algebra Semantics

along the recursively-defined constructors – such as \texttt{cons} in Example 4.33.

If the initial algebra exists, Lambek’s lemma establishes a correspondence between
the categorical concept of initial algebra and the type-theoretic notion of datatype con-
structor:

\textbf{4.35 Lemma (Lambek [1968]).} Let \( F : C \rightarrow C \) be a functor that admits an initial algebra
\((\mu F, \text{in}).\)

The initial algebra \( \text{in} : F \mu F \rightarrow \mu F \) is an isomorphism.

(4.36) \textbf{Computational interpretation.} In particular, Lambek’s lemma tells us that the \textit{only}
inhabitants of the carrier \( \mu F \) are the constructors obtained from \( F \mu F \). The initial algebra
\( \text{in} \) is the generic datatype constructor. Applied to our Example 4.33, this means that the
\textit{only} constructors of \texttt{List} are \texttt{nil} and \texttt{cons}.

(4.37) So far, we have used descriptions to represent signature functors in type theory.
Putting our type-theoretic hat and our categorical glasses on, we now interpret the
categorical concept of initial algebra (Definition 4.31) in type theory. Instead of consid-
ering any functor \( F \), we now focus on the functor \([D], \text{for } D \text{ a description. Being strictly}
positive, these functors admit an initial algebra. We can therefore legitimately extend
our type theory with an operator taking the least fixpoint of a description.

\textbf{4.38 Definition (Least fixpoint).} We extend our type theory with a new type former
\( \mu \) taking a description and returning its least fixpoint. Following Definition 4.31, the
inhabitants of the least fixpoint are the inhabitants of \([D] (\mu D)\), mapped through the
initial algebra \( \text{in} \) that plays the rôle of a generic datatype constructor:

\[
\begin{array}{c}
\Gamma \vdash D : \text{Desc} \\
\Gamma \vdash \mu D : \text{SET} \\
\Gamma \vdash \text{xs} : [D](\mu D) \\
\Gamma \vdash \text{in} \ \text{xs} : \mu D
\end{array}
\]

(4.39) \textbf{Computational interpretation.} In effect, this definition “ties the knot”: an inhabitant
of \( \mu D \) is a \([D]\)-structure for which every subnode – described by a \texttt{var} code – is a \( \mu D \)
itself. If one reads the functor \([D]\) as describing the signature of a datatype, the least
fixpoint of that functor corresponds to the least set closed under this signature, or put
otherwise, it is the set of tree-like structures (of finite depth) built from such nodes.

\subsection*{4.2.2. Examples}

(4.40) We can now define some actual datatypes and their constructors. We take this oppor-
tunity to recast our earlier examples as tagged descriptions. The resulting description
code is exactly the same, but the tagged syntax is more palatable.
4. A Universe of Inductive Types

4.41 Example (Natural number, Example 4.18).

\[
\begin{align*}
\text{Nat} & : \text{SET} \\
\text{Nat} & \mapsto \mu \left\{ \begin{array}{l}
\text{'0} : 1 \\
\text{'suc} : \text{var} \times 1 \\
\end{array} \right. \\
0 & : \text{Nat} \\
0 & \mapsto \text{in} [\text{'0}] \\
\text{suc} (n : \text{Nat}) & : \text{Nat} \\
\text{suc} n & \mapsto \text{in} [\text{'suc } n]
\end{align*}
\]

4.42 Remark (Notation). In the definition of constructors, we use the more lightweight tuple notation instead of pairs. Recall that this notation elaborates to the expected right-nested pairs terminated by the inhabitant of unit (¶ 3.12), i.e. we have:

\[
\begin{align*}
[\text{'0}] & \triangleq (\text{'0} , *) \\
[\text{'suc } n] & \triangleq (\text{'suc} , (n , *))
\end{align*}
\]

Note that the tuple \([\text{'suc } n]\) must not be confused with the constructor \text{suc } n: the former inhabits \([\text{NatD}] \text{Nat}\), while the latter inhabits \text{Nat}. The isomorphism \text{in} mediates the two sides. In effect, compared to the representation \(X \mapsto 1 + X\) based on sums, the tuple \([\text{'0}]\) corresponds to the more primitive \text{inj} _l\ast, while the tuple \([\text{'cons } n]\) corresponds to \text{inj} _r n.

4.43 Example (List, Example 4.20). Lists follow exactly the same pattern, apart from the presence of a parameter \(A\) and, for the \text{cons} constructor, the presence of a non-recursive argument of type \(A\):

\[
\begin{align*}
\text{List}(A : \text{SET}) & : \text{SET} \\
\text{List} A & \mapsto \mu \left\{ \begin{array}{l}
\text{'nil} : 1 \\
\text{'cons} : \Sigma A \lambda - . \text{var} \times 1 \\
\end{array} \right. \\
\text{nil} & : \text{List} A \\
\text{nil} & \mapsto \text{in} [\text{'nil}] \\
\text{cons} (a : A) (xs : \text{List} A) & : \text{List} A \\
\text{cons} a xs & \mapsto \text{in} [\text{'cons } a xs]
\end{align*}
\]

4.44 Example (Tree, Example 4.21). Binary trees, on the other hand, take an extra recurs-
4.2. Initial Algebra Semantics

sive argument in the ‘node case:

\[
\text{Tree} (A : \text{SET}) : \text{SET} \\
\text{Tree} \quad A \quad \mapsto \quad \mu \left\{ \begin{array}{l}
\text{leaf} : 1 \\
\text{node} : \text{var} \times \Sigma A \times . \text{var} \times 1
\end{array} \right. \\
\text{leaf} : \text{Tree} A \\
\text{leaf} \mapsto \text{in} ['\text{leaf}'] \\
\text{node} (lt : \text{Tree} A) (a : A) (rt : \text{Tree} A) : \text{Tree} A \\
\text{node} \, \text{l} a \, \text{r} t \mapsto \text{in} ['\text{node} \, \text{l} a \, \text{r} t']
\]

4.2.3. Induction

MODEL: Chapter4.Desc.Induction, Chapter4.Desc.Lifting

(4.45) Categorically, for an algebra \((X, \alpha)\), we know that the existence of an initial algebra induces a (unique) morphism – the catamorphism – from \(\mu D\) to \(X\). Type theoretically, this property corresponds to an operator taking an algebra and iterating the algebra over the recursive structure:

\[
(L (\alpha : J D K X) : X) : \mu D \rightarrow X
\]

(4.46) However, the catamorphism is inadequate for dependent computation. The catamorphism is simply typed and, as such, does not let us work on predicates. We need induction to write functions whose type depends on inductive data. The two notions are not estranged from each other: categorically, one can define induction from the catamorphism [Hermida and Jacobs, 1998, Fumex, 2012]. In intensional type theory, we must provide induction in the meta-theory [Paulin-Mohring, 1996, Geuvers, 2001]. To do so, we follow the guideline provided by the categorical construction – from which we shall reuse the notation and vocabulary – and, unsurprisingly, obtain an induction principle akin to the one known by type theorists [Dybjer, 1994, Benke et al., 2003].

4.47 Remark (Order of presentation). In the following, we define induction in a top-down manner. We choose to focus on giving the big picture before pitching into the nitty-gritty details. We start by defining induction (Definition 4.48). This definition depends on a canonical lifting \(\Box_D\) (Definition 4.52) and its associated lifting map \(\Box_D^\ast\) (Definition 4.59). We define these two operators \(a posteriori\), building on our intuition of induction to ease our way into the overall construction.

4.48 Definition (Induction). The induction principle is the dependent version of the catamorphism: given a predicate \(P : \mu D \rightarrow \text{SET}\) and an \(x : \mu D\), it establishes \(P \, x\). To do
so, the user must provide the *inductive step* $\alpha$:

\[
\text{induction} \ (\alpha : \Box_D P \to P \circ \text{in}) \ (x : \mu D) : P x
\]

\[
\text{induction} \ \alpha \ (\text{in} \ xs) \mapsto \alpha \ (\Box_D \ (\text{induction} \ \alpha) \ xs)
\]

Intuitively, $\Box_D$ – pronounced “everywhere in $D$” – represents the induction hypothesis (Definition 4.52): $\Box_D P \ x s$ states that $P$ holds in all subnodes of $xs$, i.e. $P$ holds everywhere in $D$. Following this reading, the function $\alpha$ of type $\Box_D P \to P \circ \text{in}$ builds a proof of $P$ from the induction hypothesis: in a proof by induction, this corresponds exactly to the inductive step. The canonical lifting map $\Box_D$ (Definition 4.59) populates the induction hypothesis $\Box_D$ by recursively calling induction on the subnodes of $xs$.

4.49 Remark (Type of the inductive step). To convey the idea that $\alpha$ is (actually) a $\Box_D$-algebra, we have used a compact notation for morphisms between predicates ($\urcorner$ 1.32) and wrote:

$$\alpha : \Box_D P \to P \circ \text{in}$$

Put explicitly, this type signature unfolds to:

$$\alpha : (xs : \llbracket D \rrbracket (\mu D)) \to \Box_D P \ x s \to P (\text{in} \ xs)$$

That is, given an inhabitant $xs$ of $\llbracket D \rrbracket (\mu D) \cong \mu D$ for which $P$ holds in all subnodes, we must prove that $P$ holds for the entire node $\text{in} \ xs$.

(4.50) **Computational interpretation.** The induction principle applies the inductive step $\alpha$ once, and then calls itself recursively on all subelements of $xs$ using the canonical lifting map $\Box_D$. In essence, this is nothing but recursion: we traverse the datatype $xs$ in a bottom-up manner, collecting the result along the way. It is however not just recursion: this computation also carries a logical content. Transporting this logical content across our datatype is what differentiates induction from the catamorphism.

(4.51) To represent the induction hypothesis, we rely on the operator $\Box_D$, called the *canonical lifting*. Intuitively, given an $xs : \llbracket D \rrbracket X$ and a predicate $P : X \to \text{SET}$, the canonical lifting states that all $x : X$ stored in the subbranches of $xs$ satisfy the predicate $P$. This corresponds exactly to the mathematical concept of an induction hypothesis.

4.52 Definition (Canonical lifting $\Box_D$). The canonical lifting is defined in Figure 4.2 by cases over the grammar of descriptions. It is reminiscent of the categorical presentation of Hermida and Jacobs [1998] and the algebraic study of Fumex [2012, §3.2].

(4.53) **Computational interpretation.** The definition of the canonical lifting follows the definition of the interpretation $\llbracket - \rrbracket$ closely: indeed, it merely lifts the interpretation of descriptions – defining endofunctors on $\text{SET}$ – over to functors on predicates. On all codes but $\text{var}$, it is identical and thus builds a $D$-structure. The key difference is in the treatment of the $\text{var}$ code, where $\Box_{\text{var}} P$ takes an $x : X$ and returns the set $P \ x$ of witnesses. In effect, $\Box_D P$ is a $D$-structure whose leaves contain witnesses that $P$ holds on
4.2. Initial Algebra Semantics

\[
\begin{array}{|l|l|l|}
\hline
\text{Meta-theory} & \text{Expression} & \text{Result} \\
\hline
\square_{(D\text{Desc})} (P : X \to \text{SET}) (xs : [D] X) & \text{SET} \\
\square_1 & P \times & I \\
\square_{\text{var}} & P \times & P_X \\
\square_{A \times B} & P \times (a, b) & \square_A P a \times \square_B P b \\
\square_{\Sigma S D} & P \times (s, d) & \square_{D^s} P d \\
\square_{\Pi S D} & P \times & (s : S) \to \square_{D^s} P (f s) \\
\hline
\end{array}
\]

Figure 4.2.: Canonical lifting

the corresponding subnodes of \(xs\).

(4.54) The canonical lifting \(\square_{D}\) can be seen as a monotonic predicate transformer, taking a predicate \(P : X \to \text{SET}\) on \(X\) – asserting a certain property on \(X\) – to the predicate \(\square_{D} : [D] X \to \text{SET}\) on \([D] X\). That latter predicate asserts that \(P\) holds on all subelements of \([D] X\). Interestingly, \(\square_{D}\) can be understood as an indexed refinement of \(D\): we shall clarify that connection in the indexed setting (Definition 5.23).

4.55 Example (Canonical lifting: natural numbers). The canonical lifting applied to the description of natural numbers unfolds to two cases. First, in the base case ‘0, we have:

\[\square_{\text{NatD}} P [\text{‘0}] \sim 1\]

That is, there is no induction hypothesis. On the inductive step ‘\text{suc}\ from \(n\), we have:

\[\square_{\text{NatD}} P [\text{‘suc} n] \sim P n \times 1\]

That is, the induction hypothesis gives us \(P n\). During induction, from this proof of \(P n\), we have to prove \(P (\text{in} [\text{‘suc} n])\), i.e. \(P (\text{suc} n)\).

4.56 Example (Canonical lifting: lists). Applied to the description of lists, the canonical lifting also gives rise to two cases. First, in the base case ‘nil, we have, as expected, no induction hypothesis:

\[\square_{\text{ListD}} A P [\text{‘nil}] \sim 1\]

In the inductive step ‘\text{cons}\ from \(xs\), we have:

\[\square_{\text{ListD}} A P [\text{‘cons} a xs] \sim P xs \times 1\]

That is, the induction hypothesis gives us \(P xs\). During induction, from this \(P xs\), we have to prove \(P (\text{in} [\text{‘cons} a xs])\), i.e. \(P (\text{cons} a n)\).

4.57 Example (Canonical lifting: trees). Applied to the description of trees, the canonical lifting also gives rise to two cases. In the base case ‘leaf, we have no induction
hypothesis:
$$\square_{\text{Tree}\ D\ A\ P\ ['\text{leaf}]} \sim 1$$

In the inductive step ‘node’ from xs, we have:
$$\square_{\text{Tree}\ D\ A\ P\ ['\text{node}\ \text{lt}\ \text{a}\ \text{rt}]} \sim P\ \text{lt} \times P\ \text{rt} \times 1$$

That is, the induction hypothesis tells us that $P$ holds on the left branch $\text{lt}$ and the right branch $\text{rt}$. During induction, we then prove $P$ (in $['\text{node}\ \text{lt}\ \text{a}\ \text{rt}']$), i.e. $P (\text{node}\ \text{lt}\ \text{a}\ \text{rt})$.

(4.58) To describe the operational behavior of induction, we use the $\square_D$ operator. This operator lets us lift a decision procedure for a predicate $P$, i.e. a function $p: (x: X) \rightarrow P\ x$, to a decision procedure proving that $P$ holds in all subbranches of a $D$-structure, i.e. a function $\square_D\ p: (xs:\ [D]\ X) \rightarrow \square_D\ P\ xs$.

4.59 Definition (Lifting map $\square_D$). The lifting map is defined in Figure 4.3, again by cases over the grammar of descriptions. Following the definition of $\square_D$, it simply traverses the $D$-structure, only to apply $p$ on the subnodes coded by var.

(4.60) Computational interpretation. Applying $\square_D\ p$ corresponds to taking a recursive step over a $D$-structure, applying $p$ to all subnodes.

4.61 Remark (Categorical origin of $\square_D$). We have presented $\square_D$ as the functorial action of $\square_D$. We reflected this idea in our denomination and notation. From the type-theoretic definition, it might not be clear that this is indeed a functorial action. Recall that we presented $\square_D$ as a predicate transformer from predicates $P: X \rightarrow \text{SET}$ on $X$ to predicates $\square_D\ P: [D]\ X \rightarrow \text{SET}$ on $[D]\ X$ (4.54). $\square_D$ is therefore the object part of a functor on predicates.

The morphism part of $\square_D$ takes a morphism of predicate

$$Q \Rightarrow P \triangleq \forall\ x: X.\ Q\ x \Rightarrow P\ x$$
4.2. Initial Algebra Semantics

to a morphism of type

$$\Box_D Q \rightarrow \Box_D P \triangleq \forall xs : [D] X. \Box_D Q \, xs \rightarrow \Box_D P \, xs$$

Looking closely at the type of $$\Box_D$$, we observe a similar pattern:

$$\Box_D (f : (x : X) \rightarrow P \, x) : (xs : [D] X) \rightarrow \Box_D P \, xs$$

Massaging the type of $$f$$ and the type of the result, we have

$$\Box_D (f : 1 \, X \rightarrow P) : 1 \, [D] X \rightarrow \Box_D P$$

where $$1 \, X \triangleq \lambda x : X. \bot$$ is the trivial predicate over $$X$$, called the terminal object functor [Fumex, 2012, Definition 1.2.13]. By unfolding the arrow of predicates (¶ 1.32), the reader will check that this transformation is purely notational.

Because $$\Box_D$$ is a canonical lifting, we have that $$\Box_D (1 \, X) \cong 1 ([D] X)$$ [Fumex, 2012, Definition 3.1.8]. We thus obtain the following equivalent type for $$\Box_D$$

$$\Box_D (f : 1 \, X \rightarrow P) : \Box_D (1 \, X) \rightarrow \Box_D P$$

thus justifying the name of lifting map for $$\Box_D$$: it corresponds exactly to the morphism part of $$\Box_D$$ applied to a predicate morphism whose domain is the trivial predicate $$1 \, X$$.

Note that we could provide the morphism part of $$\Box_D$$ in full generality, and then specialise it to predicates of trivial domain. We refrained from doing so to keep our presentation simpler. We will not need the morphism part in its full generality.

4.62 Remark (Categorical interpretation [Hermida and Jacobs, 1998, Fumex, 2012]). As hinted at by the previous remark, our definition of induction is but an instance of a more general categorical pattern. In order to put our presentation into perspective, let us clarify the correspondence between the categorical structures and our type-theoretic definitions. First, we have already pointed out that $$\Box_D$$ (Definition 4.52) defines the object part of a functor from the slice over $$X$$ to the slice over $$[D] X$$. This functor is known in the literature as the canonical lifting [Fumex, 2012, Theorem 3.1.13]. We also remarked that $$\Box_D$$ (Definition 4.59) is the morphism part of that functor on a subclass of predicate morphisms. From the canonical lifting, we have defined the induction principle (Definition 4.48). We remarked that the inductive step is nothing but a $$\Box_D$$-algebra in disguise (Remark 4.49). This is justified categorically by the fact that induction is derivable from the initial algebra semantics of $$\Box_D$$-algebras [Fumex, 2012, Corollary 4.3.3].

(4.63) The induction principle is a generic operation. For any datatype we describe, it automatically comes equipped with an induction principle. Induction is a datatype-generic function: it is defined once and for all datatypes to come. Note that we had to extend the meta-theory to provide it. Indeed, induction is not derivable within the type theory [Geuvers, 2001], or even realisable [Paulin-Mohring, 1989]. In other words, we cannot write induction as a generic program in the type theory: it must be provided by the meta-theory. Having bitten the bullet and introduced a generic induction principle,
4. A Universe of Inductive Types

we are then able to derive generic programs without further extending the type theory.

4.64 Remark (Greatest fixpoint and terminal algebra semantics). The very same functors \([D]\) also admit greatest fixpoints [Morris, 2007]. We could legitimately extend our type theory to support coinduction, i.e. their terminal algebra semantics [McBride, 2009, Fumex, 2012]. The treatment of coinduction is beyond the scope of this thesis.

4.3. Extending Type Propagation

(4.65) The universe-based approach offers many benefits. First, its meta-theory is clearly delimited, unlike a generative approach à la Coq and Agda. Indeed, in those provers, a positivity checker verifies the validity of the user’s inductive definition and axiomatically extends the type theory with the corresponding set former, constructors, and elimination principle: such a type theory is thus open-ended, unlike ours. Second, it is strongly tied to the categorical concept of initial algebra. Third, for the programmer, it provides a generic programming system: one can write functions that are defined over all – or particular subclasses of – descriptions.

However, we should not lose usability from sight: while we expect generic programs to explicitly manipulate the inner-codings of inductive types, we must make sure that programming with specific datatypes is natural. In particular, during datatype-specific programming, a programmer should not be faced with the low-level encoding of datatypes.

(4.66) A first obstacle to specific programming is the inherently low-level presentation of constructors. Our type theory has a unique, generic constructor \(\text{in}\), from which all other constructors are coded. For instance, in Example 4.41, we had to manually define the constructors of the \(\text{Nat}\) datatype. By extending the bidirectional type checker, we can add constructors to our expression language. The type checker automatically elaborates the constructor expression to its low-level representation as nested tuples.

(4.67) Elaboration of constructors. Upon elaborating an expression \(c\ a_0\ldots a_k\) against the fixpoint of a tagged description, we replace this elaboration problem with the one consisting in elaborating the tuple of the constructor tag and the arguments:

\[
\Gamma \vdash \mu (\Sigma (\text{Enum}T E) T) \ni \text{in } [c\ a_0\ldots a_k]_{\text{Chk}} \rightsquigarrow t
\]

\[
\Gamma \vdash \mu (\Sigma (\text{Enum}T E) T) \ni c\ a_0\ldots a_k \rightsquigarrow t
\]

4.68 Example (Elaborating the constructors of \(\text{Nat}\)). Now we can write the constructors
of \( \text{Nat} \) directly, knowing that they are elaborated to the desired terms:

\[
\begin{align*}
\Gamma \vdash \text{Nat} \ni \text{in} \left[ \text{'0} \right] \text{Chk} & \quad \Gamma \vdash \text{Nat} \ni \text{in} \left[ \text{'suc} \, \text{n} \right] \text{Chk} \\
\Gamma \vdash \text{Nat} \ni \text{in} \left[ \text{'0} \right] \text{Chk} & \quad \Gamma \vdash \text{Nat} \ni \text{in} \left[ \text{'suc} \, \left( \text{n}, \ast \right) \right] \text{Chk}
\end{align*}
\]

The type explains the legible presentation, the constructor form, as well as the low-level representation, a right-nested tuple.

**4.69 Remark.** The elaboration of constructor expressions justifies our care in always terminating the telescopes of constructors’ arguments with a unit type. Indeed, we elaborate a constructor as a tuple of its tag and arguments. Recall that tuples are elaborated as \( \Sigma \)-telescopes terminated by a unit type (§3.12). The presence of this terminating unit type is therefore key to our ability to use the constructor expressions.

**4.70** A second obstacle to usability is the definition of datatypes. At the moment, to define an inductive type, we have to code its signature functor with a description. Then, we must manually build its least fixpoint. In Chapter 7, we present an elaboration of inductive definitions down to these codes. In the meantime, we freely use the high-level notation introduced in Section 3.2.2, with the confidence that it can be elaborated to descriptions.

**4.71** We believe that, with these two devices, a universe-based presentation of inductive types is usable for specific programming. Thanks to constructor expressions, the programmer can write standard constructors, instead of using the generic \text{in} constructor. Inductive definitions ought to automatically and transparently translate to descriptions. And finally, using the \text{by} \ (\Leftarrow) gadget, we can readily appeal to induction on inductive types. In other words, we can already write programs manipulating datatypes.


4. A Universe of Inductive Types

Conclusion

In this chapter, we have extended our type theory with a universe of descriptions, \textit{Desc}, that captures inductive types. Using this toy universe, we have presented the standard equipment of data-types: constructors and induction. We gained an intuition for descriptions by developing a few examples and relating our constructions to categorical concepts. While this universe does not let us describe \textit{dependent} inductive types, its simplicity makes it an excellent platform for experimentation.

Related work

The use of universes to internalise inductive definitions goes back to Martin-Löf’s wellorderings [Martin-Löf, 1984], or W-types for short. However, this presentation was meant for an extensional type theory. More intensional characterisations of inductive definitions were later given by Pfeifer and Ruess [1998] and Benke et al. [2003], with – already – an eye for generic programming. Our presentation is itself inspired by the work of Morris [2007], where we have left aside the internal fixpoints. Our original universe [Chapman et al., 2010] was a restricted version of the coding scheme for inductive-recursive definitions [Dybjer and Setzer, 1999]. In this thesis, influenced by the work of Gambino and Kock [2013] and Gambino and Hyland [2004], we directly gave an algebraic presentation organised around the \( \Pi \) and \( \Sigma \) codes. As we shall see in the next chapter, the interpretation of descriptions computes containers and therefore, extensionally, the datatypes described by our universe are equivalent to the W-types. Our system is therefore as expressive as W-types, whilst being suitable for an intensional type theory.
5. A Universe of Inductive Families

(5.1) So far, we have explored the realm of inductive types, building on intuitions from ML datatypes, using type dependency as a descriptive tool in DESC and its interpretation. Let us now make dependent types the object of our study. Dependent datatypes provide a way to work at higher level of precision \textit{a priori}, reducing the sources of failure we might otherwise need to manage. For the perennial example, consider vectors - lists indexed by length. By making length explicit in the type, we can prevent hazardous operations (the type of head demands vectors of length \textit{suc} \textit{n}) and offer stronger guarantees (pointwise addition of \textit{n}-vectors yields an \textit{n}-vector). However, these datatypes are not individually inductive. For instance, we have to define the whole family of vectors mutually, in one go:

\[\text{data Vec } [A : \text{SET}] (n : \text{Nat}) : \text{SET} \text{ where} \]
\[\quad \text{Vec } A \ (n = 0) \ni \text{nil} \]
\[\quad \text{Vec } A \ (n = \text{suc} \text{n'}) \ni \text{cons} \ (\text{n'} : \text{Nat}) (a : A) (\text{vs} : \text{Vec } A \text{n'})\]

The definition of vectors indexed by \textit{suc} \textit{n} depends on the definition of vectors indexed by \textit{n}, and so on until reaching the empty vector indexed by 0. On the other hand, we could instantiate the parameter \textit{A} to, say, Booleans and such a specialised definition would be perfectly valid: parameters are used uniformly in the datatype definition, unlike indices. In dependently-typed languages, the basic grammar of datatypes is that of inductive families. To capture this grammar, we must account for indexing.

5.1. The Universe of Indexed Descriptions

\textbf{MODEL: Chapter5.IDesc}

(5.2) We presented the DESC universe as a grammar of strictly-positive endofunctors on SET and developed inductive types by taking a fixpoint. To describe inductive families indexed by \textit{I} : \text{SET}, we play a similar game with endofunctors on the category \text{SET}^\text{I}, whose objects are families of sets \textit{X}, \textit{Y} : \text{I} \rightarrow \text{SET}, and whose morphisms are families of functions in \textit{X} \rightarrow \textit{Y} \triangleq \forall i : \text{I}. \textit{X} \iota \rightarrow \textit{Y} \iota (where the index \textit{i} is kept implicit, as per ¶3.55).

5.3 Remark (Slices and exponentials). Let \textit{C} be a category. Let \textit{I} be an object of \textit{C}.

The slice category \textit{C}/\textit{I} is defined as follows:

\textbf{Objects:} the functions \textit{P}, \textit{Q} of codomain \textit{I}

\textbf{Morphisms:} A morphism from \textit{P} to \textit{Q} is a function \textit{f} : \text{dom}(\textit{P}) \rightarrow \text{dom}(\textit{Q}) such that \textit{P} = \textit{Q} \circ \textit{f}
5. A Universe of Inductive Families

The category $\text{SET}^I$ – i.e. the functor category from the discrete category $I$ to $\text{SET}$ – and the slice category $\text{SET}/I$ are equivalent. We shall therefore conflate the two categories and talk mainly about slices of sets, by categorical habits, even though we often work with the exponential presentation in type theory.

(5.4) At the level of signature functors, this change of category gives access to a richer language of signatures. Endofunctors on $\text{SET}$ let us describe mono-sorted signatures: there is only one sort of operation. When defining a signature functor from $\text{SET}/I$ to $\text{SET}/J$, the inhabitants of the index $I,J$ can be understood as sorts. By picking a particular index $i:I$ at a recursive argument, we statically enforce that subnodes at that argument must be of sort $i$. By restricting which operations are exposed at index $j:J$, we make sure that only operations of sort $j$ are available.

5.5 Example (Vectors). A typical example of a $\text{Nat}$-sorted signature are vectors. The nil operation is only available for vectors of sort $0$. On the other hand, the cons operation is only available for vectors of strictly-positive sort $\text{suc}\ n$ and it recursively requires a vector of sort $n$. Thanks to this indexing discipline, we guarantee that a vector of sort $n$ is exactly of length $n$.

5.6 Remark. We are interested in describing functors from $\text{SET}^I$ to $\text{SET}^J$. However, we have that

$$[\text{SET}^I,\text{SET}^J] \cong [\text{SET}^I,\text{SET}]^J$$

where $[C,D]$ denotes the category of functors from $C$ to $D$. In type theory, this corresponds to a straightforward currying-uncurrying:

$$(I \to \text{SET}) \to (J \to \text{SET}) \cong J \to (I \to \text{SET}) \to \text{SET}$$

This isomorphism lets us focus on describing functors from $\text{SET}^I \to \text{SET}$, with the $J$-indexing being pulled out in the exponential.

(5.7) As for descriptions (Definition 4.6), we obtain an intensional presentation of multi-sorted signature through a universe construction. Following Remark 5.6, we focus first on describing signature functors in the category $[\text{SET}^I,\text{SET}]$. In Definition 5.12, we shall lift this universe to endofunctors on slices of $\text{SET}$. Our universe is thus defined by a set of codes $\text{IDesc}$ and its extension $[\_\_\_\_\_\_]$, whose types are:

$$\text{IDesc}\ (I:\text{SET}) : \text{SET} \quad [\_\_\_\_\_\_] : \text{IDesc}\ I \to (I \to \text{SET}) \to \text{SET}$$

An $\text{IDesc}$ code is a syntactic object describing a functor from $\text{SET}^I$ to $\text{SET}$. To compute its extension, we interpret the code with $[\_\_\_\_\_]$.

5.8 Definition (Indexed descriptions $\text{IDesc}$). The universe of indexed descriptions is defined in Figure 5.1. It is closed under the monotonic type formers:

- 1 codes the unit type;
- $\Sigma$ and $\Pi$ respectively code $\Sigma$-types and $\Pi$-types of constant domain;
- $\sigma$ and $\times$ respectively code $n$-ary sum and binary product, using a first-order representation;
- $\text{var}$ codes a recursive argument, taken at a given sort $i:I$. 

88
5.9 Remark (First-order representations). In Remark 4.7, we discussed the benefits of having first-order representation in Desc. The same remark applies here: besides the general-purpose connectives $\Pi$ and $\Sigma$, we offer their first-order counterparts $\times$ and $\sigma$.

Extensionally, we have:

\[
\begin{align*}
[A \times B] X & \cong [\Pi \text{Bool} \{ 'true \mapsto A \}, 'false \mapsto B \}] X \\
[\sigma \ E \ T] X & \cong [\Sigma (\text{EnumT} \ E) \lambda e. \text{switch} \ T \ e] X
\end{align*}
\]

However, intensionally, $\sigma$ and $\times$ are more prescriptive: they encode the first-order nature of the resulting datatype, this structure being then exploitable inside type theory. For example, any datatype described through a combination of $1$, $\sigma$, $\times$, and $\text{var}$ admits a decidable equality (Example 7.96).

5.10 Remark ($\text{IDesc}$ is an inductive definition). The code $\text{IDesc} \ I$ (Definition 5.1a) is not obviously inductive. Indeed, the argument $T$ of the $\sigma$ code has type $\pi E (\lambda \ - \ . \ \text{IDesc} \ I)$. We must make sure that $\pi$ computes to a type where $\text{IDesc} \ I$ does not occur to the left of an arrow, so as to ensure the strict positivity of this definition. A closer inspection of $\pi$ (Definition 2.49) reveals that $\pi$ is indeed strictly positive in the predicate.

So far, we have given an intensional characterisation of the signature functors on $[\text{SET}, \text{SET}]$. To capture functors between slices of $\text{SET}$, we apply the isomorphism presented earlier (Remark 5.6) and obtain a universe describing multi-sorted signatures.

5.12 Definition (Universe of indexed descriptions func). We obtain the universe of (indexed) descriptions $\text{func}$ by simply pulling the $J$-index to the front. The interpretation of indexed descriptions extends pointwise to $\text{func}$:

\[
\text{func} (I : \text{SET}) (J : \text{SET}) : \text{SET} \quad \quad \quad \quad (D : \text{func} \ I \ J) (X : I \to \text{SET}) : J \to \text{SET} \\
\text{func} I J \mapsto I \to \text{IDesc} \ I \\
D X \mapsto \lambda j. [D j] X
\]

Inhabitants of the $\text{func}$ type are called descriptions, or indexed descriptions when the context is ambiguous.

5.13 Example (Indexed description: vectors). We are now able to express the signature functor of vectors (Example 5.5) as a description. Recall that $\text{nil}$ is only available at index 0, while $\text{cons}$ is only available at index $\text{suc n}$ and requires an argument of index $n$. The most natural way to define this signature is by pattern-matching on the index: if it is 0, we describe the $\text{nil}$ constructor; if it is strictly positive, we describe the $\text{cons}$ constructor:

\[
\begin{align*}
\text{VecD} (A : \text{SET}) & : \text{func} \ \text{Nat} \ \text{Nat} \\
\text{VecD} A & \quad 0 \mapsto 1 \\
\text{VecD} A (\text{suc n}) & \mapsto \Sigma A \lambda \var n \ . \ \text{var} \ n \times 1
\end{align*}
\]

5.14 Remark (Strictly-positive family). By induction on the interpretation function, we observe that the functors captured by the $\text{func}$ universe are strictly positive, by construction [Morris et al., 2009]. In Section 5.2, we develop their initial algebra semantics. In Section 5.3.2, we prove that the class of descriptions is exactly equivalent to contain-
5. A Universe of Inductive Families

(a) Code

\[
\begin{align*}
\Gamma \vdash I : \text{SET} & \quad \Gamma \vdash \text{IDesc} I : \text{SET} \\
\Gamma \vdash \text{var} I : \text{IDesc} I & \quad \Gamma \vdash \text{VALID} : \text{IDesc} I \\
\Gamma \vdash A : \text{IDesc} I & \quad \Gamma \vdash B : \text{IDesc} I \\
\Gamma \vdash A \times B : \text{IDesc} I & \\
\Gamma \vdash E : \text{EnumU} & \quad \Gamma \vdash T : \pi E (\lambda_\text{Desc} I) \\
\Gamma \vdash \sigma E T : \text{IDesc} I & \\
\Gamma \vdash S : \text{SET} & \quad \Gamma \vdash T : S \rightarrow \text{IDesc} I \\
\Gamma \vdash \Pi S T : \text{IDesc} I & \\
\Gamma \vdash S : \text{SET} & \quad \Gamma \vdash T : S \rightarrow \text{IDesc} I \\
\Gamma \vdash \Sigma S T : \text{IDesc} I & 
\end{align*}
\]

(b) Interpretation

Figure 5.1.: Universe of indexed descriptions
5.2. Initial Algebra Semantics

At a high-level, the following development matches point by point our presentation of induction for (non-indexed) descriptions in Section 4.2. Indeed, categorically, adding indices requires but a mild generalisation of the non-indexed framework [Fumex, 2012, Chapter 5] to justify, for example, computations over inductive families that simultaneously compute over the index [Fumex, 2012, Example 5.2.3]. We shall not dwell on the categorical aspects here. However, when faced with an overwhelmingly indexed definition, the reader should not hesitate to confront it to its non-indexed counterpart to grasp its essence.

5.2.1. Least fixpoint

MODEL: Chapter5.IDesc.Fixpoint, Chapter5.IDesc.InitialAlgebra

5.16 Definition (Least fixpoint). We extend the type theory with a fixpoint operator. Given the description of an endofunctor on \( \text{SET} \), we compute the fixpoint at some index \( i \) and tie the knot with the fixpoint \( \mu D \):

\[
\Gamma \vdash I : \text{SET} \\
\Gamma \vdash D : \text{func} I I \\
\Gamma \vdash i : I \\
\Gamma \vdash x : [D] (\mu D) i \\
\Gamma \vdash \text{in} x : \mu D i
\]

This is a generalisation of the least fixpoint of descriptions (Definition 4.38).

5.17 Remark (Indexed catamorphism). Categorically, we know that, for an indexed algebra \([D] X \rightarrow X\), there exists a (unique) function from \( \mu D i \) to \( X i \). In type theory, this corresponds to the catamorphism:

\[
((\alpha : [D] X \rightarrow X)) : \mu D \rightarrow X
\]

In this section, we focus on induction, which provides dependent computation. The catamorphism is derivable from induction, as we shall see in Section 6.2.1.

5.2.2. Induction

MODEL: Chapter5.IDesc.Induction, Chapter5.IDesc.Lifting

We equip these initial algebras with a generic induction principle. Again, the following definition of induction mirrors the non-indexed one (Definition 4.48). The (only)
5. **A Universe of Inductive Families**

The difference is that predicates are defined on the pair of an index and an element of the inductive type at that index.

5.19 **Remark** (Overloaded notation). Following the non-indexed presentation, indexed induction is defined by a canonical lifting and its associated map. We shall overload notation and write $\square_D$ for the canonical lifting of an indexed description $D$ (Definition 5.23), and $\square_D^\to$ for its maps (Definition 5.27). Thanks to the type of $D$, it always clear which of the indexed or non-indexed liftings we are referring to.

5.20 **Definition** (Induction). For a predicate $P : \mu D i \to \text{SET}$ and a term $x : \mu D i$, the induction principle lets us prove $P x$. To do so, we must prove the inductive step $\alpha$:

\[
\text{META-THEORY}
\]

\[
\text{iinduction } \alpha : \square_D P \Rightarrow P \circ \text{in} \quad \text{(in } xs) \mapsto \alpha (\square_D^\to (\text{iinduction } \alpha) \text{xs})
\]

Computationally, we take an inductive step $\alpha$ and, then, recursively apply induction on the substructures through the lifting map $\square_D^\to$.

5.21 **Remark** (Inductive step, explicitly). Following Remark 4.49, we have written the inductive step using the more concise arrow of predicates:

\[
\alpha : \square_D P \Rightarrow P \circ \text{in}
\]

It is a good exercise to desugar this definition. We obtain:

\[
\alpha : (i : I)(xs : [D] (\mu D) i) \to \square_D P \text{xs} \Rightarrow P (\text{in xs})
\]

That is, the inductive steps takes, for an index $i$, an inhabitant $xs$ of the least fixpoint at $i$ for which the induction hypothesis holds, i.e. $P$ holds in all subelements of $xs$. We then have to prove that $P$ holds for $\text{in } xs$.

(5.22) To define the inductive step, we use the canonical lifting operator $\square_D$. Its role (and definition) is akin to the non-indexed canonical lifting (Definition 4.52). Following Remark 4.54, $\square_D$ can be understood as a monotone predicate transformer, i.e. a functor, from a predicate $P : (i : I) \times X i \to \text{SET}$ on $(i : I) \times X i$ to a predicate $\square_D P : [D] X \to \text{SET}$ on $[D] X$. Interestingly, we can represent this functor with an indexed description. We thus define $D\square_D$, a description indexed by $(i : I) \times X i$ and $[D] X$. We obtain the canonical lifting $\square_D$ by interpreting $D\square_D$ at $P$.

5.23 **Definition** (Canonical lifting $\square_D$). The (internalised) definition of $\square_D$ is given in Figure 5.2. In the non-indexed case (Definition 4.52), we have defined the canonical lifting with set formers: here, we simply replace these connectives by the corresponding IDesc codes. Apart from this specificity, the definition is exactly the same. This definition then lifts pointwise to $\text{func}$. Overloading notation, we obtain a description indexed
5.2. Initial Algebra Semantics

\[
\begin{align*}
\square_{(\text{Desc } I)} (\mu D \times X) & : \text{Desc } ((i : I) \times X_i) \\
\square_1 & : * \to 1 \\
\square_{\text{var } i} & : \text{var}(i, x) \\
\square_{A \times B} & : \text{var}(a, b) \to \square_A a \times \square_B b \\
\square_{\text{ED}} & : (e, d) \to \square_\text{switch } D e d \\
\square_{\text{ISD}} & : f \to \Pi \lambda s. \square_D s (f s) \\
\square_{\text{ESD}} & : (s, d) \to \square_D s d
\end{align*}
\]

Figure 5.2.: Canonical lifting (internalised)

from predicates on \((i : I) \times X_i\) to predicates on \((j : J) \times [D] X_j\):

\[
\square_{(\text{func } I J)} : \text{func } ((i : I) \times X_i) ((j : J) \times [D] X_j)
\]

from which we derive the predicate transformer \(\square_D\), by interpreting the description:

\[
\square_{(\text{func } I J)} (P : (i : I) \times \mu D i \to \text{SET}) (\mu D : (i : I) \times [D] \mu D) i : \text{SET} \\
\square_D P xs \to [D \square_D] P xs
\]

5.24 Example (Canonical lifting: vectors). Applied to the description of vectors (Example 5.13), the canonical lifting gives rise to two cases depending on the value of the index \(n : \text{Nat}\). First, for a vector of length 0, \textit{i.e.} the empty vector, the lifting provides no induction hypothesis:

\[
\square_{\text{Vec } A} P [\text{nil}] \sim 1
\]

For a vector of length \(\text{suc } n\), \textit{i.e.} a constructor node, the lifting unfolds to:

\[
\square_{\text{Vec } A} P [\text{cons } a xs] \sim P (n, xs) \times 1
\]

That is, the induction hypothesis gives us \(P xs\).

5.25 Remark. This definition of the canonical lifting is an example of \textit{internalising} the definition of (part of) the type theory into itself. This is made possible by our universe-based approach: because descriptions and, therefore, monotone functors are syntactically reified in our type theory, we can reuse these basic blocks in its own definition.

5.26 Remark (A generic program). Beyond the definition of the induction principle, the canonical lifting is useful on its own. It is indeed a generic program that asserts that a property \(P\) holds \textit{everywhere} in the subnodes of \(xs : [D] X\). For instance, we rely on the canonical lifting to define algebraic ornaments (Definition 8.41).

5.27 Definition (Lifting map \(\square_D\)). Following the non-indexed lifting map (Definition 4.59), we define the lifting map in Figure 5.3. It consists of a case analysis on the indexed de-
5. A Universe of Inductive Families

### Meta-theory

\[
\begin{align*}
\boxrightarrow{\text{Desc}} & (p : (x : X i) \to P x) \ (xs : [D] X) : \boxrightarrow{D} P xs \\
\boxrightarrow{1} & p \quad * \quad \mapsto * \\
\boxrightarrow{\text{var}} & p \quad x \quad \mapsto p x \\
\boxrightarrow{A \times B} & p \quad (a, b) \quad \mapsto (\boxrightarrow{A} p a, \boxrightarrow{B} p b) \\
\boxrightarrow{\sigma E D} & p \quad (e, d) \quad \mapsto \boxrightarrow{\text{switch}} D e p d \\
\boxrightarrow{\Pi S D} & p \quad f \quad \mapsto \lambda s. \boxrightarrow{D} a p (f a) \\
\boxrightarrow{\Sigma S D} & p \quad (s, d) \quad \mapsto \boxrightarrow{D} s p d
\end{align*}
\]

Figure 5.3.: Lifting map

scription. Again, the construction is essentially the same, modulo some indexing noise. This definition lifts pointwise to func and we overload notation.

5.28 Remark (Categorical origin of \(\boxrightarrow{D} \)). In the non-indexed setting, we explained how the type of \(\boxrightarrow{D} \) can be massaged to reveal its functorial origin (Remark 4.61). The same reasoning can be carried out on \(\boxrightarrow{D} \).

(5.29) Building on our study of induction for descriptions, we have extended our type theory with least fixpoints of indexed descriptions. We then provided induction on such fixpoints, based on the canonical lifting and its action map. We thus obtain a generic elimination principle for inductive families: we can program with inductive families.

5.2.3. Examples

**MODEL:** Chapter5.IDesc.Examples

(5.30) In this section, we show some examples of inductive families and their encoding in our universe.

5.31 Example (From Desc to IDesc 1). First of all, indexed descriptions subsume non-indexed descriptions. A non-indexed description is but a description indexed by the unit set. We could manually recast our earlier examples of descriptions (Section 4.1.1) in the indexed setting. Better still, we can write a generic program that computes the indexed code from the non-indexed one:

\[
\begin{align*}
tolDesc (D : \text{Desc}) & : \text{IDesc 1} \\
tolDesc 1 & \mapsto 1 \\
tolDesc \text{var} & \mapsto \text{var} * \\
tolDesc (A \times B) & \mapsto (tolDesc A) \times (tolDesc B) \\
tolDesc (\Sigma S D) & \mapsto \Sigma S \lambda s. \text{tolDesc} (D s) \\
tolDesc (\Pi S D) & \mapsto \Pi S \lambda s. \text{tolDesc} (D s)
\end{align*}
\]
5.2 Initial Algebra Semantics

5.32 Remark. Formally, we are not yet able to write `toIDesc` in our type theory. It is defined by induction over `Desc` codes but we have not provided an induction principle for the inductively-defined type `Desc`! In Chapter 6, we show that it is not necessary to add such an induction principle to the meta-theory. By presenting `Desc` as the fixpoint of a description, we can use induction to define functions over the code of descriptions itself. In such a system, `toIDesc` is natively implementable.

5.33 Example (Natural numbers). For instance, our definition of natural numbers as a description (Example 4.18) lifts to:

```
toIDesc NatD ∼ Σ EnumT 
{ '0 :→ 1 
  'suc :→ var × 1 } 
```

We can similarly lift any inductive type to a trivially indexed inductive family.

5.34 Example (Mutually-inductive definition). With indexed descriptions, we can represent mutually-inductive datatypes. For, say $n$ mutually inductive definitions, the trick consists in writing a single description that is indexed by an enumeration of size $n$ [Paulin-Mohring, 1996, Yakushev et al., 2009]. For example, the mutual definition of trees and forests

```
data NTree [A: SET]: SET where 
  NTree A ⊃ node (a:A)(ts:NForest A) 

data NForest [A: SET]: SET where 
  NForest A ⊃ nil 
  | cons (t:NTree A)(ts:NForest A) 
```

is described by the following code:

```
TreeKind : SET
TreeKind ∼ EnumT { 'NTree 'NForest }

NTreesD (A:SET) : func TreeKind TreeKind
NTreesD A 'NTree ∼ { 'node : Σ A λ . var 'NForest × 1 
  'nil : 1 
  'cons : var 'NTree × var 'NForest × 1 

NTree (A:SET) : SET
NTree A ∼ μ (NTreesD A) 'NTree 

NForest (A:SET) : SET
NForest A ∼ μ (NTreesD A) 'NForest
```

(5.35) So far, our examples are mere inductive types, with no or very little indexing. We now consider some actual inductive families.

5.36 Example (Vectors, with constraints). In Example 5.13, we gave the signature functor of vectors. This definition style, by pattern-matching on the index, is not supported by mainstream theorem provers, such as Agda or Coq. In these systems, one would
write a definition like:

```haskell
data Vec [A : SET] (n : Nat) : SET where
Vec A (n = 0) ⊇ nil
Vec A (n = suc n') ⊇ cons (n' : Nat) (a : A) (vs : Vec A n')
```

This definition style is sometimes referred to as the “Henry Ford principle” [McBride, 1999]: the datatype is defined for any index n, as long as it satisfies some constraints. Thus, each constructor imposes its constraints – here using propositional equality – on the legitimate values of the index. For vectors, this corresponds to the following description:

```haskell
VecD (A : SET) : func Nat Nat
VecD A ↦→ λn. Σ EnumT {’nil ↦→ Σ (n = 0) λ – .1
’cons ↦→ Σ Nat λm. Σ (n = suc m) λ – .
Σ A λ – . var m × 1

Vec (A : SET) (n : Nat) : SET
Vec A n ↦→ µ (VecD A) n
```

That is, we may choose nil for any index we like as long as that index is 0; in the cons case, the index must be a successor of some number m and the recursive argument is taken at $m \triangleq n - 1$.

5.37 Remark (Choice of equality). In Remark 3.48, we hinted at the fact that our presentation of inductive families is orthogonal to a particular choice of propositional equality. The above example further illustrates our stance. We capture constraints on indices by falling back to propositional equality. Doing so, we remain agnostic about the actual definition of propositional equality: any will do. In this particular example, since equality of natural numbers is decidable, we could rely on the recursively-computed equality of natural numbers. Doing so, we entirely sidestep the question of propositional equality by using a decision procedure.

Conversely, our presentation of inductive families cannot be used to introduce a notion of propositional equality: one has to pre-exist for equations on indices to be expressible. However, to ensure backward compatibility, one could certainly expose an underlying propositional equality through the identity (data)type. This would be but a presentation trick.

5.38 Example (Vector, with computation). However, sometimes, equations are redundant. Looking back at Vec A, we find that the equations constrain the choice of constructor and stored the tail index retrospectively. But inductive families need not store their indices [Brady et al., 2003]! Indeed, recall that our indexed functors are built on $[\text{SET} / I, \text{SET}]$: we are actually defining a function of domain J. We can therefore compute over this index, and in particular perform case analysis.

In our example, analysing the incoming Nat-index, we can tidy up our description of
vectors, as we did in Example 5.13:

\[
\begin{align*}
  \text{VecD} (A : \text{SET}) : & \text{func} \ \text{Nat} \ \text{Nat} \\
  \text{VecD} \ A \ 0 & \mapsto \{ \text{nil} : 1 \\
  \text{VecD} \ A \ (\text{suc} \ n) & \mapsto \{ \text{cons} : \Sigma A \lambda . \var n \times 1 \\
  \text{Vec} (A : \text{SET}) (n : \text{Nat}) & : \text{SET} \\
  \text{Vec} \ A \ n & \mapsto \mu (\text{VecD} \ A) \ n
\end{align*}
\]

The choice of constructors and equations have simply disappeared.

5.39 Example (Finite sets, with constraints). A similar example is \(\text{Fin}\), the type of finite sets. It is traditionally presented with constraints:

\[
\begin{align*}
\text{data} \ \text{Fin} \ (n : \text{Nat}) : & \text{SET} \ \text{where} \\
\text{Fin} \ (n = \text{suc} \ n') \ & \ni \ f_0 \ (n' : \text{Nat}) \\
| \ f_{\text{suc}} \ (n' : \text{Nat}) (k : \text{Fin} \ n')
\end{align*}
\]

This definition is described by the following description:

\[
\begin{align*}
\text{FinD} : & \text{func} \ \text{Nat} \ \text{Nat} \\
\text{FinD} \ n & \mapsto \{ f_0 : \Sigma \text{Nat} \lambda m. \Sigma (n = \text{suc} \ m) \lambda . 1 \\
| f_{\text{suc}} : \Sigma \text{Nat} \lambda m. \Sigma (n = \text{suc} \ m) \lambda . \var m \times 1
\end{align*}
\]

5.40 Example (Finite sets, with computation). However, by matching on the index, we can eliminate the equations (but not the choice of constructors):

\[
\begin{align*}
\text{FinD} : & \text{func} \ \text{Nat} \ \text{Nat} \\
\text{FinD} \ 0 & \mapsto \Sigma 0 \ 0-\text{elim} \ - \text{No constructor} \\
\text{FinD} \ (\text{suc} \ n) & \mapsto \{ f_0 : 1 \\
| f_{\text{suc}} : \var n \times 1
\end{align*}
\]

5.41 Remark (Forcing and detagging [Brady et al., 2003]). This technique of extracting information by case analysis on indices applies to descriptions exactly where Brady’s forcing and detagging optimisations apply in compilation. They eliminate just those constructors, indices and constraints which are redundant even in open computation.

5.42 Definition (Forcing). Forcing consists in computing an argument \(x : X\) of a constructor from its index \(I\). Hence, for forcing to apply, we must have a function from \(I\) to \(X\). Our alternative presentation of \(\text{Fin}\) is obtained by forcing (Example 5.40): instead of storing an index \(m\), we pattern-match on the index \(n\) and use it directly in the recursive argument.

5.43 Definition (Detagging). Detagging consists in removing the choice of constructors by matching on the index. For detagging to apply, constructors must be in injective correspondence with the indices. Our presentation of \(\text{Vec}\) (Example 5.38) is obtained by detagging. By noticing whether the index is \(0\) or \(\text{suc}\), we deduce the vector constructor.

5.44 Definition (Tagged indexed descriptions). Let us reflect this index analysis tech-
5. A Universe of Inductive Families

We divide tagged indexed descriptions into two categories: first, the constructors that do not depend on the index, inhabiting $\text{Tags} I$; then, the constructors that do, inhabiting $\text{iTags} I$:

$$\text{tagDesc} \ (x: \text{SET}) \ : \ \text{SET}_1$$

$$\text{tagDesc} \ I \ \mapsto \ \text{Tags} I \times \text{iTags} I$$

The non-dependent part mirrors the definition of non-indexed descriptions:

$$\text{Tags} \ (x: \text{SET}) \ : \ \text{SET}_1$$

$$\text{Tags} \ I \ \mapsto \ (E: \text{EnumU}) \times (i: I) \rightarrow \pi \ E \lambda - . \text{IDesc} I$$

The index-dependent part indexes the choice of constructors. Thus, by inspecting the index, it is possible to vary the choice of constructors:

$$\text{iTags} \ (x: \text{SET}) \ : \ \text{SET}_1$$

$$\text{iTags} \ I \ \mapsto \ (F: I \rightarrow \text{EnumU}) \times (i: I) \rightarrow \pi \ F \ i \lambda - . \text{IDesc} I$$

(5.45) We have thus reflected our discipline for inductive definitions (Remark 3.52) back into type theory. As for tagged descriptions (¶ 4.23), this normal form is constructor-oriented and thus eases the implementation of datatypes transformations, such as the indexed free monad construction.

5.46 Remark (Notation). As for tagged description (Remark 4.25), we adopt a more intuitive notation for tagged signatures. To do so, we move from a mono-sorted signature style to a multi-sorted one by indicating the operation’s sort on the left. This affords us a pattern-matching notation to restrict operations to a certain pattern of indices.

For instance, our definition of $\text{Vec}$ (Example 5.38) is more concisely defined as:

$$\text{VecD} \ (A: \text{SET}) \ : \ \func \ \text{Nat} \ \text{Nat}$$

$$\text{VecD} \ A \ \mapsto \ \left\{ \begin{array}{ll}
0 \leftarrow '\text{nil} : 1 \\
\text{suc} \ n \leftarrow '\text{cons} : \Sigma \ A \lambda - . \text{var} \ n \times 1
\end{array} \right.$$ 

Similarly, the signature of the judgments of minimal logic (Example 3.51) is given by:

$$\vdash_D \ : \ \func (\text{Context} \times \text{Type}) (\text{Context} \times \text{Type})$$

$$\vdash D \ \mapsto \ \left\{ \begin{array}{ll}
(\Gamma, T) \leftarrow '\text{var} : \Sigma (T \in \Gamma) \lambda - . 1 \\
(\Gamma, T) \leftarrow '\text{app} : \Sigma \text{Type} \lambda S . \text{var} (\Gamma, S \Rightarrow T) \times \text{var} (\Gamma, S) \times 1 \\
(\Gamma, \text{unit}) \leftarrow '\ast : 1 \\
(\Gamma, A \Rightarrow B) \leftarrow '\lambda \text{m} : \text{var} (\Gamma; A, B)
\end{array} \right.$$ 

5.47 Remark (Notation). We write $\text{toDesc} D$ to denote the description computed from the tagged indexed description $D$. Its expansion is similar to the definition of $\text{toDesc}$ for tagged descriptions (Definition 4.23), except that it must also append the dependent and non-dependent parts. We shall transparently switch from tagged to raw descrip-
5.3. Categorical Semantics of Inductive Families

5.48 Example (Typed expressions). As a larger example, we define a syntax for a small language with two types, natural numbers and Booleans:

\[
\begin{align*}
\text{Ty} & : \text{SET} \\
\text{Ty} & \mapsto \text{EnumT} \{ '\text{nat'}, '\text{bool}' \}
\end{align*}
\]

This language has values, conditional expression, addition and comparison. Informally, their signatures are:

\[
\begin{align*}
\text{val} & : \text{Val} \; \text{ty} \rightarrow \text{ty} \\
\text{plus} & : '\text{nat}' \rightarrow '\text{nat}' \rightarrow '\text{nat}' \\
\text{cond} & : '\text{bool}' \rightarrow \text{ty} \rightarrow \text{ty} \rightarrow \text{ty} \\
\text{le} & : '\text{nat}' \rightarrow '\text{nat}' \rightarrow '\text{bool}'
\end{align*}
\]

The function Val interprets the object language types in the host language, so that arguments to val fit their expected type.

\[
\begin{align*}
\text{Val} (\text{ty} : \text{Ty}) & : \text{SET} \\
\text{Val} '\text{nat}' & \mapsto \text{Nat} \\
\text{Val} '\text{bool}' & \mapsto \text{Bool}
\end{align*}
\]

We take Nat and Bool to represent natural numbers and Booleans in the host language, equipped with addition and comparison:

\[
\begin{align*}
- + - & : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
- \le - & : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Bool}
\end{align*}
\]

We express our syntax as a tagged description, indexing over object types Ty:

\[
\begin{align*}
\text{ExprD} & : \text{tagIDescTy} \\
\text{ExprD} & \mapsto \begin{cases}
\text{ty} & \leftarrow '\text{val} : \Sigma (\text{Val ty}) \lambda .1 \\
\text{ty} & \leftarrow '\text{cond} : \text{var} '\text{bool} \times \text{var} \text{ty} \times \text{var} \text{ty} \times 1 \\
'\text{nat} & \leftarrow '\text{plus} : \text{var} '\text{nat} \times \text{var} '\text{nat} \times 1 \\
'\text{bool} & \leftarrow '\text{le} : \text{var} '\text{nat} \times \text{var} '\text{nat} \times 1
\end{cases}
\end{align*}
\]

Note that some constructors are always available, namely val and cond. On the other hand, the plus and le constructors are index-dependent, with plus available only when building a 'nat, le only for 'bool. The code reflects our specification, with the first two components uniformly offering val and cond, the next selectively offering plus or le.

5.3. Categorical Semantics of Inductive Families

(5.49) While the universe of descriptions is a good medium for programming in type theory, its syntactic nature hampers abstract reasoning. We would like to be able to reason more extensionally in order to focus on the abstract, algebraic properties of descriptions. To this end, we first recall a few definitions from the theory of (indexed) contain-
5. A Universe of Inductive Families

Containers model strictly-positive family in the language of locally Cartesian-closed categories [Seely, 1983] (LCCCs). For a purely diagrammatic presentation, we refer the reader to the work of Gambino and Hyland [2004] and Gambino and Kock [2013] on polynomial functors. We shall seamlessly use results from the polynomial functor literature, the two approaches being equivalent [Gambino and Kock, 2013, §2.18].

Following the standard presentation of containers [Morris and Altenkirch, 2009], we work exclusively in the internal language of LCCCs: extensional type theory [Hofmann, 1995, Curien, 1993]. This allows us to stay within type theory, with the proviso that, in this section, equality is extensional (Remark 5.51). In this framework, we formalise the connection between our universe-based presentation of datatypes and the theory of containers. In particular, we prove that the functors represented by our universe corresponds exactly polynomial functors. This key result lets us transparently switch from our concrete presentation of datatypes to the more abstract containers.

5.51 Remark (Extensional type theory). In extensional type theory, the following equality reflection rule is admissible

\[ \Gamma \vdash p : x = y \]
\[ \Gamma \vdash x \equiv y : A \]

In effect, it collapses the propositional equality \( x = y \) into the definitional equality \( a \equiv b \). Such a type theory was introduced by Martin-Löf [1984] and forms the basis of the NuPRL system [Constable, 1986]. Note that we have left this rule out of our type theory: adding it would make type checking undecidable. Indeed, to check whether \( x \) and \( y \) are definitionally equal, the type checker may have to generate, out of thin air, a proof \( p \) that \( x \) and \( y \) are propositional equal. Our logic is too powerful for propositional equality to be decidable.

Nevertheless, this section is developed in the extensional model of an LCCC. In this categorical model, equal objects are freely exchangeable, leading to less bureaucratic proofs. Nonetheless, all the computational objects presented in this thesis have been implemented in Agda, an intensional type theory.

5.3.1. Containers

Model: Chapter5.Container
Chapter5.Container.Examples

Containers provide a categorical model for indexed families [Dybjer, 1991] by capturing the signature of inductive definitions. A container is a small, extensional object that admits a rich algebra. Containers are then interpreted as strong functors – the polynomial functors – between slices of SET. Let us recall the definition of containers, and their interpretation.

5.53 Definition (Container). Let \( I, J : \text{SET} \). \n
100
A container indexed by $I$ and $J$ is a triple

\[
\begin{align*}
\text{Op}: & \; J \to \text{SET} \\
\text{Ar}: & \; \text{Op } j \to \text{SET} \\
\text{Sort}: & \; \text{Ar } op \to I
\end{align*}
\]

where Op stands for “operations”, and Ar stands for “arities”. Such a triple is denoted $\text{Op} \triangleleft \text{Sort} \text{Ar}$. The class of containers indexed by $I$ and $J$ is denoted $\text{ICont } I J$.

**Intuition.** Containers can be understood as multi-sorted signatures. The indices specify the sorts. The set $\text{Op}$ for a given index $j$ specifies the operations available at that sort. The set $\text{Ar}$ for a given operation $op$ specifies the arity of that operation. The function $\text{Sort}$ specifies, for each operation and for each argument of that operation, the sort of the argument.

**5.55 Example** (Natural numbers). Natural numbers are described by the signature functor $X \mapsto 1 + X$. The corresponding container is given by:

\[
\begin{align*}
\text{NatCont} & \triangleq \left\{ \begin{array}{l}
\text{OpNat}(\ast:1) : \text{SET} \\
\text{OpNat} \ast \mapsto 1 + 1 \\
\text{ArNat}(op: \text{OpNat} \ast) : \text{SET} \\
\text{ArNat}(\text{inj}_l \ast) \mapsto 0 \\
\text{ArNat}(\text{inj}_r \ast) \mapsto 1 \\
\text{SortNat}(ar: \text{ArNat } op) : 1 \\
\text{SortNat} \ast \mapsto \ast
\end{array} \right. 
\end{align*}
\]

There are two operations, one to represent the 0 case, the other to represent the successor case, $\text{suc}$. For the arities, none is offered by the operation 0, while the operation $\text{suc}$ offers one. Note that the signature functor is not indexed: the container is therefore indexed by the unit set and the arguments’ sort is trivial.

**5.56 Example** (Lists). The signature functor describing a list of parameter $A$ is $X \mapsto 1 + A \times X$. The container is given by:

\[
\begin{align*}
\text{ListCont}_A & \triangleq \left\{ \begin{array}{l}
\text{OpList}(\ast:1) : \text{SET} \\
\text{OpList} \ast \mapsto 1 + A \\
\text{ArList}(op: \text{OpList} \ast) : \text{SET} \\
\text{ArList}(\text{inj}_l \ast) \mapsto 0 \\
\text{ArList}(\text{inj}_r a) \mapsto 1 \\
\text{SortList}(ar: \text{ArList } op) : 1 \\
\text{SortList} \ast \mapsto \ast
\end{array} \right. 
\end{align*}
\]

Note the similarity with natural numbers. There are $1 + A$ operations, i.e. either the empty list $\text{nil}$ or the list constructor $\text{cons}$ of some $a : A$. The arity of the operation $\text{nil}$ is nil, while the operation $\text{cons}$ has arity one. Again, indices are trivial, for this datatype.
5. A Universe of Inductive Families

is not indexed.

5.57 Example (Vectors). To give an example of an indexed datatype, we consider vectors, i.e., lists indexed by their length. The signature functor of vectors is given by

\[ \{ X_n \mid n \in \mathbb{Nat} \} \mapsto \{ n = 0 \mid n \in \mathbb{Nat} \} + \{ A \times X_{n-1} \mid n \in \mathbb{Nat}, n \neq 0 \} \]

where the empty vector \( \text{nil} \) requires the length \( n \) to be 0, while the vector constructor \( \text{cons} \) must have a length \( n \) of at least one and takes its recursive argument \( X \) at index \( n - 1 \). The container representing this signature is given by:

\[
\text{VecCont}_A \triangleq \begin{cases} 
\text{Op}_\text{Vec} (n: \mathbb{Nat}) : \text{SET} \\
\text{Op}_\text{Vec} 0 \mapsto \mathbb{1} \\
\text{Op}_\text{Vec} \text{(suc} n \text{)} \mapsto A \\
\text{Ar}_\text{Vec} (n: \mathbb{Nat}) (op: \text{Op}_\text{Vec} n) : \text{SET} \\
\text{Ar}_\text{Vec} 0 * \mapsto 0 \\
\text{Ar}_\text{Vec} \text{(suc} n \text{)} a \mapsto 1 \\
\text{Sort}_\text{Vec} (n: \mathbb{Nat}) (op: \text{Op}_\text{Vec} n) (ar: \text{Ar}_\text{Vec} ar) : \mathbb{Nat} \\
\text{Sort}_\text{Vec} \text{(suc} n \text{)} a * \mapsto n 
\end{cases}
\]

At index 0, only the operation \( \text{nil} \) is available while the index \( \text{suc} n \) offers a choice of \( A \) operations. As for lists, the arity of the operation \( \text{nil} \) is nil while the operation \( \text{cons} \) has arity one. It is necessary to compute the sort (i.e., the length of the tail) only when the input index is \( \text{suc} n \), in which case the sort is \( n \).

5.58 Definition (Interpretation of containers). Following the algebraic intuition (¶ 5.54), we interpret a container as, first, a choice (\( \Sigma \)-type) of operation; then, for each (\( \Pi \)-type) arity, a variable \( X \) taken at the sort for that arity:

\[
[(\sigma : \text{ICont}_I J)] \text{Cont} (X : I \to \text{SET}) : J \to \text{SET} \\
[\text{Op} \triangleleft \text{Sort} \text{Ar}] \text{Cont} X \mapsto \lambda j. (op: \text{Op} j) \times ((ar: \text{Ar} op) \to X (\text{Sort} ar))
\]

5.59 Definition (Polynomial functor). A functor \( F \) is called a polynomial functor if it is isomorphic to the interpretation of a container, i.e. there exists \( \text{Op}, \text{Ar}, \) and \( \text{Sort} \) such that

\[
F \cong \lambda j. \lambda X. (op: \text{Op} j) \times ((ar: \text{Ar} op) \to X (\text{Sort} ar))
\]

This terminology is justified by fact that the interpretation computes a multivariate polynomial (in \( J \) variables) defined as an \( \text{Op} \)-indexed sum of monomials \( X \) taken at some exponent \( ar: \text{Ar} op \), or put informally:

\[
[\text{Op} \triangleleft \text{Sort} \text{Ar}] \text{Cont} \{ X_i \mid i \in I \} \mapsto \left\{ \sum_{op \in \text{Op}_j} \prod_{ar \in \text{Ar}_op} X_{\text{Sort} ar} \mid j \in J \right\}
\]

We leave it to the reader to verify that the interpretation of \( \text{NatCont} \) (Example 5.55), \( \text{ListCont} \) (Example 5.56), and \( \text{VecCont} \) (Example 5.57) are indeed equivalent to the signature functors we aimed at representing. With this exercise, one gains a better intuition
of the respective contribution of operation, arity, and sorts to the encoding of signature functors. The corresponding datatypes are obtained by taking the initial algebra of these functors. More generally, the initial algebras of containers are exactly the indexed W-types [Petersson and Synek, 1989, Morris et al., 2009].

5.3. Categorical Semantics of Inductive Families

5.3.2. Descriptions are Equivalent to Containers

MODEL: Chapter5.IDesc.Algebra

(5.61) In this section, we set out to prove the equivalence between our presentation of inductive families in type theory (Chapter 5) and containers.

**5.62 Definition** (Described functor). A $F$ functor is described if it is isomorphic to the interpretation of an indexed description, i.e. there exists $D : \text{func} \ I \ J$ such that $F \cong [D]$.

(5.63) We first prove that described functors are polynomial (Lemma 5.64), and then prove that the class of polynomial functors is included in the class of described functors (Lemma 5.70). From which we conclude with the equivalence (5.71).

**5.64 Lemma.** The class of described functors is included in the class of polynomial functors.

**Proof.** Let $F : \text{SET}^I \rightarrow \text{SET}^J$ be a described functor.

By definition of the class of described functor, $F$ is naturally isomorphic to the interpretation of a description. That is, for every $j : I$, there is a $D : \text{IDesc} \ I$ such that:

$$\lambda X. F X j \cong \lambda X. [D] X$$

We show that $[D]$ is isomorphic to a container, i.e. there exists $\text{Op}_D : \text{SET}$, $\text{Ar}_D : \text{Op}_D \rightarrow \text{SET}$, and $\text{Sort}_D : \text{Ar}_D \text{op} \rightarrow I$ such that $[D] X \cong (\text{op} : \text{Op}_D) \times (\text{ar} : \text{Ar}_D \text{op}) \rightarrow X (\text{Sort}_D \text{ar})$. We proceed by induction over $D$:

**Case** $D = 1$: We have $[1] X \cong 1 \times X^0$, which is clearly polynomial.

**Case** $D = \text{var} i$: We have $[\text{var} i] X \cong 1 \times (X i)^\sharp$, which is clearly polynomial.

**Case** $D = \Sigma S T$: We have $[\Sigma S T] X = (s : S) \times [T s] X$. By induction hypothesis, $[T s] X \cong (x : T s) \times (p : P_{T s} x) \rightarrow X (n_{T s} p)$. Therefore, we have that:

$$[\Sigma S T] X \cong (s : S) \times (x : T s) \times (p : P_{T s} x) \rightarrow X (n_{T s} p)$$

$$\cong (sx : (s : S) \times T s) \times (p : P_{T (\pi_0 sx)} (\pi_1 sx)) \rightarrow X (n_T (\pi_0 sx) p)$$

This last functor is clearly polynomial.

**Case** $D = \Pi S T$: We have $[\Pi S T] X = (s : S) \rightarrow [T s] X$. By induction hypothesis, $[T s] X \cong (x : T s) \times (p : P_{T s} x) \rightarrow X (n_{T s} p)$. Therefore, we have that:

$$[\Pi S T] X \cong (s : S) \rightarrow (x : T s) \times (p : P_{T s} x) \rightarrow X (n_{T s} p)$$

$$\cong (f : (s : S) \rightarrow T s) \times (s : S)((p : P_{T s} (f s)) \rightarrow X (n_{T s} p))$$

$$\cong (f : (s : S) \rightarrow T s) \times (sp : (s : S) \times P_{T s} (f s)) \rightarrow X (n_T (\pi_0 sp) (\pi_1 sp))$$
To prove the other inclusion – that polynomial functors are a subclass of described functors – we rely on an algebraic characterisation of polynomial functors [Gambino and Kock, 2013, Corollary 1.14]. This result states that the class of polynomial functors is the smallest class of functors between slices of \( \text{SET} \) containing the reindexing functors and their adjoints (whose definition is recalled in Remark 5.66), closed under composition and natural isomorphism. We show that described functors include polynomial functors simply by programming these algebraic operations in \( \text{func} \).

5.66 Remark (The reindexing functor and its adjoints). The reindexing functor, and its adjoints form the cornerstone of many categorical models of type theory [Seely, 1983, Jacobs, 2001]. While we shall not delve into the categorical details, let us define them in (extensional) type theory.

As suggested by its name, the reindexing functor \( \Delta \) reindexes a predicate \( P : J \to \text{SET} \) into a predicate over \( I \) along a renaming function \( \sigma : I \to J \):

\[
\Delta_{(\sigma : I \to J)} (P : J \to \text{SET})(I : \text{SET}) : \text{SET}
\]

\[
\Delta_{\sigma} P i \mapsto P(\sigma i)
\]

Seen as a functor from \( \text{SET} / J \) to \( \text{SET} / I \), the reindexing functor \( \Delta_{\sigma} \) has a left adjoint \( \Sigma_{\sigma} \), which generalises \( \Sigma \)-types:

\[
\Sigma_{(\sigma : I \to J)} (P : I \to \text{SET})(j : J) : \text{SET}
\]

\[
\Sigma_{\sigma} P j \mapsto (i : I) \times \sigma i = j \times P i
\]

It also has a right adjoint \( \Pi_{\sigma} \), which generalises \( \Pi \)-types:

\[
\Pi_{(\sigma : I \to J)} (P : I \to \text{SET})(j : J) : \text{SET}
\]

\[
\Pi_{\sigma} P j \mapsto (i : I) \times \sigma i = j \to P i
\]

We thus have the following string of adjunctions

\[ \Sigma_{\sigma} \dashv \Delta_{\sigma} \dashv \Pi_{\sigma} \]

5.67 Remark (Inverse image). A crucial ingredient in the following proof is the inverse image construction. The inverse of a function \( f \) is defined by the following inductive type:

\[
data \quad \begin{array}{l}
[f : A \to B]^{-1} (b : B) : \text{SET} \\
data \text{ where } f^{-1} (b = f a) \equiv \text{inv } (a : A)
\end{array}
\]

Equivalently, it can be defined with a \( \Sigma \)-type:

\[
(f : A \to B)^{-1} (b : B) : \text{SET}
\]

\[
f^{-1} b \mapsto (a : A) \times f a = b
\]
5.3. Categorical Semantics of Inductive Families

5.68 Lemma. Described functors are closed under reindexing and its adjoints.

Proof. We describe the pullback functors and their adjoints by:

\[
\begin{align*}
D\Delta_{(\sigma : A \to B)} &: \text{func } B A \\
D\Delta_{\sigma} &: \mapsto \lambda a. \text{var } (\sigma a)
\end{align*}
\]

\[
\begin{align*}
D\Sigma_{(\sigma : A \to B)} &: \text{func } A B \\
D\Sigma_{\sigma} &: \mapsto \lambda b. \Sigma \sigma^{-1} b \lambda a. \text{var } a
\end{align*}
\]

\[
\begin{align*}
D\Pi_{(\sigma : A \to B)} &: \text{func } A B \\
D\Pi_{\sigma} &: \mapsto \lambda b. \Pi \sigma^{-1} b \lambda a. \text{var } a
\end{align*}
\]

It is straightforward to check that these descriptions are interpreted to the expected operation on slices of \( \text{SET} \), i.e. that we have:

\[[D\Delta_{\sigma}] \cong \Delta_{\sigma} \quad [D\Sigma_{\sigma}] \cong \Sigma_{\sigma} \quad [D\Pi_{\sigma}] \cong \Pi_{\sigma}\]

\[\square\]

5.69 Lemma. Described functors are closed under composition.

Proof. We define composition of descriptions by:

\[
(D : \text{func } B C) \circ_D (E : \text{func } A B) : \text{func } A C
\]

\[D \circ_D E \mapsto \lambda c. \text{compose } (D c) E\]

Which relies on the Kleisli extension:

\[
\begin{align*}
\text{compose } (D : \text{IDesc } B) (E : \text{func } B A) &: \text{IDesc } A \\
\text{compose } (\text{var } b) E &: \mapsto E b \\
\text{compose } 1 E &: \mapsto 1 \\
\text{compose } (A \times B) E &: \mapsto \text{compose } A E \times \text{compose } B E \\
\text{compose } (\sigma \text{ En } T) E &: \mapsto \sigma \text{ En } \lambda e. \text{compose } (\text{switch } T e) E \\
\text{compose } (\Sigma S T) E &: \mapsto \Sigma S \lambda s. \text{compose } (T s) E \\
\text{compose } (\Sigma S T) E &: \mapsto \Sigma S \lambda s. \text{compose } (T s) E
\end{align*}
\]

It is then straightforward to check that this is indeed computing the composition of the functors, i.e. that we have:

\[[D \circ_D E] \cong [D] \circ [E]\]

\[\square\]

5.70 Lemma. The class of polynomial functors is a subclass of described functors.

Proof. Described functors are closed under reindexing, together with its left and right adjoint (Lemma 5.68), and are closed under composition (Lemma 5.69). Described functors are defined up to natural isomorphism (Definition 5.62). We have from Gambino and Kock [2013, Corollary 1.14] that the class of polynomial functors is the least such
5. A Universe of Inductive Families

set. Therefore, the class of polynomial functor is included in the class of described functors.

We conclude with the desired equivalence:

5.71 Proposition. The class of described functors corresponds exactly to the class of polynomial functors.

Proof. By Lemma (5.64) and Lemma (5.70), we have both inclusions.

The benefit of this algebraic approach is its flexibility with respect to the universe definition: for practical purposes, we are likely to introduce new \( \text{IDesc} \) codes. However, the implementation of reindexing and its adjoints will remain unchanged. Only composition would need to be verified. Besides, these operations are useful in practice, so we are bound to implement them anyway. In the rest of this thesis, we shall conflate descriptions, containers, and polynomial functors, silently switching from one to another as we see fit.

5.3.3. An alternative proof

MODEL: \text{Chapter5.Equivalence.ToContainer} \text{Chapter5.Equivalence.ToDesc}

An alternative proof, followed by Morris [2007], consists in translating the codes of the universe directly to containers, and conversely. This less algebraic approach is more constructive. However, to be absolutely formal, it calls for proving some rather painful (extensional) equalities. While the proofs are laborious, the translation itself is not devoid of interest. In particular, it gives an intuition of descriptions in terms of operation, arity, and sorts. This slightly more abstract understanding of our universe will be useful in this thesis, and is useful in general when reasoning about datatypes.

5.74 Definition (From descriptions to containers). We formalise the translation in Figure 5.4, mapping descriptions to containers. The message to take away from that translation is which code contributes to which part of the container, \textit{i.e.} operation, arity, and/or sorts. Crucially, the 1 and \( \Sigma \) codes contribute only to the operation. The \text{var} and \( \Pi \) codes, on the other hand, contribute to the arity. Finally, the \text{var} code is singly defining the sorts.

5.75 Definition (From containers to descriptions). The inverse translation is otherwise trivial and given here for the sake of completeness:

\[
\begin{align*}
((\sigma: \text{ICont} I J))^{-1} & : \text{func} I J \\
\langle \text{Op} \triangleleft \text{Sort Ar} \rangle^{-1} & \mapsto \Sigma \text{Op} \lambda \text{op}. \Pi (\text{Ar op}) \lambda \text{ar}. \text{var} (\text{Sort ar})
\end{align*}
\]
\[ \left( D : \text{func} \ I \ J \right) : \text{ICont} \ I \ J \]

\[ \langle D \rangle \mapsto \lambda j. \text{Op}_{\text{func}}(D \ j) \triangleleft \lambda \ j. \text{Sort}_{\text{func}}(D \ j) \lambda j. \text{Ar}_{\text{func}}(D \ j) \]

where

\[
\begin{align*}
\text{Op}_{\text{func}}(D : \text{IDesc} \ I) & : \text{SET} \\
\text{Op}_{\text{func}} \ var \ i & \mapsto 1 \\
\text{Op}_{\text{func}} \ 1 & \mapsto 1 \\
\text{Op}_{\text{func}} \ A \times B & \mapsto \text{Op}_{\text{func}} A \times \text{Op}_{\text{func}} B \\
\text{Op}_{\text{func}} \ \sigma \ E \ T & \mapsto (e : \text{Enum} T \ E) \times \text{Op}_{\text{func}} (\text{switch} T e) \\
\text{Op}_{\text{func}} \ \Pi \ S \ T & \mapsto (s : S) \rightarrow \text{Op}_{\text{func}} (T \ s) \\
\text{Op}_{\text{func}} \ \Sigma \ S \ T & \mapsto (s : S) \times \text{Op}_{\text{func}} (T \ s)
\end{align*}
\]

\[
\begin{align*}
\text{Ar}_{\text{func}}(D : \text{IDesc} \ I) (\text{op} : \text{Op}_{\text{func}} D) & : \text{SET} \\
\text{Ar}_{\text{func}} \ var \ i & \mapsto * \mapsto 1 \\
\text{Ar}_{\text{func}} \ 1 & \mapsto * \mapsto 0 \\
\text{Ar}_{\text{func}} \ A \times B (a, b) & \mapsto \text{Ar}_{\text{func}} A a + \text{Ar}_{\text{func}} B b \\
\text{Ar}_{\text{func}} \ \sigma \ E \ T (e, t) & \mapsto \text{Ar}_{\text{func}} (\text{switch} T e) t \\
\text{Ar}_{\text{func}} \ \Pi \ S \ T f & \mapsto (s : S) \times \text{Ar}_{\text{func}} (T \ s) (f s) \\
\text{Ar}_{\text{func}} \ \Sigma \ S \ T (s, t) & \mapsto \text{Ar}_{\text{func}} (T \ s) t
\end{align*}
\]

\[
\begin{align*}
\text{Sort}_{\text{func}}(D : \text{IDesc} \ I) (\text{ar} : \text{Ar}_{\text{func}} D \ \text{op}) & : I \\
\text{Sort}_{\text{func}} \ var \ i & \mapsto * \mapsto i \\
\text{Sort}_{\text{func}} \ A \times B (\text{inj}_a \ ar) & \mapsto \text{Sort}_{\text{func}} A ar_a \\
\text{Sort}_{\text{func}} \ A \times B (\text{inj}_b \ ar) & \mapsto \text{Sort}_{\text{func}} B ar_b \\
\text{Sort}_{\text{func}} \ \sigma \ E \ T ar & \mapsto \text{Sort}_{\text{func}} (\text{switch} T (\pi_0 \ sh)) ar \\
\text{Sort}_{\text{func}} \ \Pi \ S \ T (s, ar) & \mapsto \text{Sort}_{\text{func}} (T \ s) ar \\
\text{Sort}_{\text{func}} \ \Sigma \ S \ T ar & \mapsto \text{Sort}_{\text{func}} (T (\pi_0 \ sh)) ar
\end{align*}
\]

Figure 5.4.: From descriptions to containers
5. A Universe of Inductive Families

(5.76) We are left to prove that these translations are indeed inverse of each other: while this proof is extremely tedious to carry formally, it is intuitively straightforward (and otherwise established by Proposition 5.71).
Conclusion

Following the previous chapter on inductive types, we moved to inductive families. We have extended our type theory with a universe of indexed descriptions, $\text{func}$. This presentation subsumes the previous one by capturing inductive families. The universe of indexed descriptions is the primary object of enquiry of this thesis: it gives us a small, intensional presentation of dependent inductive types.

Related work

Traditionally, inductive types are implemented through a syntactic scheme. The notion of inductive definition corresponds to a signature satisfying a positivity condition. In practice, the positivity is verified by a piece of software, the positivity checker. This is the presentation adopted by systems like Coq and Agda. It originates from the work of Paulin-Mohring [1996] on extending the Calculus of Constructions [Coquand and Huet, 1988] with inductive types. The treatment of fixpoint definitions over inductive types was later treated by Giménez [1995] who showed that these definitions could always be justified by reduction to the induction principle. The meta-theory of such type theories with inductive definitions has been worked out by Luo [1994] and Werner [1994], using an untyped presentation of conversion; and by Goguen [1994] and Barras [2013], using a judgmental presentation of conversion.

The internalised presentation of datatypes originate from the work of Martin-Löf [1984], in the guise of wellorderings (W-types), and Petersson and Synek [1989], for their indexed variant. Whilst these original presentations were meant for an extensional type theory, Dybjer [1997] gave a more intensional presentations based on universes, covering various classes of datatypes [Dybjer, 1994, Benke et al., 2003]. Pursuing this approach of enriching the structure of datatypes, Morris [2007] gave an intensional universe with codes for least and greatest fixpoints. Compared to Morris [2007] and certain universes of [Benke et al., 2003], we have purposely taken a step back: while the objective of these authors was to be as expressive as possible, ours has been to provide a minimal – yet practical – platform in which to experiment with bootstrapping (Part III) and ornaments (Part IV). The result of our experiments ought to carry over to these more expressive systems, subject to (potentially substantial) adaptations.
Part III.

Generic Programming

In this third part, we further explore the design space opened by our presentation of inductive types. In Chapter 6, we bootstrap the universe of descriptions in itself: rather than extending type theory with a code for descriptions, we show how this code can be self-described. In the resulting system, generic programs are first-class citizens: we shall give a few examples of such programs.

In Chapter 7, we further bootstrap the theory of inductive types. First, we formalise the elaboration of inductive definitions – the user interface of inductive types – to our universes of datatypes. Second, we demonstrate how functionalities of the type theory – such as specialised induction principles, or domain-specific decision procedures – can be implemented from within type theory. In effect, we demonstrate our ability to reason about the inductive fragment in type theory itself.
6. Bootstrapping Inductive Types

6.1. The Art of Levitation

The art of levitation consists in reflecting parts of our type theory into the inductive fragment, while maintaining the consistency of the whole. We shall gain practice in this art by first reflecting the enumeration code `EnumU` (Definition 2.35). We then present the reflection of the description code `Desc` as the fixpoint of a description. Although a similar operation can be carried on indexed descriptions, we choose to focus on descriptions, for their simplicity. Pedagogically, we adopt a "trial and error" narrative: doing so, we hope to convey the dangers and subtleties of self-description.

6.1.1. Implementing enumerations

MODEL: Chapter6.EnumU

In Section 2.2, we specified the universe of enumerations `EnumU`. The formation and introduction rules were given as:
6. Bootstrapping Inductive Types

\[
\Gamma \vdash \text{VALID} \\
\Gamma \vdash \text{EnumU} : \text{SET}
\]

However, we remark that the \text{nilE} and \text{consE} codes are respectively the \text{nil} and \text{cons} constructors of an ordinary list with elements of type \text{UId}. This datatype can – and ought to – be coded by a description. In fact, we have already given its code by way of our description of lists (Example 4.20). By specialising the parameter to be \text{UId}, we reflect \text{EnumU} as an inductive type in the type theory.

(6.4) Following this intuition, we define:

\[
\text{EnumU} : \text{SET} \\
\text{EnumU} \mapsto \mu \left( \Sigma \text{EnumT} \{ \text{'nilE} \to 1, \text{'consE} \to \Sigma \text{UId} \times \text{var} \times 1 \} \right)
\]

By relying on the constructor expressions (4.67), we can type check \text{nilE} and \text{consE} against \text{EnumU}. These expressions elaborate to the low-level codes:

\[
\Gamma \vdash \text{EnumU} \ni \text{nilE} \text{Chk} ; \text{'nilE} \] \\
\Gamma \vdash \text{EnumU} \ni \text{consE} t E \text{Chk} ; \text{'consE} t E
\]

(6.5) We have thus defined \text{EnumU} as a description. Our code clearly satisfies the specification we promised to realise. We are therefore tempted to substitute the meta-level \text{EnumU} with this definition. However, this raises two questions. First, we must make sure that this move is \textit{sound}: what if our description of enumerations paradoxically depends on itself? Indeed, to list the constructors, we are making use of an enumeration!

Second and this shall help us answer the first question, in order to extend the type theory, we must provide a type-theoretic \textit{term}. So far, we have given a high-level \textit{expression}. Nonetheless, we expect that expression to elaborate to a well-typed term. It is therefore sound to extend the type theory with this term: \text{EnumU} is an admissible type.

\textbf{6.6 Theorem (Admissibility of EnumU).} There exists a term \text{EnumUD} whose fixpoint satisfies the specification of \text{EnumU}, i.e. we have:

\[
\Gamma \vdash \text{VALID} \Rightarrow \Gamma \vdash \mu \text{EnumUD} : \text{SET} \\
\Gamma \vdash \text{VALID} \Rightarrow \Gamma \vdash \text{in ('nilE, *)} : \mu \text{EnumUD} \\
\Gamma \vdash t : \text{UId} \land \Gamma \vdash e : \mu \text{EnumUD} \Rightarrow \Gamma \vdash \text{in ('consE, (t, (e, *)))} : \mu \text{EnumUD}
\]

and these are the only inhabitants of that type, encoding respectively \text{nilE} and \text{consE}. 

114
6.1. The Art of Levitation

**Proof.** Without surprise, we derive the code of \textit{EnumUD} by type checking our (putative) definition of \textit{EnumU} (§ 6.4):

\[ \vdash \text{SET} \ni \text{EnumU} \implies \mu \text{EnumUD} \]

We proceed in two steps. First, we elaborate the collection of constructors and their codes. Second, we elaborate the enumeration of constructors. We elaborate the collection of constructors and its elimination following standard elaboration rules (respectively, § 3.16 and § 3.18). This results in the following (intermediate) judgment:

\[ \vdash \text{consE} \cdot \text{nilE} \cdot (\text{consE} \cdot \text{consE} \cdot \text{nilE}) \ni \text{EnumU} \implies t \]

\[ \vdash \text{SET} \ni \text{EnumU} \implies \mu (\Sigma (\text{EnumT} \cdot t) \cdot (\text{switch}(1, (\Sigma \text{UId} \cdot \text{var} \cdot 1, *))))) \]

Note that the enumeration of constructors relies on the constructors \text{nilE} and \text{consE}, which do not formally exist! However, recall that \textit{EnumT} has type \textit{EnumU} \rightarrow \textit{SET}: the enumeration \text{consE} \cdot \text{nilE} \cdot (\text{consE} \cdot \text{consE} \cdot \text{nilE}) is thus checked against \textit{EnumU}. We are back to an elaboration problem:

\[ \vdash \text{EnumU} \ni \text{consE} \cdot \text{nilE} \cdot (\text{consE} \cdot \text{consE} \cdot \text{nilE}) \cdot \text{Chk} ? \]

As pointed out earlier (§ 6.4), we have access to the constructor expressions for \textit{EnumU}: the elaboration problem above is thus simply a succession of 3 such elaboration problems. After elaboration, we obtain the following term:

\[ \vdash \text{EnumU} \ni \text{consE} \cdot \text{nilE} \cdot (\text{consE} \cdot \text{consE} \cdot \text{nilE}) \cdot \text{Chk} \]

\[ \text{in (‘consE, (‘nilE, in (‘consE, (‘consE, in (‘nilE, *)))))} \]

For convenience, we underline the tag belonging to the collection. The others come from the elaboration of the \textit{EnumU} constructors. This notational convenience is purely aesthetic, with no semantical implication. Wrapping up, the initial expression (§ 6.4) elaborates to the fully elaborated term:

\[ \vdash \text{SET} \ni \text{EnumU} \implies \mu (\Sigma (\text{EnumT} \cdot (\text{in (‘consE, (‘nilE, in (‘consE, (‘consE, in (‘nilE, *)))))),)}
\]

\[ (\text{switch}(1, (\Sigma \text{UId} \cdot \text{var} \cdot 1, *))))) \]

We therefore take \textit{EnumUD} to be

\[ \text{EnumUD} \triangleq (\Sigma (\text{EnumT} \cdot (\text{in (‘consE, (‘nilE, in (‘consE, (‘consE, in (‘nilE, *)))))),)}
\]

\[ (\text{switch}(1, (\Sigma \text{UId} \cdot \text{var} \cdot 1, *))))) \]

and define \textit{EnumU} by

\[
\begin{align*}
\text{EnumU} & : \text{SET}_t, \\
\text{EnumU} & \mapsto \mu \text{EnumUD}
\end{align*}
\]

115
6. Bootstrapping Inductive Types

<table>
<thead>
<tr>
<th>Object</th>
<th>Role</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnumU,nilE, consE</td>
<td>Build enumerations</td>
<td>Reflected</td>
</tr>
<tr>
<td>π</td>
<td>Build finite function space</td>
<td>Reflected</td>
</tr>
<tr>
<td>EnumT,E, 0, 1+</td>
<td>Index into enumerations</td>
<td>Meta-theory</td>
</tr>
<tr>
<td>switch</td>
<td>Eliminate indices</td>
<td>Meta-theory</td>
</tr>
</tbody>
</table>

Table 6.1.: Status of enumerations

We check that:
- this definition is well-typed, and
- the resulting inductive type has only two constructors: the desired nilE and consE.

Note that under this definition, we can effectively type check our high-level definition of EnumU (§6.4): we (reassuringly) obtain the same low-level term.

Moreover, the $\pi E P$ operator, which computes the finite function space (Definition 2.49), does not need to be provided by the meta-theory either: we can just use the generic induction principle and write it as an ordinary program. Indeed, $E: EnumU$ is just like any datatype now.

6.8 Definition (Reflected $\pi$). We therefore define $\pi$ by induction over $E$:

\[
\begin{align*}
\pi (E: EnumU) (P: EnumT E \rightarrow \text{SET}) : & \text{SET} \\
\pi E & \quad P \quad \leftarrow \text{induction } E \\
\pi \text{ nilE} & \quad P \quad \mapsto 1 \\
\pi (\text{cons}E t E) & \quad P \quad \mapsto P \circ 0 \times \pi E (P \circ 1+) \\
\end{align*}
\]

6.9 Remark (Resulting type theory). Our construction shows that, for a type theory with a universe $\text{Desc}$, we can spare ourselves from extending the meta-theory with EnumU, its constructors nilE and consE, and $\pi$. Indeed, we can simply define them by a description (for EnumU), by induction over the resulting datatype (for $\pi$), and obtain the constructors by elaboration (for nilE and consE).

Our only requirement on the type theory is that it exports the type EnumU and the $\pi$ operator we have defined. The rest of the meta-theory is left unchanged. In particular, the universe decoder EnumT $E$ remains in the meta-theory, together with the primitives 0 and 1+, and the switch operator.

The situation is summarised in Table 6.1. An object is “reflected” when it is either built from a description, or it is implemented within type theory. An object is said to belong to the “meta-theory” if it is introduced by specialised typing and equality rules.
6.1. The Art of Levitation

Overall, without affecting the content of our type theory, we have reflected the code of enumerations, making it an ordinary datatype. We can write generic programs that manipulate them using the generic induction principle of descriptions. Our next step is similar: we are going to condense the entire coding scheme of datatypes onto itself.

6.1.2. Reflecting descriptions

MODEL: Chapter6.Desc

We have introduced the code of descriptions Desc in Section 4.1 (Figure 4.1), presenting its inference rules as a specification. In this section, we set out to fulfil this specification. The key idea is to notice that Desc is itself a datatype, the codes being the constructors of that datatype. We are therefore in the same situation as with EnumU: by describing descriptions themselves, we remove the need for introducing Desc at the meta-level and gain the generic equipment of datatypes for it. In such a system, descriptions are first-class citizens, born equipped with an induction principle.

6.12 Remark (Universe levels). Because we will be living on the edge of a self-referential paradox, we shall be explicit about universe stratification in this section. We therefore write our definitions by annotating them with the universe level they live in. For example, we write

\[
\begin{align*}
T^\ell &: \text{SET}^\ell \\
T^\ell &\mapsto \ldots
\end{align*}
\]

to define the type \( T \) simultaneously at every level of the hierarchy of sets. Similarly, we write

\[
\begin{align*}
t^\ell &: T^\ell \\
t^\ell &\mapsto \ldots
\end{align*}
\]

to inhabit such a type across the entire hierarchy. Thanks to this informal notation, the reader can easily check that our definitions are well stratified.

However, our type theory does not formally support this kind of definition. Agda implements “set polymorphism”, where a similar universal quantification over universe levels is possible. However, the meta-theory behind it is not well-developed. To back up our presentation, we build a stratified model of levitation in Section 6.3. By defining a hierarchy of types by induction-recursion, we can write such definitions simultaneously over all universe levels, hence justifying their validity.

At a high-level, our strategy is as follows:

- We assume the existence of Desc and its fixpoint;
- We describe the Desc datatype in this hypothetical universe;
- We obtain a code DescD;
- Taking the fixpoint of DescD, we obtain a self-described Desc.

All the difficulty stands in the definition of DescD, in which we must give a code describing Desc that does not mention Desc itself.

The first step is to code the choice of constructors. As usual by now, we represent
6. Bootstrapping Inductive Types

the choice of constructors by an enumeration:

\[
\begin{align*}
\text{Desc}_{D_\ell} : & \quad \text{Desc}_{\ell+1} \\
\text{Desc}_{D_\ell} \mapsto \Sigma \text{EnumT} : & \quad \left\{ \begin{array}{c}
'1' \mapsto \{?\} \\
'\text{var}' \mapsto \{?\} \\
'\times' \mapsto \{?\} \\
'\Sigma' \mapsto \{?\} \\
'\Pi' \mapsto \{?\}
\end{array} \right. \\
\end{align*}
\]

We are then left to assign a code to each constructor case.

(6.15) Since the unit (1) and variable (var) cases take no argument, their encoding is trivial:

\[
\begin{align*}
\text{Desc}_{D_\ell} : & \quad \text{Desc}_{\ell+1} \\
\text{Desc}_{D_\ell} \mapsto \Sigma \text{EnumT} : & \quad \left\{ \begin{array}{c}
'1' \mapsto 1 \\
'\text{var}' \mapsto 1 \\
'\times' \mapsto \text{var} \times \text{var} \times 1 \\
'\Sigma' \mapsto \{?\} \\
'\Pi' \mapsto \{?\}
\end{array} \right. \\
\end{align*}
\]

(6.16) The next step is to encode the binary product (\(\times\)), which takes two recursive arguments described by var:

\[
\begin{align*}
\text{Desc}_{D_\ell} : & \quad \text{Desc}_{\ell+1} \\
\text{Desc}_{D_\ell} \mapsto \Sigma \text{EnumT} : & \quad \left\{ \begin{array}{c}
'1' \mapsto 1 \\
'\text{var}' \mapsto 1 \\
'\times' \mapsto \text{var} \times \text{var} \times 1 \\
'\Sigma' \mapsto \{?\} \\
'\Pi' \mapsto \{?\}
\end{array} \right. \\
\end{align*}
\]

6.17 Remark. We are careful to terminate each branch with a 1 code: this way, we will be able to use constructor expressions (4.67) for description codes.

(6.18) Finally, we code the higher-order constructors \(\Sigma\) and \(\Pi\). To do so, we use a \(\Sigma\) code to introduce an \(S\) of type \(\text{SET}_\ell\) and then a \(\Pi\) to code the exponential by \(S\):

\[
\begin{align*}
\text{Desc}_{D_\ell} : & \quad \text{Desc}_{\ell+1} \\
\text{Desc}_{D_\ell} \mapsto \Sigma \text{EnumT} : & \quad \left\{ \begin{array}{c}
'1' \mapsto 1 \\
'\text{var}' \mapsto 1 \\
'\times' \mapsto \text{var} \times \text{var} \times 1 \\
'\Sigma' \mapsto \Sigma \text{SET}_\ell \lambda S. (\Pi S \lambda \cdot \text{var}) \times 1 \\
'\Pi' \mapsto \Sigma \text{SET}_\ell \lambda S. (\Pi S \lambda \cdot \text{var}) \times 1
\end{array} \right. \\
\end{align*}
\]

Following Remark 6.17, we terminate the codes with a 1 code.
At first glance, we have achieved our objective. We have described the codes of the universe of descriptions. We are naturally lead to define $\text{Desc}$ by taking its fixpoint:

$$
\text{Desc}_\ell : \text{SET}_{\ell+1} \\
\text{Desc}_\ell \mapsto \mu \text{Desc}_{D_\ell}
$$

However, as such, this definition is invalid. Indeed, this apparent levitation of the definition in itself relies, just as in the magic trick, on an “invisible cable” holding everything together. We shall now explain where the paradoxical self-reference is, and present one way of avoiding it.

The definition $\text{Desc}_\ell \mapsto \mu \text{Desc}_{D_\ell}$ is circular, but the offensive recursion is concealed by a high-level expression: the elimination of the enumeration of constructors. Elaborating this expression (§ 3.18) and exhibiting the motive $P$ of $\text{switch}$ reveals the issue:

$$
\begin{align*}
\text{Desc}_{D_\ell} & : \text{Desc}_{\ell+1} \\
\text{Desc}_{D_\ell} & \mapsto \Sigma \text{Enum}_T \{ '1 \text{ 'var } \times \Sigma '\Pi \} \\
\left( \text{switch} \left( P := \lambda - . \mu \text{Desc}_{D_{\ell+1}} \right) \right)
\end{align*}
$$

The circularity arises from the fact that we must specify the return type of the general-purpose $\text{switch}$ eliminator: we claim to return a $\mu \text{Desc}_{D_{\ell+1}}$ but this type does not exist yet! Although type propagation allows us to hide this detail, the elaborated term does contain the offensive circularity. Our only way out is to remove any mention of $\text{Desc}$ from its own definition.

6.21 Remark (Attempting to prove the admissibility of $\text{Desc}$). In the previous section, we have shown that it is correct to introduce a self-hosted $\text{Enum}_U$ by giving a low-level term satisfying its specification. It is instructive to attempt a similar construction by elaborating our tentative definition of $\text{Desc}$.

So far, we have elaborated the elimination over the choice of constructors. Let us assume that the choice of constructors elaborates to a term $\text{choice}$ and the tuple of codes elaborates to a term $\text{codes}$:

$$
\begin{align*}
\vdash \text{Enum}_U \ni \{ '1 \text{ 'var } \times \Sigma '\Pi \}^{\text{Chk}} & \sim \text{choice} \\
\vdash \text{Enum}_T \text{choice} \rightarrow \text{Desc} \ni \begin{bmatrix}
1 \\
1 \\
\text{var } \times \text{var } \times 1 \\
\Sigma \text{SET}_T \lambda S. (\Pi S \lambda - . \text{var}) \times 1 \\
\Sigma \text{SET}_T \lambda S. (\Pi S \lambda - . \text{var}) \times 1
\end{bmatrix}^{\text{Chk}} \sim \text{codes}
\end{align*}
$$
We could therefore be tempted to define $\text{DescD}$ with the term:

\[
\text{DescD}_\ell : \text{Desc}_{\ell + 1} \\
\text{DescD}_\ell \mapsto \Sigma (\text{EnumT choice}) (\text{switch} (P := \lambda - . \mu \text{DescD}_{\ell + 1}) \text{codes})
\]

This definition is invalid! The definition of $\text{DescD}$ depends on $\text{DescD}$ itself. Adding this term to our type theory would break strong normalisation and its corollary: decidability of type checking.

Thus, the problem stands solely in the fact that we must supply a return type for $\text{switch}$. There is an easy way around that: since we know what this return type must be, we eliminate the dreaded circularity by specialising $\text{switch}$:

\[
\begin{array}{l}
\text{switch}_D (ps : \pi E (\lambda - . \text{Desc}_\ell)) (x : \text{EnumT E}) : \text{Desc}_\ell \\
\text{switch}_D b 0 \mapsto \pi_0 b \\
\text{switch}_D b (1 + x) \mapsto \text{switch}_D (\pi_1 b) x
\end{array}
\]

The magician’s art rests here, in this extension. Whilst this legerdemain is not the only way to achieve our objectives, it is the one that suggests itself. We shall hint at an alternative in Remark 6.32.

**6.23 Remark** (Intuition). As we shall see formally in Section 6.3, this self-description is only an illusion. In fact, we rely on $\text{Desc}_{\ell + 1} : \text{SET}_{\ell + 2}$ to describe $\text{DescD}_{\ell}$, with the intent of defining the object $\text{Desc}_\ell : \text{SET}_{\ell + 1}$ next. This is the reason why we could not provide a motive to $\text{switch}$: we would need to refer to $\text{Desc}_\ell$ while it is not yet defined. By specialising $\text{switch}$ to $\text{switch}_D$, we free ourselves from the need to justify the existence of a type $\text{Desc}_\ell$ (more precisely, in this case, of a tuple of said type). Freed from that necessity, we can define $\text{DescD}_{\ell}$ while avoiding any mention of $\text{Desc}_\ell$ – i.e. a circularity – altogether.

**6.24 Remark.** The reader might be concerned about the fact that the type of $\text{switch}_D$ mentions $\text{Desc}$. Recall that we must avoid a circular definition of $\text{DescD}$: as long as the term defining $\text{DescD}$ does not require $\text{Desc}$ to pre-exist, we are in a safe territory. A similar situation arose when describing $\text{EnumU}$: the type of $\text{EnumT}$ depends on $\text{EnumU}$ and $\text{EnumT}$ is used to define $\text{EnumU}$.

**6.25 Definition** (Reflecting descriptions). By replacing $\text{switch}$ with a specialised $\text{switch}_D$, we do not need to mention the return type: it is hard-coded in the definition of $\text{switch}_D$. 

120
This definition is therefore non-circular:

\[
\text{Desc}_\ell : \text{Desc}_{\ell+1}
\]

\[
\text{Desc}_\ell \mapsto \Sigma \text{Enum}_T \{ '1' \var ' \times ' \Sigma \Pi \} \\
(\text{switch}_D)
\]

\[
\begin{array}{c}
\{1
\var \\times \var \times 1
\Sigma \text{SET}_\ell \lambda S. (\Pi S \lambda . \var) \times 1
\Sigma \text{SET}_\ell \lambda S. (\Pi S \lambda . \var) \times 1
\end{array}
\]

(6.26) We conceal \text{switch}_D behind a type propagation rule for the finite function spaces of codomain \text{Desc} that we apply with higher priority than \text{switch}:

\[
\Gamma \vdash \text{Enum}_U \ni \{ 'l_0 \ldots 'l_k \} \stackrel{\text{Chk}}{\sim} E \quad \Gamma \vdash \pi E (\lambda . \text{Desc}_\ell) \ni [e_0 \ldots e_k] \stackrel{\text{Chk}}{\sim} ts
\]

(6.27) Thanks to this elaboration rule and the special-purpose \text{switch}_D, we are able to prove the validity of our self-definition. The approach is exactly the same as for enumerations. First, guided by elaboration, we define a term which satisfies the specification of \text{Desc}.

6.28 Theorem (Admissibility of \text{Desc}_D). There exists a term \text{Desc}_D whose fixpoint satisfies the specification of \text{Desc}, i.e. we have:

\[
\begin{align*}
\Gamma \vdash \text{VALID} & \Rightarrow \Gamma \vdash \mu \text{Desc}_D : \text{SET}_{\ell+1} \\
\Gamma \vdash \text{VALID} & \Rightarrow \Gamma \vdash \text{in} ('1,*') : \mu \text{Desc}_D \\
\Gamma \vdash \text{VALID} & \Rightarrow \Gamma \vdash \text{in} ('\var,*') : \mu \text{Desc}_D \\
\Gamma \vdash A : \mu \text{Desc}_D \land \Gamma \vdash B : \mu \text{Desc}_D & \Rightarrow \Gamma \vdash \text{in} ('\times', (A, (B, *))') : \mu \text{Desc}_D \\
\Gamma \vdash S : \text{SET}_\ell \land \Gamma \vdash T : S \rightarrow \mu \text{Desc}_D & \Rightarrow \Gamma \vdash \text{in} ('\Sigma', (S, (T, *))') : \mu \text{Desc}_D \\
\Gamma \vdash S : \text{SET}_\ell \land \Gamma \vdash T : S \rightarrow \mu \text{Desc}_D & \Rightarrow \Gamma \vdash \text{in} ('\text{TI}', (S, (T, *))') : \mu \text{Desc}_D
\end{align*}
\]

and these are the only inhabitants of that type, encoding 1, \var, \times, \Sigma, and \Pi.

Proof. The construction of the term \text{Desc}_D consists simply in unfolding the bidirectional type checking rules. First, we elaborate the enumeration of constructors. Then, we elaborate the elimination of this enumeration, obtaining a \text{switch}_D term. Unlike our previous attempt (Remark 6.21), the resulting term does not mention \text{Desc}_D in the motive since we have removed it. Finally, just as \text{Enum}_U uses its constructors in its own definition, \text{Desc} uses the \text{Desc} constructors to describe itself: these constructors are in-

121
When describing a signature functor, we do not want to use the low-level representations of the constructors at the expression level, thanks to elaboration. Descendants of the constructors (Definition 4.6): we encode them at the term level and we use them in the term language with \( S \) \( \rightarrow \) \( \mu \) \( \rightarrow \) \( DescD_\ell \)

Putting all the pieces together, we obtain a fully-elaborated term:

\[
\begin{align*}
\text{Desc}_\ell & \triangleright 1 \sim_{Chk} (1, *) \\
\text{Desc}_\ell & \triangleright \text{var} \sim_{Chk} (\text{var}, *)
\end{align*}
\]

\[
A : \text{Desc}_\ell; B : \text{Desc}_\ell \triangleright \text{Desc}_\ell \triangleright A \times B \sim_{Chk} (\times, (A, (B, *)))
\]

\[
S : \text{SET}_\ell; T : S \rightarrow \text{Desc}_\ell \triangleright \text{Desc}_\ell \triangleright \Sigma S T \sim_{Chk} (\Sigma, (S, (T, *)))
\]

This term is not meant for human consumption. We choose to put it in full here for two reasons. First, we verify in a glance that \( \text{DescD} \) does not appear on the right-hand side: this definition is not circular. Second, this horrendous term illustrates once more the importance of elaboration: it protects us from these low-level encodings. We can now define \( \text{Desc} \) to be the fixpoint of that low-level term:

\[
\text{Desc}_\ell : \text{SET}_{\ell + 1}
\]

\[
\text{Desc}_\ell \mapsto \mu \text{DescD}_\ell
\]

Under this definition, we can now formally type check our high-level definition of \( \text{DescD} \) (¶ 6.25): we obtain the same term.

---

(6.29) When describing a signature functor, we do not want to use the low-level representations of the \( \text{Desc} \) constructors. Instead, we want to write the standard constructor and have it elaborate to its low-level representation. We obtain this syntactic convenience by bootstrapping: now that \( \text{Desc} \) is but a datatype, we benefit from the elaboration of constructor expressions (¶ 4.67). This is why we did not extend the term language with the \( \text{Desc} \) constructors (Definition 4.6): we encode them at the term level and we use constructors at the expression level, thanks to elaboration.

(6.30) As we did for enumerations (¶ 6.9), let us clarify the status of the universe of descriptions. The kit for defining datatypes is summarised in Table 6.2. For each operation, we describe its role and its status, making clear which components are self-described and which ones are part of the meta-theory. We have reflected the code of descriptions into the universe itself. This self-description is only possible because the least fixpoint \( \mu \)
6.1. The Art of Levitation

<table>
<thead>
<tr>
<th>Object</th>
<th>Role</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desc</td>
<td>Describe signature functors</td>
<td>Reflected</td>
</tr>
<tr>
<td>([ - ])</td>
<td>Interpret descriptions</td>
<td>Meta-theory</td>
</tr>
<tr>
<td>(\mu, \in)</td>
<td>Define, inhabit fixpoints</td>
<td>Meta-theory</td>
</tr>
<tr>
<td>induction, (\square_), (\square)</td>
<td>Induction principle</td>
<td>Meta-theory</td>
</tr>
</tbody>
</table>

Table 6.2.: Summary of constructions on descriptions

(which is defined using the functorial interpretation \([ - ]\)) is part of the meta-theory. Intuitively, through the least fixpoint, our type theory initially contains all the inductive types that can be defined. When “defining” a datatype, we merely exhibit a code. But, essentially, that code already existed on its own. Levitation is achieved by applying this remark to \(\text{Desc}\) itself: by realising that it already implicitly exists, we derive its raw, non-elaborated form (Theorem 6.28) and denote that term “\(\text{Desc}\)”. Induction must be provided as part of the meta-theory for model-theoretic reasons (¶ 4.63).

6.31 Remark (Implementation). As for enumerations, we now consider a type theory in which \(\text{Desc}\) is self-described. In this type theory, we can define other datatypes, but also use the generic equipment of datatypes – such as induction – on \(\text{Desc}\) itself: we natively support generic programming.

Other generic programming systems, such as Generic Haskell [Hinze et al., 2002], PolyP [Jansson and Jeuring, 1997], or Agda with reflection, offer similar operations: they provide a syntactic incarnation of their (internal) representation of datatypes. Indeed, there is an isomorphism between the grammar of datatypes and this internal representation: by providing the user a (meta-level) quote and unquote mechanism, it is possible to cross the isomorphism to and from the reflected grammar and its internal representation. However, reasoning, in type theory, about these reflective operators is hard, or extremely limited. Our presentation collapses this isomorphism, defining the internal representation with its reflected encoding. Consequently, there is no need for a quote/unquote mechanism, and an associated meta-theory. We shall present the isomorphism formally in Section 6.3.

(6.32) **Levitation of indexed descriptions.** There is no conceptual difficulty forbidding the same manoeuvre to be carried on indexed descriptions. There are actually several ways to do it. Let us sketch a few. First, we note that \(\text{IDesc}_I\) is a plain inductive type, parameterised by \(I\). Therefore, we could simply describe \(\text{IDesc}\) with a \(\text{Desc}\) code, as we coded \(\text{EnumU}\). However, this introduces an unnecessary dependency on the universe of descriptions. We would like the universe of indexed descriptions to be standalone.

We could also simply follow the recipe described above, specialising the elimination of enumerations for indexed descriptions to

\[
\text{switch}_{\text{ID}} (ps : \pi E (\lambda - . \text{IDesc}_I I))(x : \text{EnumT} E) : \text{IDesc} \ell I
\]

Having a specialised \(\text{switch}_{\text{ID}}\), we can safely eliminate over the enumeration of con-
6. Bootstrapping Inductive Types

In fact, the only requirement is to avoid any mention of \texttt{IDesc} in the definition of its code and interpretation. Alternatively, we could code the choice of constructors choice with a finite sum \(\sigma\):

\[
\text{IDescD}_\ell \left( I : \text{SET} \right) : \text{IDesc}_{\ell + 1} \ I
\]

Doing so, we hoist \texttt{switch} from the description of \texttt{IDesc} to the interpretation function of \texttt{IDesc}. It is the interpretation function that must then use the specialised \texttt{switchID} elimination principle, as follows:

\[
\begin{align*}
\text{IDescD}_\ell I & \mapsto \sigma \\
1 & \mapsto 1 \\
\text{var} & \mapsto \Sigma \text{ID} \lambda . \text{var} \times \text{var} \\
\times & \mapsto \Sigma \text{EnumU} \lambda E. \pi E \\
\sigma & \mapsto \Sigma \text{SET} \lambda S. (\Pi \text{S} \lambda . \text{var} \times \times 1 \\
\Sigma & \mapsto \Sigma \text{SET} \lambda S. (\Pi \text{S} \lambda . \text{var} \times \times 1 \\
\end{align*}
\]

6.33 Remark. To define the code of the constructor \(\sigma\), we take the liberty of writing \(\pi E\). Indeed, \(\pi E P\) is strictly-positive in the predicate \(P\) (Remark 5.10). As such, it can therefore be described by an inhabitant of \texttt{IDesc} \((\text{EnumU} E)\). Implementing this description is an interesting exercise in programming with the algebraic structure of descriptions. For the sake of completeness, we give its implementation here:

\[
\begin{align*}
\pi \left( E : \text{EnumU} \right) & : \text{IDesc}_\ell \left( \text{EnumU} E \right) \\
\pi \left( \text{nilE} \right) & \mapsto 1 \\
\pi \left( \text{consE} t E \right) & \mapsto \text{var} 0 \times (\text{compose} \ (\pi E) \ D\Delta_{1+} )
\end{align*}
\]

6.2. A Few Generic Constructions

(6.34) We have reflected the code of descriptions in itself. Beyond its pedagogical value, this exercise has several practical outcomes. First, it confirms that the \texttt{Desc} type is just plain data. As any piece of data, it can therefore be inspected and manipulated. Moreover, it is defined by a description. As a consequence, it comes equipped, for free, with an induction principle. Our ability to inspect and program with \texttt{Desc} is not restricted to the meta-language: we have the necessary equipment to program with data, i.e. program over datatypes. \textit{Generic programming is just programming}.
In this section, we illustrate this motto with three examples. In Section 6.2.1, we derive the catamorphism from the induction principle. We thus demonstrate our ability to write generic programs over the universe of datatypes. In Section 6.2.2 and Section 6.2.3, we present, respectively, the free monad and the free indexed monad construction on descriptions and indexed descriptions. This demonstrates our ability to make new datatype from old. Besides, the free monad has some interesting mathematical structure, which, in turn translates into generic operations.

6.2.1. The generic catamorphism

MODEL: Chapter6.Desc.InitialAlgebra

In Section 4.2, we hardwired a dependent induction principle, but sometimes, the catamorphism suffices. We recall the definition of the catamorphism and generically construct it.

6.37 Definition (Catamorphism). Let $F$ be an endofunctor admitting an initial algebra. Let $(X, \alpha : FX \to X)$ be an $F$-algebra.

The catamorphism (or fold) is the unique $F$-algebra morphism $(\alpha) : (\mu F, \text{in} : F \mu F \to \mu F)$ to the algebra $\alpha$. Put explicitly, $(\alpha)$ is the unique morphism making the following diagram commute:

$$
\begin{array}{ccc}
F(\mu F) & \to & FX \\
\downarrow \text{in} & & \downarrow \alpha \\
\mu F & \to & X \\
\end{array}
$$

6.38 Remark. As we have seen in Part II, our functors always admit an initial algebra. However, in intensional type theory, we have only access to weak initial algebras: the catamorphism exists but we cannot prove its uniqueness.

This construction translates quite naturally into type theory. First, we represent functors with a code in Desc. Their initial algebra is then given by $\mu D$. The catamorphism takes an algebra $\alpha : ([D] T \to T)$, an inhabitant of the fixpoint, and returns a $T$:

$$\langle \alpha : [D] T \to T \rangle (x : \mu D) : T$$

Operationally, the catamorphism applies the algebra recursively over the structure of the functor $[D]$. To do so, we proceed by induction over $x$. Since the return type is non-dependent, setting up the induction is an easy step:

$$\langle \alpha : [D] T \to T \rangle (x : \mu D) : T$$

$$\langle \alpha \rangle x \mapsto \text{induction } \{ ? : (xs : [D] (\mu D))(ih : \Box D (\lambda T xs) \to T) \mapsto T \} x$$
6. Bootstrapping Inductive Types

We are then left to implement the inductive step. Given a node \( \text{xs} \) and the induction hypotheses \( \text{ih} \), we must build an element of \( T \). Provided that we know how to make an element of \( J \), \( K \), \( T \), this step will be performed by the algebra \( \alpha \):

\[
\begin{align*}
&\alpha \colon J \rightarrow K \rightarrow T \\
&\text{L} \alpha M \text{x} \mapsto \text{induction} (\lambda \text{xs} \text{ih} . \alpha \{? : J \} T) \text{x}
\end{align*}
\]

To complete the final hole, we have \( \text{xs} \colon J \rightarrow K \rightarrow T \) and \( \text{ih} \colon \square D (\lambda . T) \text{x} \) to hand, and we need a \( J \rightarrow K \rightarrow T \). The argument \( \text{xs} \) has the right shape, but its subcomponents have the wrong type. However, for each such component, \( \text{ih} \) holds the corresponding value in \( T \). We thus implement a function to replace the former with the latter:

\[
\begin{align*}
\text{replace} (D : \text{Desc}) (\text{x} : J \rightarrow K \rightarrow T) (\text{ih} : \square D (\lambda . T) \text{x}) & : J \rightarrow K \rightarrow T \\
\text{replace} \ 1 & \mapsto 1 \\
\text{replace} \ \text{var} & \mapsto (\text{x}, \text{y}) \\
\text{replace} (A \times B) (a, b) & \mapsto (\text{replace} A a x, \text{replace} B b y) \\
\text{replace} (\Sigma S D) (s, d) & \mapsto (s, \text{replace} (D s) d) \\
\text{replace} (\Pi S D) f & \mapsto \lambda s . \text{replace} (D s) (f s)
\end{align*}
\]

6.43 Remark. This pattern-matching is justified by appeal to induction over the code of descriptions. This is made possible by levitation: \( D \) is just like any datatype, we can therefore use the generic induction principle on it.

6.44 Definition (Catamorphism, generically). This concludes our development. Filling the hole with \( \text{replace} D \text{x} \text{ih} \), we have implemented a generic catamorphism function:

\[
\begin{align*}
&\alpha \colon J \rightarrow K \rightarrow T \\
&\text{L} \alpha M \text{x} \mapsto \text{induction} (\lambda \text{xs} \text{ih} . \alpha \{\text{replace} D \text{x} \text{ih}\}) \text{x}
\end{align*}
\]

6.46 Example (Height of a tree). A typical example of a catamorphism is the function computing the maximum height of a binary tree (Example 4.21). The algebra consists in returning a height of 0 on a leaf, while returning the size of the heighest subtree plus one on a node:

\[
\begin{align*}
\text{heightAlg} (\text{ts} : \text{Tree} A) & : \text{Nat} \\
\text{heightAlg} \ [\text{leaf}] & \mapsto 0 \\
\text{heightAlg} \ [\text{node lh rh}] & \mapsto \text{suc} (\text{max lh rh})
\end{align*}
\]

By recursively iterating this algebra over the tree, the catamorphism gives us the desired function:

\[
\begin{align*}
\text{height} (t : \text{Tree} A) & : \text{Nat} \\
\text{height} t & \mapsto \text{heightAlg} t
\end{align*}
\]

6.47 Example (Semantics of typed expressions). In Example 5.48, we described a typed syntax for a language of expressions. Let us now supply the semantics. We implement
an evaluator as a catamorphism:

\[
\begin{align*}
\text{eval}_\mu (tm: \mu \text{ExprD ty}) & : \text{Val ty} \\
\text{eval}_\mu tm & \mapsto (\text{eval}_\mu) tm
\end{align*}
\]

To finish the job, we must supply the algebra that implements a single step of evaluation, given already evaluated subexpressions.

\[
\begin{align*}
\text{eval}_\mu (ty: \text{Ty}) (xs: [[\text{ExprD}] \text{Val}]) & : \text{Val ty} \\
\text{eval}_\mu \text{val} & \mapsto x \\
\text{eval}_\mu \text{cond true } x & \mapsto x \\
\text{eval}_\mu \text{cond false } y & \mapsto y \\
\text{eval}_\mu \text{nat } [\text{plus } x \ y] & \mapsto x + y \\
\text{eval}_\mu \text{bool } [\text{le } x \ y] & \mapsto x \leq y
\end{align*}
\]

Hence, we have a type-safe syntax and a tagless interpreter for our language, in the spirit of Augustsson and Carlsson [1999], with help from the generic catamorphism.

6.2.2. The generic free monad

MODEL: Chapter6.Desc.FreeMonad

We now study a more ambitious generic operation. Given a functor – understood as a signature of operations and represented by a tagged description – we build its free monad, extending the signature with variables and substitution. Our presentation will therefore appeal to functional programmers, being close to the standard construction in Haskell (Remark 6.51). Our objective is then to adapt this construction to indexed descriptions: this shall be the topic of the next section.

Gambino and Hyland [2004, Theorem 11] have proved that every polynomial functor – and therefore any description – has a free monad. Besides, the authors have shown that the free monad of a polynomial functor is itself a container [Gambino and Hyland, 2004, Theorem 16]. Constructively, this suggests that, given a description, we can compute another description that represents its free monad.

6.51 Remark (Free monad in Haskell). To gain some intuition, we recall the free monad construction in Haskell. Given a functor \( f \), the free monad over \( f \) is given by:

```haskell
data FreeMonad f x
  = Ret x
  | Op (f (FreeMonad f x))
```

In effect, the free monad over a signature functor \( f \) consists of \( f \)-terms – constructed from \( \text{Op} \) – and variables – introduced by \( \text{Ret} \). Provided that \( f \) is an instance of \text{Functor}.
we may take \( \text{Ret} \) for \( \text{return} \) and use \( f \)'s \( \text{fmap} \) to define the monadic bind \( >>= \) as substitution:

```haskell
instance Functor f => Monad (FreeMonad f) where
  -- return :: a → FreeMonad f a
  return = Ret
  -- (>>=) :: FreeMonad f a → (a → FreeMonad f b) → FreeMonad f b
  (Ret x) >>= t = t x
  (Op fx) >>= t = Op (fmap (>>= t) fx)
```

6.52 **Intuition.** Examining the definition of the Haskell \( \text{FreeMonad} \) datatype, we note that it is isomorphic to the least fixpoint of the signature functor:

\[
(F^* \ X) \ Z \mapsto X + F Z
\]

Taking the fixpoint of this signature parameterised by \( F \) and \( X \) gives the \( \text{FreeMonad} \) datatype for \( F \) at \( X \). Our task now is to **describe** this signature, for any description.

6.53 **Definition** (Free monad). Our construction works on tagged descriptions (Definition 5.44). Given a set \( X \) of variables, we compute the signature of the free monad:

\[
(D : \text{tagDesc})^* (X : \text{SET}) : \text{tagDesc}
\]

\[
(E, D)^* X \mapsto \begin{cases} \text{ret} : \Sigma X \lambda . 1 \\ E : D \end{cases}
\]

Effectively, we simply add a constructor \( \text{ret} \) that takes an element of \( X \). The collection of constructors \( E \) and their codes \( D \) stay put, leaving the other constructors unchanged.

Unfolding the interpretation of this definition, we check that the following isomorphism holds:

\[
[D^* X] Z \cong X + [D] Z
\]

Taking the fixpoint of \( D^* X \) ties the knot and we obtain the free monad:

\[
(D : \text{tagDesc})^* (X : \text{SET}) : \text{SET}
\]

\[
D^* X \mapsto \mu (D^* X)
\]

We overload the denotation (but not the colour) for the free monad. It is clear from the context whether we refer to the tagged description or its fixpoint.

6.54 **Remark** (Construction on tagged descriptions). The rationale for defining the free monad on tagged descriptions, as opposed to any description, is twofold. Firstly, any description can be turned into a tagged description (Definition 4.23): we do not lose expressive power by focusing on the tagged ones. Secondly, and more importantly, by working on tagged descriptions, we preserve the constructor-oriented presentation of the signature. The (tagged) signature is a collection of operation names, mapped to their code. The free monad extends this collection with one called \( \text{'ret} \), the variable introduced by the free monad. We can thus use constructor expressions for the monadic operations.
6.55 Example (Free monad: lists). The list datatype can be obtained through a free construction. We define the following signature:

\[
\Sigma_{\text{List}} \ (A : \text{SET}) : \text{Desc} \\
\Sigma_{\text{List}} \ A \mapsto \{ \text{cons} : \Sigma \ A \lambda \rightarrow . \var \times 1 \}
\]

We get \( \text{List} \ A \) by considering the free monad on the unit set:

\[ \text{List} \ A \cong (\Sigma_{\text{List}} \ A)^* 1 \]

6.56 Remark. This monadic construction of \( \text{List} \) is not to be confused with the so-called List monad, or non-determinism monad. What we are describing above is how we obtain the \( \text{List} \) datatype from a free monad construction. As we shall see in Example 6.65, we can then use the generic monadic operations to derive concatenation of lists. We exploit here the characterisation of lists as free monoids over an alphabet \( A \).

The non-determinism monad, on the other hand, uses the functor \( \text{List} : \text{SET} \rightarrow \text{SET} \) as a monad on the category \( \text{SET} \). The monadic structure is thus related to the parameter of \( \text{List} \). It is also equationally much richer than the very syntactic free monad.

6.57 Example (Free monad: terminal I/O). Free monads are often used in the context of algebraic theories, as a syntactic representation of a theory (stripped of its equations) [Swierstra, 2008]. For example, one could specify a toy terminal input/output system with the signature:

\[
\Sigma_{\text{IO}} : \text{tagDesc} \\
\Sigma_{\text{IO}} \mapsto \{ \text{put} : \Sigma \text{String} \lambda \rightarrow (\Pi 1 \lambda \rightarrow . \var) \times 1 \\
\text{get} : (\Pi 1 \text{String} \lambda \rightarrow . \var) \times 1 \}
\]

Following standard terminology [Plotkin and Power, 2003], the constructors \( \text{put} \) and \( \text{get} \) are called the operations of the signature. Taking the free monad of this signature, we obtain a \( \text{TermIO} \) monad:

\[ \text{TermIO} \ (X : \text{SET}) : \text{SET} \\
\text{TermIO} \quad X \quad \mapsto \Sigma_{\text{IO}}^* X \]

Equipped with the following generic operations:

\[
\begin{align*}
\text{putString} \ (s : \text{String}) & : \text{TermIO} \ 1 \\
\text{putString} \quad s & \mapsto \text{put} \ s \ \text{ret} \\
\text{getString} & : \text{TermIO} \ \text{String} \\
\text{getString} & \mapsto \text{get} \ \text{ret}
\end{align*}
\]

(6.58) We now go on a foray in the categorical structure that comes with the free monad. As traditional in the programming community, we shall work with the Kleisli category. Our presentation is therefore similar to the code presented in Remark 6.51. Let us recall first the definition of the Kleisli category.
6. Bootstrapping Inductive Types

6.59 Definition (Kleisli category). Let \((T, \eta, \mu)\) be a monad over \(C\).

The Kleisli category of \(T\) is the category \(C_T\) whose objects are objects of \(C\) and morphisms from \(C\) to \(D\) in \(C_T\) are morphisms of from \(C\) to \(TD\) in \(C\). In the programming community, the identity is called \text{return}, while the (postfix) extension operator (which defines composition) is called \text{bind}, denoted \(\gg\gg\), of type \(TA \to (A \to TB) \to TB\).

We construct the Kleisli category of the free monad \(D^*\) by defining the identity (Definition 6.61) and extension operator (Definition 6.64).

6.61 Definition (Identity). The identity trivially consists in returning the ‘ret constructor:

\[
\text{return} \quad (x : X) : D^* X \\
\text{return} \quad x \mapsto \text{ret} x
\]

Intuitively, \text{return} introduces variables.

6.62 Meanwhile, the postfix extension operator \(\text{subst} \ mx \ \sigma\) corresponds to the substitution of the variables in \(mx\) – introduced by ‘ret – with the corresponding term given by \(\sigma\). We generically implement this substitution using the catamorphism. Let us write the type, and start filling in the blanks:

\[
\text{subst} \ \sigma \ (\text{apply} \ \sigma) \ mx \\
\text{subst} \ \sigma \ mx \mapsto \langle \{? : \{xs : [D^* X] (D^* Y) \to D^* Y\}\} \ mx
\]

6.63 We are left with implementing the algebra of the catamorphism. Its role is to substitute [‘ret \(x\)] with \(\sigma \ x\). This corresponds to the following definition:

\[
\text{apply} \ (\sigma : X \to D^* Y) \ (xs : [D^* X] (D^* Y)) : D^* Y \\
\text{apply} \ \sigma \ [\text{ret} \ x] \quad \mapsto \quad \sigma \ x \\
\text{apply} \ \sigma \ (\text{in} (\sigma, xs)) \quad \mapsto \quad \text{in} (\sigma, xs)
\]

6.64 Definition (Extension operator and Composition). We can therefore complete the hole left in \(\$\) 6.62 and obtain the extension operator:

\[
\text{subst} \ (\sigma : X \to D^* Y) \ (mx : D^* X) : D^* Y \\
\text{subst} \ \sigma \ mx \mapsto \langle \text{apply} \ \sigma \rangle \ mx
\]

And thus composition in the Kleisli category:

\[
\rho : Y \to D^* Z \circ (\sigma : X \to D^* Y) : X \to D^* Z \\
\rho \circ \sigma \mapsto \text{subst} \ (\sigma) \circ \sigma
\]

6.65 Example (Free monad: lists). Recall from Example 6.55 that a list can be presented as the free monad of a single operation, ‘cons. In this setting, the nil constructor is but the variable case of the free monad. The extension operator corresponds to substitution of these variables. We can therefore derive the concatenation of two lists \(xs\) and \(ys\) by
“substituting” \(ys\) for the \texttt{nil} constructor of \(xs\):

\[
(xs: \text{List } A) \mapsto (ys: \text{List } A): \text{List } A
\]

\[
xs \mapsto ys \mapsto \text{subut } xs \lambda * . ys
\]

6.66 Example (Free monad: terminal I/O). In Example 6.57, we have defined a domain-specific language that captures terminal inputs/outputs. In the \texttt{TermIO} monad, we can denote the program that reads two strings and prints their concatenation:

\[
\begin{align*}
\text{example} &: \text{TermIO 1} \\
\text{example} \mapsto \text{getString} \mapsto= \lambda s_1. \\
\text{getString} \mapsto= \lambda s_2. \\
\text{putString} (s_1 \mapsto s_2)
\end{align*}
\]

Here, \(\mapsto\) denotes the postfix extension operator, \textit{i.e.} \(mx \mapsto= \sigma \triangleq \text{subut } \sigma mx\).

(6.67) In this section, we have presented the free monad construction over descriptions. Every tagged description can be seen as a signature of operations: we can uniformly add a variable, building a new type from an old one. We have also provided, generically, the monadic structure (\textit{i.e.} substitution of variables) associated with this monad.

6.2.3. Indexed free monad

\textbf{MODEL: Chapter6.IDesc.FreeMonad}

(6.68) The free monad construction can be extended to multi-sorted signatures. In this section, we present this extension, together with the monadic structure accompanying the free construction. \textit{Gambino and Kock [2013, Theorem 4.5]} have shown that the free monad of a polynomial functor is again a polynomial functor. In a sense, what follows is a constructive reading of that theorem.

6.69 Definition ((Indexed) free monad). In Section 6.2.2, we obtained the free monad by adding a variable constructor \texttt{ret}. At a high-level, we do the same here, working again on tagged (indexed) descriptions. In the indexed case, tagged descriptions are segregated in the permanent choices and the indexed choices: we introduce the variable constructor amongst the permanent choices, leaving the index-dependent constructors untouched. This is the only difference. Our definition is thus:

\[
\begin{align*}
(D: \text{tagIDesc} I) &\mapsto (X: I \mapsto \text{SET}) : \text{tagIDesc } I \\
\left\{ i \leftarrow E : T \right\} &\mapsto \left\{ i \leftarrow \text{‘ret} : \hat{\Sigma} (X i) \lambda = . 1 \\
(p \leftarrow F : U) &\mapsto \left\{ i \leftarrow E : T \\
p \leftarrow F : U
\right\}
\end{align*}
\]

(6.69) In line with Example 6.57, we can use the indexed free monad to describe multi-sorted algebraic theories. In what follows, we describe a filesystem interface that guarantees, by indexing, that only legitimate
6. Bootstrapping Inductive Types

operations are performed. In particular, we want to make sure that:
• We do not open a file that is already open;
• We only read from open files;
• We close only open files.

For simplicity, we are going to consider an interface for a single file. We thus do not need to keep track of file handles. The state (open or closed) of that file will be conventionally denoted \( s \). Informally, this API is specified by:

\[
\begin{align*}
\{ s = \text{open} \} & \quad \text{readFile} : \text{String} \quad \{ s = \text{open} \} \\
\{ s = \text{closed} \} & \quad \text{openFile} ( fn : \text{String} ) : 1 \quad \{ s = \text{open} \} \\
\{ s = \text{open} \} & \quad \text{closeFile} : 1 \quad \{ s = \text{closed} \}
\end{align*}
\]

Following McBride, we implement this specification with the following description:

\[
\Sigma_{\text{FS}} : \text{IDesc State}
\]

\[
\Sigma_{\text{FS}} \mapsto \\
\begin{align*}
\text{closed} & \quad \mapsto \text{openFile} : \Sigma \text{String } \lambda . ( \Pi \text{State } \lambda s . \Pi 1 \lambda . \text{var } s ) \times 1 \\
\text{open} & \quad \mapsto \text{readFile} : ( \Pi \text{State } \lambda s . \Pi ( s = \text{open} \times \text{String} ) \lambda . \text{var } s ) \times 1 \\
\text{open} & \quad \mapsto \text{closeFile} : ( \Pi \text{State } \lambda s . \Pi ( s = \text{closed} \times 1 ) \lambda . \text{var } s ) \times 1
\end{align*}
\]

Taking the free monad of that signature, we obtain the \( \text{FS} \) monad

\[
\text{FS} ( X : \text{State} \to \text{SET} ) : \text{State} \to \text{SET} \\
\text{FS} X \mapsto \Sigma_{\text{FS}} X
\]

that is equipped with the generic operations:

\[
\begin{align*}
\text{openFile} ( fn : \text{String} ) & : \text{FS} ( \lambda . 1 ) \text{open} \\
\text{openFile} f n & \mapsto \text{openFile} f n \lambda s . \text{ret} \\
\text{readFile} & : \text{FS} ( \lambda s . s = \text{open} \times \text{String} ) \text{open} \\
\text{readFile} & \mapsto \text{readFile} \lambda s . \text{ret} \\
\text{closeFile} & : \text{FS} ( \lambda s . s = \text{closed} \times 1 ) \text{closed} \\
\text{closeFile} & \mapsto \text{closeFile} \lambda s . \text{ret}
\end{align*}
\]

6.71 Example (Indexed descriptions). An interesting instance of free monad is \( \text{IDesc} \) itself. Indeed, \( \text{var} \) is nothing but the return of a free monad. The remaining constructors form the operations, trivially indexed by \( 1 \). The signature functor is described by:

\[
\text{IDescFreeD} : \text{tagIDesc } \Pi
\]

\[
\text{IDescFreeD} \mapsto \\
\begin{align*}
- & \mapsto 1 : 1 \\
- & \mapsto \times : \text{var } \times \text{var } \times 1 \\
- & \mapsto \sigma : \Sigma \text{EnumU } \lambda E . \pi E \times 1 \\
- & \mapsto \Sigma : \Sigma \text{SET } \lambda s . ( \Pi s \lambda . \text{var } ) \times 1 \\
- & \mapsto \Pi : \Sigma \text{SET } \lambda s . ( \Pi s \lambda . \text{var } \times ) \times 1
\end{align*}
\]

We obtain \( \text{IDesc } 1 \) by applying the free monad construction on this signature, at con-
6.2. A Few Generic Constructions

stant index $I$:

$$\text{IDescD } (I : \text{SET}) : \text{tagIDesc } 1$$

$$\text{IDescD } I \mapsto \text{IDescFreeD } \lambda - . I$$

6.72 Example (Typed expressions). In Example 5.48, we presented a language of arithmetic expressions. However, this language only lets us define and manipulate closed terms. Adding variables, it becomes possible to build and manipulate open terms, that is, terms in a context. We shall get this representation for free thanks to the free monad construction.

An open term is defined with respect to a context, represented by a snoc-list of types:

```plaintext
data Context : SET where
  Context ⊢ []
  Context ⊢ snoc (Γ : Context) (ty : Ty)
```

An environment realises a context, packing a value for each type:

```plaintext
Env (Γ : Context) : SET
Env [] ⊢ 1
Env (snoc Γ S) ⊢ Env Γ × Val S
```

In this setting, we define type variables, $\text{Var}$ by

```plaintext
Var (Γ : Context) (T : Ty) : SET
Var [] T ⊢ 0
Var (snoc Γ S) T ⊢ (Var Γ T) + (S = T)
```

while $\text{Val}$ maps the type to the corresponding host type, $\text{Var}$ indexes a value in the context, obtaining a proof that the types match.

The $\text{lookup}$ function precisely follows this semantics:

```plaintext
lookup (γ : Env Γ) (v : Var Γ T) : Val T
lookup (γ, t) (inj_r refl) ⊢ t
lookup (γ, t) (inj_l x) ⊢ lookup γ x
```

Consequently, taking the free monad of $\text{ExprD}$ by $\text{Var Γ}$, we obtain the language of open terms in a context $Γ$:

```plaintext
openTm (Γ : Context) : Ty → SET
openTm Γ ⊢ ExprD⁺ (Var Γ)
```

In this setting, the language of closed terms corresponds to the free monad assigning an empty set of values to variables:

```plaintext
closeTm : Ty → SET
closeTm ⊢ ExprD⁺ (λ - . 0)
```
6. Bootstrapping Inductive Types

Declaring variables in the empty set amounts to forbidding variables, so closeTm and ExprD describe isomorphic datatypes. Consequently, we can update an old ExprD to a shiny closeTm:

\[
\text{update } (tm:\mu\text{ExprD ty}) : \text{closeTm ty} \\
\text{update } tm \mapsto ((\lambda(tag,tm).\text{in } (1+ tag,tm))) tm
\]

The other direction of the isomorphism is straightforward, the 'ret case being impossible. We are therefore entitled to reuse the eval function to define the semantics of closeTm.

(6.73) As for descriptions, the indexed free monad is equivalent to a Kleisli category that comes with an identity – return – and an extension operator– a substitution operation subst.

6.74 Definition (Identity). The identity is essentially the same as in the non-indexed case (Definition 6.61) and is defined as the variable case:

\[
\text{return } (x:X i) : D^* X i \\
\text{return } x \mapsto \text{ret } x
\]

Intuitively, it lets us introduce variables in the terms generated by the signature D.

6.75 Definition (Extension operator). The definition of the extension in the indexed case follows the non-indexed one (Definition 6.64). Its definition is

\[
\text{substI } (\sigma:X \rightarrow D^* Y) : D^* X \rightarrow D^* Y \\
\text{substI } \sigma \mapsto (\text{applyI } \sigma)
\]

where applyI is defined as follows:

\[
\text{applyI } (\sigma:X \rightarrow D^* Y) (xs:[D^* X] D^* Y i) : D^* Y i \\
\text{applyI } \sigma \ [\text{ret } x] \mapsto \sigma i x \\
\text{applyI } \sigma (c,ys) \mapsto \text{in } (c,ys)
\]

Composition follows straightforwardly.

6.76 Example (Free monad: filesystem interface). In the non-indexed setting, we have used the terminal interface (Example 6.57) to write a simple effectful program (Example 6.66). In the indexed setting, we enforce the underlying protocol of our interface by its type. As a result, our effectful programs obey – by construction – the protocol specified by the signature.

For instance, the following program tries to read the content of a file, with the guar-
6.3. Modelling Levitation

Example (Semantics of typed expressions). Now we would like to give semantics to the language of open terms (Example 6.72). We proceed in two steps: first, we substitute variables by their value in a context; then, we evaluate the resulting closed term. Thanks to eval⇓, the second problem is already solved.

Let us focus on substituting variables from the context. Again, we can subdivide this problem: first, discharging a single variable from the context, with the function discharge; then, applying this function to every variable in the term. The discharge function maps values to themselves, and variables to their value in context:

$$\text{discharge} \quad \gamma \quad v \quad \mapsto \quad \text{val} \left( \text{lookup} \ \gamma \ v \right)$$

We are now left with applying discharge over all variables of the term. We simply have to pass the right arguments to substI, the type guiding us:

$$\text{substExpr} \quad \sigma \quad tm \quad \mapsto \quad \text{substI} \ \sigma \ tm$$

This completes our implementation of the open term’s interpreter:

$$\text{evalTerm}_\gamma \quad (\gamma : \text{Env} \ \Gamma) \quad (tm : \text{openTm} \ \Gamma \ T) \quad : \quad \text{Val} \ T$$
$$\text{evalTerm}_\gamma \quad \gamma \ tm \quad \mapsto \quad \text{eval}_\gamma \ (\text{substExpr} \ (\text{discharge} \ \gamma) \ tm)$$

Without much effort, we have described the syntax of a well-typed language, together with its semantics.

6.3. Modelling Levitation

MODEL: Chapter6.Universe

(6.78) In this chapter, we have seen how to levitate descriptions. At first glance, this operation is dangerously paradoxical. However, our informal annotations shed some light on its stratification: in an infinite hierarchy of types, we code the description at level $\ell$.
6. **Bootstrapping Inductive Types**

using the description at level \( \ell + 1 \). Levitation thus breaks free of paradox through this downward spiral of definitions.

In this section, we formalise this intuition. To do so, we build an inductive-recursive model of an infinite hierarchy of types [Palmgren, 1995] closed under W-types. In this model, we construct the tower of descriptions, parametrically over the level at which the spiral starts. We then use this internalised type theory to, constructively, prove an isomorphism between an “hard-coded” – that is, meta-level – presentation of descriptions and its “levitated” – that is, reflected – presentation. We then explain why Section 6.1 is harmlessly collapsing an isomorphism.

6.79 **Remark** (Type theory as a mathematical system). In this section, we use type theory as a formal framework for (constructive) mathematics. We are going to define three mathematical objects: an infinite hierarchy of types closed under a few set formers, a tower of universes of descriptions, and a tower of levitated universes of descriptions. We then reason about these objects inside type theory, establishing an isomorphism between descriptions. We obtain a mathematical result, under the proviso that one believes in the coherence of type theory equipped with induction-recursion. In this section, we shall make that assumption and rely on inductive-recursive definitions.

To summarise, our “paper” type theory extended with induction-recursion and W-types (Remark 6.80) will be our meta-meta-theory: it provides a mathematical framework in which to study self-description. In this system, we construct a hierarchy of types: this defines a meta-theory (Definition 6.83) in which functions can be instantiated in one go across the hierarchy of types. In that meta-theory, we model a hard-wired definition of descriptions (Definition 6.87): we obtain a model of a type theory with an infinite tower of universes of descriptions, all provided at the meta-level. Finally, we model levitation by describing, using the hard-wired descriptions, an infinite tower of (levitated) descriptions (Definition 6.93). In this final model, we prove that the hard-wired descriptions and their levitated counterparts are isomorphic, in fact equal on the nose. Diagrammatically, our development models the type theoretic construction of the previous section in the following way:

\[
\begin{array}{c}
\vdash \\
\text{(Def. 6.93)} & \mu \text{Desc}_{\ell} \\
\text{(Def. 6.93)} & \cong \\
\text{(Th. 6.96)} & \text{Desc}_{\ell+1} : \text{SET}_{\ell+2} \\
\end{array}
\]

6.80 **Remark** (W-types). W-types [Martin-Löf, 1984, Nordström et al., 1990] were introduced to represent wellorderings in type theory (Figure 6.1). As for (non-indexed) containers, a W-type is specified by a set \( \text{Op} \) – the collection of nodes – and \( \text{Ar} \) – the arity of each node. The introduction rule follows naturally from this intuition of W-types as trees of finite depth. The elimination rule corresponds to transfinite induction over such structure. Note that this corresponds exactly to the least fixpoint of a container (Definition 5.53) indexed by the unit set. Even though W-types are subsumed by
6.3. Modelling Levitation

\[
\Gamma \vdash \text{Op} : \text{SET} \\
\Gamma \vdash \text{Ar} : \text{Op} \rightarrow \text{SET} \\
\Gamma \vdash \mathcal{W} \text{Op} \text{Ar} : \text{SET}
\]

\[\Gamma \vdash \text{op} : \text{Op} \]

\[\Gamma \vdash \text{xs} : \text{Ar} \text{op} \rightarrow \mathcal{W} \text{Op} \text{Ar} \]

\[\Gamma \vdash \sup \text{op} \text{xs} : \mathcal{W} \text{Op} \text{Ar} \]

\[
\begin{align*}
\text{W-elim} & \quad (ih : (op : \text{Op})(xs : \text{Ar} op \rightarrow \mathcal{W} \text{Op} \text{Ar}) \rightarrow ((ar : \text{Ar} op) \rightarrow Q (xs ar)) \rightarrow Q (\sup \text{op} \text{xs})) \\
(xs : \mathcal{W} \text{Op} \text{Ar}) : Q xs
\end{align*}
\]

\[\text{W-elim} \ ih (\sup \text{op} \text{xs}) \mapsto ih \text{op} \text{xs} (\lambda ar. \text{W-elim} \ ih (xs \ ar))\]

Figure 6.1.: W-types

induction-recursion, we bear with this redundancy for the sake of simplicity: dealing
with W-types through their encoding as inductive-recursive types would needlessly
obscure our presentation.

(6.81) Following Palmgren [1995], we first internalise a hierarchy of types. The hierarchy is
built by iterating a next universe operator. This operator is parameterised by a universe
\((U, El)\). It is closed, as usual, under standard type formers – i.e. \(\Sigma\)-types, \(\Pi\) types, and
enumerations – and W-types. It is also closed under names of \(U\) and their decoding.
Such a definition is necessarily inductive-recursive because the domain of, say, the \(\Pi\)-
type is a code (an inductive type) that is then (recursively) interpreted in the type theory
in order to define its codomain: this simultaneous definition of an inductive type and
its recursive interpretation falls beyond the reach of inductive families.

6.82 Definition (Next universe operator). We translate our algebraic intuition into the
following inductive-recursive definition:

\[
\text{data} \ \hat{U} \ [U : \text{SET}][El : U \rightarrow \text{SET}] : \text{SET} \ 
\text{where}
\]

\[
\hat{U} U El \ni U' \\
| El' (T : U) \\
| \Pi' (S : \hat{U} U El) \\
| (T : \hat{El} S \rightarrow \hat{U} U El) \\
| \Sigma' (S : \hat{U} U El) \\
| (T : \hat{El} S \rightarrow \hat{U} U El) \\
| \text{EnumT'} (E : \text{EnumU}) \\
| \mathcal{W}' (S : \hat{U} U El) \\
| (P : \hat{El} S \rightarrow \hat{U} U El)
\]

\[
\hat{El} (T : \hat{U} U El) : \text{SET} \\
\hat{El} U' \rightarrow U \\
\hat{El} (El' T) \rightarrow El T \\
\hat{El} (\Pi' S T) \rightarrow (s : \hat{El} S) \rightarrow \hat{El} (T s) \\
\hat{El} (\Sigma' S T) \rightarrow (s : \hat{El} S) \times \hat{El} (T s) \\
\hat{El} (\text{EnumT'} E) \rightarrow \text{EnumT} E \\
\hat{El} (\mathcal{W}' S P) \rightarrow \mathcal{W} (\hat{El} S) (\lambda s. \hat{El} (P s))
\]

6.83 Definition (Hierarchy of types). We obtain the indexed hierarchy of types by iter-
ating the next universe operator over levels, starting from the empty universe :

\[
\hat{U} (\ell : \text{Level}) : \text{SET} \\
\hat{U} 0 \mapsto 0 \\
\hat{U} (\text{succ} \ell) \mapsto \hat{U} (\hat{U} \ell) \hat{El}
\]

\[
\hat{El} (\ell : \text{Level}) (T : \hat{U} \ell) : \text{SET} \\
\hat{El} 0 T \mapsto 0 \\
\hat{El} (\text{succ} \ell) T \mapsto \hat{El} T
\]

137
6. Bootstrapping Inductive Types

We restrict ourselves to a countable collections of universes, thus defining levels by:

\[
\text{data Level : Set where}
\begin{align*}
\text{Level} & \ni 0 \\
& \mid \text{suc}(\ell : \text{Level})
\end{align*}
\]

Each level of types in the hierarchy thus reflects the types that are lower in the hierarchy – by \(U'\) – and includes them – by \(El'\) – starting the empty universe at the lowest level. It also includes enumerations’ formers – by \(\text{EnumT}'\). Every level is then closed under \(W\)-types, \(\Sigma\)-types and \(\Pi\)-types – thanks to, respectively, \(W'\), \(\Sigma'\), and \(\Pi'\).

Since the lowest-level universe is empty, the actual hierarchy starts at level \(\text{suc } 0\). We therefore define

\[
\tilde{U}^+ (\ell : \text{Level}) : \text{Set} \quad \tilde{U}^+ \ell \mapsto \tilde{U} (\text{suc } \ell)
\]

and work with this hierarchy, whose trivial bottom layer has been removed.

In this model, we can formally write definitions inhabiting simultaneously the entire hierarchy (Remark 6.12). To do so, we simply quantify over \(\ell : \text{Level}\) and build inhabitants of \(\tilde{El}^+ T \ell\). We call such definition a level-parametric definition.

**6.86 Example (Level-parametric Nat)**. We can define natural numbers simultaneously in all universes of the hierarchy by quantifying over all \(\ell : \text{Level}\):

\[
\begin{align*}
\text{Nat} (\ell : \text{Level}) & : \tilde{U}^+ \ell \\
\text{Nat}' & \mapsto W' \left( \text{EnumT}' \left\{ '0', \text{\textquoteleft suc\textquoteright} \right\} \right) \\
& \left\{ '0' \mapsto \text{EnumT}' \{ \} \right. \\
& \left. '\text{\textquoteleft suc\textquoteright} \mapsto \text{EnumT}' \{ '\ast' \} \right\}
\end{align*}
\]

\[
\begin{align*}
0 : \tilde{El}^+ \text{Nat}' \ell & \quad \text{suc} : \tilde{El}^+ (\Pi' \text{Nat}' \lambda \text{\textdash } \text{Nat}') \\
0 & \mapsto \text{sup } '0' -\text{el} \text{im} \\
\text{suc} & \mapsto \lambda n. \text{sup } '\text{\textquoteleft suc\textquoteright} \lambda \ast. n
\end{align*}
\]

**6.87 Definition (Hard-wired descriptions)**. Using \(W\)-types, we define a hard-wired universe of descriptions. We obtain the datatype \(\text{Desc}\) by defining its \(W\)-type at every level:

\[
\begin{align*}
\text{Desc}_M (\ell : \text{Level}) & : \tilde{U}^+ \ell \\
\text{Desc}_M \ell & \mapsto W' \left( \text{EnumT}' \left\{ \begin{array}{c}
1 \\
\var \\
\Sigma \\
\Pi 
\end{array} \right\} \right) \\
& \left\{ 1 \mapsto \text{EnumT}' \{ '\ast' \} \right. \\
& \left. \var \mapsto \text{EnumT}' \{ '\ast' \} \right. \\
& \left. \Sigma \mapsto \Sigma' U' \lambda S. \Pi' (E' S) \lambda \text{\textdash } \text{EnumT}' \{ '\ast' \} \right. \\
& \left. \Pi \mapsto \Sigma' U' \lambda S. \Pi' (E' S) \lambda \text{\textdash } \text{EnumT}' \{ '\ast' \} \right\}
\end{align*}
\]

We automatically obtain the \(\text{Desc}\) constructors by inhabiting this \(W\)-type. We shall skip their definition, which are otherwise unsurprising, and denote them \(1_M, \var_M, \Sigma_M,\) and \(\Pi_M\), where “\(M\)” stands for “Meta”. This definition corresponds to our presentation of \(\text{Desc}\) (Definition 4.6) as a meta-object in Chapter 4. Using the \(W\)-types, we conveniently model the formation and introduction rules of \(\text{Desc}\).

**6.89** Note that the definition we have just written contains some apparently unnecessary
noise in the form of unit types. For convenience, we have underlined this noise. We shall come back to this point in Remark 6.99. In the meantime, we check that the code of descriptions arising from this definition is extensionally equivalent to the one we are interested in modelling.

6.90 Definition (Least fixpoint of \( \text{Desc}_M \)). Given a code in \( \text{Desc}_M \), we can build its least fixpoint, again using a \( W \)-type. The idea is to manually unfold the interpretation function \( I^−K \) (Definition 4.6) in the definition of the fixpoint (Definition 4.38).

We obtain the following code:

\[
\begin{align*}
\mu_M (D: \text{Desc}_M) & : \tilde{U}^+ \ell \\
\mu_M D & \mapsto W' \mu_{\text{Sh}} \mu_{\text{Pos}} \\
\text{where} \\
\mu_{\text{Sh}} (D: \tilde{E}^+ (\text{Desc}_M)) & : \tilde{U}^+ \ell \\
\mu_{\text{Sh}} 1_M & \mapsto \text{EnumT'} \{ 'suc \} \\
\mu_{\text{Sh}} \text{var}_M & \mapsto \text{EnumT'} \{ 'suc \} \\
\mu_{\text{Sh}} (\Sigma M S T) & \mapsto \Sigma' S \lambda s. \mu_{\text{Sh}} (T(s, 'suc)) \\
\mu_{\text{Sh}} (\Pi M S T) & \mapsto \Pi' S \lambda s. \mu_{\text{Sh}} (T(s, 'suc)) \\
\mu_{\text{Pos}} (D: \tilde{E}^+ (\text{Desc}_M)) \ (sh: \tilde{E}^+ (\mu_{\text{Sh}} D)) & : \tilde{U}^+ \ell \\
\mu_{\text{Pos}} 1_M & 'suc \mapsto \text{EnumT'} \{ \} \\
\mu_{\text{Pos}} \text{var}_M & 'suc \mapsto \text{EnumT'} \{ \} \\
\mu_{\text{Pos}} (\Sigma M S T) \ (s, sh) & \mapsto \mu_{\text{Pos}} (T(s, 'suc)) \ (s, sh) \\
\mu_{\text{Pos}} (\Pi M S T) \ f & \mapsto \Sigma' S \lambda s. \mu_{\text{Pos}} (T(s, 'suc)) \ (f, s) \\
\end{align*}
\]

At this stage, we are therefore able to define inductive types using descriptions in our model. To illustrate this possibility, let us describe natural numbers.

6.91 Example (Natural numbers, in \( \text{Desc}_M \)). We define natural numbers exactly as we did in Example 4.18. This time, however, we work in \( \text{Desc}_M \):

\[
\begin{align*}
\text{NatD} \ (\ell: \text{Level}) & : \tilde{E}^+ (\text{Desc}_M) \ell \\
\text{NatD} & \mapsto \Sigma M \text{EnumT'} \{ '0 \mapsto 1_M , 'suc \mapsto \text{var}_M \} \\
\text{Nat} \ (\ell: \text{Level}) & : \tilde{U}^+ \ell \\
\text{Nat} & \mapsto \mu_M \text{NatD} \ell \\
\end{align*}
\]

The constructors of the datatype \( \text{Nat} \) are thus the inhabitants of the \( W \)-type \( \mu_M \text{NatD} \): unfolding its definition, we verify that zero and successor are the (only) inhabitants of this type. We have described natural numbers.

(6.92) This definition has to be contrasted with the definition based on \( W \)-types (Example 6.86). The encoding based on \( W \)-types models a meta-theoretic presentation of natural numbers, i.e. a type theory coming with a notion of natural numbers. The description-based encoding models a presentation inside a \( \text{Desc} \) universe, as we did in Example 4.18. In this case, our only requirement is that the meta-theory provides a universe of descriptions, from which, inside the theory, we build natural numbers.
6. Bootstrapping Inductive Types

At this stage, we have built a model of the type theory presented in Chapter 4. The description grammar \( \text{Desc} \) is modelled by \( \text{Desc}_M \), while its fixpoint \( \mu \) is modelled by \( \mu_M \). We have shown how natural numbers are described in this model. We are now in the position to repeat the levitation exercise from Section 6.1.2: we reflect the datatype \( \text{Desc}_\ell \) using \( \text{Desc}_{\ell+1} \).

6.93 Definition (Levitated descriptions). We levitate the descriptions’ code by writing exactly the same code as in Definition 6.22. The only difference is that we are working in the model, thus we have to use the constructor of \( \text{Desc}_M \) and we quantify over sets using the code \( U' \):

\[
\begin{align*}
\text{Desc}_R (\ell : \text{Level}) & : \tilde{\text{El}}_\ell^+(\text{Desc}_M^{(\text{suc} \ell)}) \ell \\
\text{Desc}_R \ell & \mapsto \Sigma_M \begin{cases} 
\text{'}1 \mapsto 1_M \\
\text{'}\text{var} \mapsto 1_M \\
\text{'}\Sigma \mapsto \Sigma_M U' \lambda S. \Pi_M (\text{El}' S) \lambda - . \text{var}_M \\
\text{'}\Pi \mapsto \Sigma_M U' \lambda S. \Pi_M (\text{El}' S) \lambda - . \text{var}_M 
\end{cases} \\
\text{Desc}_R (\ell : \text{Level}) & : \tilde{U}^+ (\text{suc} \ell) \\
\text{Desc}_R \ell & \mapsto \mu_M \text{Desc}_R \ell
\end{align*}
\]

Diagrammatically, the situation is as follows:

The dashed lines representing the simultaneous instantiation of \( \text{Desc}_M \) over all universe levels, while the arrows represents the implementation of the target \( \text{Desc}_R \) using a code inhabiting its source \( \text{Desc}_M \).

6.96 Theorem. At every level \( \ell \), the datatype \( \text{Desc}_M \ell \) is isomorphic to \( \text{Desc}_R \ell \) and the bijection is an identity.
6.3. Modelling Levitation

Proof. Looking at the constructors of the respective datatype, the isomorphism is trivial:

<table>
<thead>
<tr>
<th>Constructors of $\text{Desc}_M$</th>
<th>(unfolds to)</th>
<th>Constructors of $\text{Desc}_R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1_M \triangleq \text{sup } (1, '\ast') \ 0\text{-elim}$</td>
<td>$\triangleq '1_R$</td>
<td></td>
</tr>
<tr>
<td>$\text{var}_M \triangleq \text{sup } (\text{var}, '\ast') \ 0\text{-elim}$</td>
<td>$\triangleq '\text{var}_R$</td>
<td></td>
</tr>
<tr>
<td>$\Sigma_M S T \triangleq \text{sup } (\Sigma, (S, \lambda \rightarrow '\ast)) \ \lambda (s, '\ast). T s$</td>
<td>$\triangleq '\Sigma_R S T$</td>
<td></td>
</tr>
<tr>
<td>$\Pi_M S T \triangleq \text{sup } (\Pi, (S, \lambda \rightarrow '\ast)) \ \lambda (s, '\ast). T s$</td>
<td>$\triangleq '\Pi_R S T$</td>
<td></td>
</tr>
</tbody>
</table>

That is, up to extensionality, there is a one-to-one mapping between the codes of the respective datatypes. Put otherwise, the W-types, i.e. the respective sets of operations and arities, $\text{Desc}_M$ and $\text{Desc}_R$ are extensionally equal.

(6.97) The levitation trick thus consists in identifying the meta-level descriptions and their reflected counterpart at every level. By collapsing the isomorphism, the self-describing presentation (Section 6.1.2) implements a zig-zag of descriptions spiralling downward:

\[
\begin{align*}
\text{Desc}_R^{\ell+2} & \cong \text{Desc}_M^{\ell+2} : \bar{U}^+ (\ell + 2) \\
\text{Desc}_R^{\ell+1} \cong \text{Desc}_M^{\ell+1} : \bar{U}^+ (\ell + 1) \\
\text{Desc}_R^{\ell} \cong \text{Desc}_M^{\ell} : \bar{U}^+ \ell
\end{align*}
\]

We can therefore dispose entirely of the tower of $\text{Desc}_M$ and present $\text{Desc}_R$ as a self-supporting tower of definitions.

6.98 Remark (Absence of $\text{switch}_D$ in the model). Our model does not capture the $\text{switch}_D$ construction. Instead, we exhibit the isomorphism between meta and reflected descriptions. It is the privilege of an implementation to actually collapse the isomorphism: in our type-theoretic model, we cannot identify distinct objects, even though they are extensionally equal. In our implementation, $\text{switch}_D$ lets us collapse this isomorphism, operationally identifying the reflected descriptions as a pre-existing, meta object – i.e. the meta-level descriptions.

6.99 Remark (Noise in $\text{Desc}_M$ definition). In 6.89, we noticed that the W-type encoding of $\text{Desc}_M$ contained some apparently unnecessary noise, in the form of functions to the unit type. The proof of Theorem 6.96 should shed some light on this decision.

Indeed, the inhabitants of $\text{Desc}_R$ are the result of the interpretation of $\text{DescD}_R$ through $\mu_M$: the resulting W-type contains some unavoidable encoding noise. Knowing this noise, we simply defined $\text{Desc}_M$ in such a way that its interpretation would artificially generate the same noise. This way, the constructors of $\text{Desc}_M$ and the constructors of $\text{Desc}_R$ match on the nose: they are trivially (extensionally) equal. If we had been less...
6. Bootstrapping Inductive Types

careful in defining $\text{Desc}_M$, the isomorphism would still hold. However, establishing the equivalence would have called for more legwork.

6.100 Remark (Set polymorphic models). In a stratified system, the self-encoded nature of $\text{Desc}$ appears only in a set polymorphic sense: the principal type of the encoded description generalises to the type of $\text{Desc}$ itself. We observe this phenomenon in our set polymorphic model of descriptions in Agda, which relies on universe polymorphism, and in our Coq model, which relies on typical ambiguity [Harper and Pollack, 1989]. In those systems, we can again prove an isomorphism between the levitated $\text{Desc}$ and its pre-existing definition. Our presentation takes a step forward and offers to collapse the isomorphism, disposing of any pre-existing definition.
Conclusion

This chapter was a pedagogical exercise in bootstrapping the theory of datatypes. We presented a step-by-step construction of inductive types within type theory itself. On the way, we gave a few examples of the risks of self-description and described the tricks of the trade. This theoretical curiosity has some interesting practical consequences. In such a system, descriptions are just like any other datatype. They come equipped with the standard toolkit for reasoning and computing over them. In particular, we do not need to introduce a special elimination principle for descriptions: we inherit the generic elimination principle of inductive types.

Besides, our presentation contrasts with the generative approaches of Agda and Coq. Our theory of inductive types is closed, defined once and for all by the grammar of descriptions. We can therefore write programs over the structure of all datatypes, the ones that are, and the ones that are to be. We illustrated this possibility by defining some generic datatypes – the free monads – and some generic programs – the catamorphism and the Kleisli categories associated with the free monads. Part IV pushes that exploration of the structure of inductive types to a much wider scale: we shall study a large class of datatype transformations, and develop generic programs for achieving code reuse in type theory.

Our presentation is also of interest to the implementer. Indeed, by condensing the universe of enumerations and descriptions onto themselves, our requirement on the base type theory is minimal: we only need to define the fixpoint of descriptions and its elimination principle. We then obtain the type \( \text{Desc} \) by self-description. This implementation trick simplifies considerably the machinery required to provide inductive types in a generic programming system. We shall explore the bootstrapping possibilities of our design in Chapter 7, in which we reduce inductive definitions to our universe, automatically generate specialised lemmas to manipulate these inductive definitions, and present a “deriving” mechanism akin to the one found in Haskell.

Finally, we have presented a model of levitation. Using induction-recursion, we have built a type-theoretic model of a hierarchy of types closed under W-types. This model lets us formally state stratified definitions, such as the ones used to define \( \text{Desc}_\ell \) from \( \text{Desc}_{\ell+1} \). Thus, we have introduced a hierarchy of hard-coded descriptions and their fixpoints. Within this universe, we have built the levitating descriptions, expressed in term of the hard-coded description one level up. Finally, we have established an isomorphism between the two objects. This justifies the correctness of our apparently self-referential definition: we are merely collapsing an isomorphism.

Related work

Generic programming is a vast topic. We refer our reader to Garcia et al. [2003] for a broad overview of generic programming in various languages. For Haskell alone, there is a myriad of proposals: Hinze et al. [2007] and Rodriguez et al. [2008] provide useful comparative surveys.
Our approach follows the polytypic programming style, as initiated by PolyP [Jansson and Jeuring, 1997]. We build generic functions by induction on signature functors. However, unlike PolyP, we exploit type-level computation to avoid the preprocessing phase: our datatypes are, natively, nothing but codes.

We have the type-indexed datatypes of Generic Haskell [Hinze et al., 2002] for free. From one datatype, we can compute others and equip them with relevant structure: the free monad construction provides one example. Our approach to encoding datatypes as data also sustains generic views [Holdermans et al., 2006], allowing us to rebias the presentation of datatypes conveniently. Tagged descriptions, giving us a sum-of-signmas view, are a natural example.

We do not support polykinded programming [Hinze, 2000b], while it is supported by Generic Haskell. Our descriptions are limited to endofunctors on $\text{Set}^I$. Whilst indexing is known to be sufficient to encode a large class of higher-kindred datatypes [Altenkirch and McBride, 2003], we should rather hope to work in a more compositional style. We are free to write higher-order programs manipulating codes, but is not yet clear whether that is sufficient to deliver abstraction at higher kinds. Similarly, it will be interesting to see whether arity-generic programming [Weirich and Casinghino, 2010] arises just by computing with our codes, or whether a richer abstraction is called for.

The Scrap Your Boilerplate [Lämmel and Peyton Jones, 2003] (SYB) approach to generic programming offers a way to construct generic functions, based on dynamic type-testing via the Typeable type class. SYB cannot compute types from codes, but its dynamic character does allow a more flexible ad hoc approach to generic data traversal. By maintaining the correspondence between codes and types whilst supporting arbitrary inspection of codes, we pursue the same flexibility statically.

Generic programming is not new to dependent types either. Altenkirch and McBride [2003] developed a universe of polykinded types in Lego; Norell [2002] gave a formalisation of polytypic programming in Alfa, a precursor to Agda; Verbruggen et al. [2008, 2010] provided a framework for polytypic programming in the Coq theorem prover. However, these works aim at modelling PolyP or Generic Haskell in a dependently-typed setting for the purpose of proving correctness properties of Haskell code. Our approach is different in that we aim at building a foundation for datatypes, in type theory, for type theory.

Closer to us are the work of Pfeifer and Ruess [1998] and Benke et al. [2003]. These seminal papers introduced the usage of universes for developing generic programs. Our universes share similarities with Benke et al.’s: our universe of descriptions is similar to their universe of iterated induction, and our universe of indexed descriptions is equivalent to their universe of finitary indexed induction. This is not surprising since we share the same source of inspiration, namely induction-recursion.
7. Elaborating Inductive Definitions

(7.1) In this chapter, we give a formal semantics to the syntax of datatypes presented in Section 3.2.2. Its semantics was hinted at by means of examples. We now give a formal specification of its elaboration down to our universes of datatypes. Effectively, we give a translation semantics to inductive definitions.

We first specify the elaboration of inductive types down to the universe of descriptions in Section 7.1. While this system is restricted to strictly-positive types, we take advantage of its simplicity to develop our intuition. We aim at presenting a general methodology for growing a language of datatypes on top of a universe. The choice of a particular universe of datatypes is in large part irrelevant.

In particular, the same ideas are at play on inductive families, whose elaboration we specify in Section 7.2. This system subsumes the previous one, so we should reuse many of the concepts developed for inductive types. The novelty of our syntax is to support computation on indices, as made possible by our universe.

Finally, we consider two potential extensions of the elaboration machinery in Section 7.3. For the proof-assistant implementer, we show how meta-theoretical results on inductive types, such as the work of McBride et al. [2004], can be internalised and formally presented in type theory. For the programmer, we show how a generic deriving mechanism à la Haskell can be implemented from within type theory. Through these examples, we demonstrate the pervasiveness of an elaboration-based approach.

7.2 Remark (Scope of this work). This chapter aims at specifying the elaboration of inductive definitions down to their representations in a universe of inductive types. At the risk of disappointing implementers, we are not describing an implementation. In particular, we shall present elaboration in a relational style, hence conveniently glancing over the operational details. Our goal is to ease the formal study of inductive definitions, hence the choice of this more abstract treatment. Nonetheless, this chapter is not entirely disconnected from implementation. It grew out of our work on the Epigram system, in which Peter Morris implemented a tactic elaborating an earlier form of inductive definition down to descriptions [Brady et al.].

7.1. Inductive Types

(7.3) In this section, we specify the elaboration of inductive types down to our Desc universe. While this universe only captures strictly-positive types, it is a good exercise to understand the general idea governing the elaboration of inductive definitions. Besides, because the syntax is essentially the same, our presentation should be easy to understand for readers familiar with Haskell, Coq, or Agda.
Our syntax for inductive definitions follows the sum-of-products canon:

\[
\text{data } D \overrightarrow{\langle p : P \rangle : \text{SET } \text{where}} \left\{ \begin{array}{l}
D \overrightarrow{p} \ni c_0 (a_0 : T_0) \\
\vdots \\
c_k (a_k : T_k)
\end{array} \right.
\]

The arguments \( \overrightarrow{p} \) are parameters. A \( T_i \) can be recursive, i.e. refer to \( D \overrightarrow{p} \), but only in a strictly-positive position. We require that the parameters are the same in the definition and the recursive arguments. Doing so, we forbid nested types that, short of an impredicative sort, we shall treat as (large) inductive families indexed by a large set, rather than (small) parameterised inductive types [Matthes, 2009].

7.5 Definition (Grammar of inductive definitions). Formally, this notation is captured by the grammar:

\[
\begin{align*}
\langle \text{data} \rangle &::= \text{data } D \left( \overrightarrow{\langle p : (T) \rangle} \right)^* : \langle T \rangle \text{ where } \langle \text{choices} \rangle \\
\langle \text{choices} \rangle &::= D \left( \overrightarrow{p} \right)^* \ni \langle \text{constructor} \rangle \left( \overrightarrow{\langle \text{constructor} \rangle} \right)^* \\
\langle \text{constructor} \rangle &::= \text{con } \langle \text{arguments} \rangle \\
\langle \text{arguments} \rangle &::= \left( \overrightarrow{\langle x : (T) \rangle} \right)^*
\end{align*}
\]

The terminals \( D \) and \( \text{con} \) correspond, respectively, to the datatype name and to the constructor names. Following our earlier convention, the non-terminal \( \langle T \rangle \) ranges over types.

The translation to descriptions follows the structure of inductive definitions. The first level structure consists of the choice of constructors (i.e. the non-terminal \( \langle \text{choices} \rangle \)) and is translated to a \( \Sigma \) code over the finite set of constructors. The second level structure consists of the \( \Sigma \)-telescope of arguments (i.e. the non-terminal \( \langle \text{arguments} \rangle \)): it translates to right-nested \( \Sigma \) codes. When elaborating arguments, we must make sure that the recursive arguments are valid, and translate them to the \( \text{var} \) code.

7.1.1. Description labels

To guide the elaboration of inductive definitions, we extend the type theory with description labels. Their role is akin to programming labels (Section 3.2.1): they guide elaboration and, in particular, help ensure that recursive arguments are correctly elaborated. A description label \( \hat{\langle l \rangle} \) consists of an identifier – the name of the datatype being defined – applied to a list of terms – the parameters of that datatype. It is simply a phantom type around descriptions: it hides a low-level \( \text{Desc} \) code with a high-level
7.1. Inductive Types

**Meta-theory**

\[
\begin{align*}
\Gamma \vdash \text{VALID} \\
\Gamma \vdash \{l\} : \text{SET}_{i+1}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash D : \text{tagDesc} \\
\Gamma \vdash \text{return } D : \{l\} \\
call_{\{l\}}(t : \{l\}) : \text{tagDesc} \\
call_{\{l\}}(\text{return } D) \mapsto D
\end{align*}
\]

Figure 7.1.: Description label

presentation, *i.e.* the name and parameters of the datatype being defined. The elaborative steps are run against a description label: thus, we can spot recursive arguments and check that parameters are preserved across a definition.

**7.8 Definition** (Description label). We extend the type theory with a new syntactic entity, the label

**Syntax**

\[
\begin{align*}
\langle l \rangle ::= D (\langle t \rangle)^* \\
\langle T \rangle ::= \ldots | \langle l \rangle
\end{align*}
\]

We then define a set former \{l\} (Figure 7.1), the description label of the datatype l. We introduce a description label using `return`. This constructor takes a (finite) enumeration of constructors and their respective code. Doing so, we ensure that we only build tagged descriptions. With `call_{\{l\}}`, we eliminate `return`, obtaining a tagged description.

**7.9 Remark.** Labelled descriptions are not strictly necessary in the context of inductive types. In particular, we extend the type theory with a description label that is never actually used as a type. It only guides the elaboration and could therefore simply index our elaboration relations. However, in our treatment of inductive families, we will need the full power of labels. This section helps us get acquainted with the concept of description label.

**7.1.2. Elaborating inductive types**

We present our translation in a top-down manner: from a complete definition, we show how the pieces fit together, giving some intuition for the subsequent translations. We then move on to disassemble and interpret each subcomponent separately. As we progress, the reader should check that the intuition we gave for the whole is indeed valid. Every elaboration step is backed by a soundness property: proving these properties is inherently bottom-up. After having presented our definitions, we can prove the soundness theorem. The proof is technically unsurprising: we shall sketch it at the end.
7. Elaborating Inductive Definitions

of this section.

7.11 Remark (Structure of elaboration judgments). For the purpose of elaboration, we are going to define several judgments. However, they all follow the same pattern:

\[(1) \vdash (2) \overset{(3)}{\rightarrow} (4)\]

In (1), to the left of the entailment symbol (\(\vdash\)), we maintain a (valid) context of typed variables. In (1) and (2), to the left of the arrow, are the inputs of the relation, while outputs are kept to the right, in (4). The inputs in (2) are generally fragments of syntactic definitions. Outputs in (4) are low-level terms in the type theory, generally description codes. The arrow is pronounced “elaborates to”, with the identifier (3) specifying which part of the inductive definition is treated.

7.12 Example. To further ease the understanding of our machinery, we illustrate each step by elaborating binary trees:

\[
data \text{Tree} [A : \text{SET}] : \text{SET} where \\
\text{Tree} A \triangleright \text{leaf} \\
| \quad \text{node} (a : A)(b : \text{Bool} \rightarrow \text{Tree} A)
\]

Note that we (needlessly) use a higher-order argument to code the pair of branches. This is for pedagogical reasons, so as to demonstrate the elaboration of higher-order recursive arguments. We expect this datatype to elaborate to:

\[
\text{Tree} (A : \text{SET}) : \text{SET} \\
\text{Tree} A \mapsto \mu \left( \Sigma \text{EnumT} \{ \text{\'leaf} \mapsto 1 \text{\'node} \mapsto \Sigma A \lambda \lambda . (\Pi \text{Bool} \lambda \lambda . \text{var}) \times 1 \} \right)
\]

7.13 Remark. In our examples, we describe the derivations that lead to the elaboration of the datatype. While convenient for specifying elaboration, displaying the derivation tree justifying our examples has many disadvantages. First, it is space-consuming and rather difficult to read after a few deduction steps. Second, it forces a backward-chaining style: from the desired conclusion, we must derive the subgoals.

Rather than writing proof trees, we write these derivations in the style of Lamport [1995]. This enables a more flat representation of derivations, while allowing a forward-chaining proof style: we can justify a goal by first presenting some hypothesis, and then appeal to a rule that lets us deduce the goal from the hypothesis. Doing so, we put the emphasis on the rules themselves, rather than on a too concrete proof tree. To further reduce space consumption, we shall maintain the context of derivations – the left of the turnstile – at the level of the proof. We indicate these assumptions using the \textit{LET} keyword.

(7.14) Elaboration of an inductive definition (Figure 7.2a.)

\[\Gamma \vdash \text{data } D (p : P) : \text{SET where } choices \overset{\Delta}{\rightarrow} \]

148
7.1. Inductive Types

\[
\begin{align*}
(\Gamma) \vdash \langle \text{data} \rangle & \overset{\text{D}}{\sim} (\Gamma) \\
\Gamma \vdash \text{SET}_1 \ni \langle p : P \rangle & \rightarrow \text{SET}^{\text{Chk}} \smallfrown \langle p : P' \rangle \rightarrow \text{SET} \\
\Gamma ; p : P \vdash D \ni \text{choices} & \overset{\text{Cs}}{\sim} \text{code} \\
\Gamma \vdash \text{data} \; D \; [p : P] : \text{SET} \; \text{where} \; \text{choices} \overset{\text{D}}{\sim} D \\
\Gamma[D \mapsto \lambda \bar{p}. \mu \text{(call} D \bar{p} \text{) code}) : (p : P') \rightarrow \text{SET}] \\
\end{align*}
\]

(a) Elaboration of definition

\[
\begin{align*}
(\Gamma) \vdash \langle l \rangle & \ni \langle \text{choices} \rangle \overset{\text{Cs}}{\sim} \langle t \rangle \\
T \triangleright l & \\
\Gamma \vdash l \ni c_i \overset{\text{Cs}}{\sim} \{t_i \mapsto \text{code}_i\} & \\
\Gamma \vdash \text{EnumU} \ni \{t_i \mapsto \text{Chk}_E\} & \\
\Gamma \vdash \pi E \lambda . \text{Desc} \ni \{t_i \mapsto \text{code}_i\} \overset{\text{Chk}}{\sim} T & \\
\Gamma \vdash l \ni \text{return} (E, T) & \\
\end{align*}
\]

(b) Elaboration of constructor choices

\[
\begin{align*}
(\Gamma) \vdash \langle l \rangle & \ni \langle \text{constructor} \rangle \overset{\text{C}}{\sim} \langle (t) \mapsto (t) \rangle \\
\Gamma \vdash \text{Uld} \ni \langle t \mapsto \text{Chk}_E \rangle & \\
\Gamma \vdash l \ni \text{args} \overset{\text{C}}{\sim} \text{code} & \\
\Gamma \vdash l \ni t \; \text{args} \overset{\text{Cs}}{\sim} \{t' \mapsto \text{code}\} & \\
\end{align*}
\]

(c) Elaboration of a constructor

Figure 7.2.: Elaboration of inductive types
7. Elaborating Inductive Definitions

\[
\langle \Gamma \rangle \vdash \langle l \rangle \ni \langle \text{arguments} \rangle^A(t)
\]

\[
\Gamma \vdash \text{SET} \ni T^\text{Chk} \rightarrow T' \\
\Gamma; x : T' \vdash l \ni \Delta^A \sim \Sigma T' \lambda x. \text{code}_\Delta
\]

\[
\Gamma \vdash l \ni \nabla^R \sim \text{code}_\nabla \\
\Gamma \vdash l \ni \Delta^A \sim \text{code}_\Delta
\]

\[
\Gamma \vdash \text{VALID} \\
\Gamma \vdash l \ni \epsilon^A \sim 1 \quad \text{(ARG-END)}
\]

(d) Elaboration of arguments

\[
\langle \Gamma \rangle \vdash \langle l \rangle \ni \langle T \rangle^R(t)
\]

\[
\Gamma \vdash \text{VALID} \\
\Gamma \vdash l \ni T^R \sim \text{var} \quad \text{(ARG-REC-VAR)}
\]

\[
\Gamma \vdash \text{SET} \ni T^\text{Chk} \rightarrow T' \\
\Gamma; t : T' \vdash l \ni \nabla^R \sim \Pi T' \lambda t. \text{code}_\nabla
\]

\[
\Gamma \vdash \text{SET} \ni T^\text{Chk} \rightarrow T' \\
\Gamma; t : T' \vdash l \ni \nabla^R \sim \Sigma T' \lambda t. \text{code}_\nabla
\]

(e) Elaboration of recursive arguments

\[
\langle T \rangle \triangleright \langle l \rangle
\]

\[
D \triangleright D \quad \text{(MATCH-NAME)} \\
T \triangleright l \quad \text{\textit{\text{\textit{\textit{T}}} \triangleright \textit{\textit{\textit{\textit{l}}} \quad \text{(MATCH-PARAM)}}}}
\]

(f) Matching label

Figure 7.2.: Elaboration of inductive types
This judgment reads as: in context $\Gamma$, the definition \textbf{data} $D \cdot \{p : P\} : \text{SET}$ where \textit{choices} extends the original context to a context $\Delta$ in which $D$ has been defined. To obtain this definition, we first elaborate the parameters – via type checking – and move onto elaborating the choice of constructors – via the judgment $\text{Cs} \Rightarrow$ – introducing a description label in the process.

\textbf{7.15 Example} (Elaborating \textit{Tree}). Applied to our example, we obtain:

\textbf{LET}: $\Gamma$ a valid context

\begin{align*}
(1) & 1. \vdash \text{data} \ \textit{Tree} \ [A : \text{SET}] : \text{SET} \ \textbf{where} \ [\textit{choices}] \\
& \text{where} \\
& \textit{choices} \triangleq \textit{Tree} \ A \ \triangledown \ \textit{leaf} \ | \ \textit{node} \ (a : A)(b : \text{Bool} \rightarrow \textit{Tree} \ A) \\
& \textit{code} \triangleq \text{return} \ \{ \begin{array}{ll}
\text{‘leaf} & \rightarrow 1 \\
\text{‘node} & \rightarrow \Sigma \ A \lambda \_ . (\Pi \text{Bool} \lambda \_ \var) \times 1
\end{array} \} \\
\end{align*}

(7.16) \textbf{Elaboration of constructor choices (Figure 7.2b.)}

\begin{align*}
\Gamma \vdash l \ni \textit{choices} \text{Cs} \Rightarrow \textit{code}
\end{align*}

This judgment reads as: in a context $\Gamma$, the sum of constructors \textit{choices} defining the datatype $l$ elaborates to a description \textit{code}. To elaborate the choice of constructors, we elaborate each individual constructor – via judgment $\text{Cs} \Rightarrow$ – hence obtaining their respective constructor name and code. We then return the finite collection of constructor names and their corresponding codes.

This elaboration step is subject to the soundness property:

\textbf{7.17 Lemma}. If $\Gamma \vdash l \ni \textit{choices} \text{Cs} \Rightarrow \textit{code}$, then $\Gamma \vdash \textit{code} : \{l\}$

\textbf{7.18 Example} (Elaborating \textit{Tree}). Applied to our example, we obtain:

\textbf{LET}: 1. $\Gamma$ a valid context

2. $A : \text{SET}$

\begin{align*}
(1) & 1. \vdash \textit{Tree} \ A \ni [\textit{choices}] \text{Cs} \Rightarrow [\textit{code}] \\
& \text{BY: By Example 7.21 applied to rule (CHOICES)}
\end{align*}

Where \textit{choices} and \textit{code} have been defined above.

(7.19) \textbf{Elaboration of a constructor (Figure 7.2c.)}

\begin{align*}
\Gamma \vdash l \ni c \text{Cs} \Rightarrow [t \mapsto \textit{code}]
\end{align*}

This judgment reads as: in context $\Gamma$, the constructor $c$ defining a datatype $l$ elaborates to a tag $t$, the constructor name, and a description \textit{code}. The role of this elaboration step is twofold. First, we extract the constructor name and elaborate it to a tag – via
Elaborating Inductive Definitions

type checking against Uld. Second, we elaborate the arguments of that constructor –
via judgment \( \Downarrow^A \) – hence obtaining a Desc code terminated by the 1 code. We return the
pair of the tag and the arguments’ code.

This step is subject to the following soundness property:

**7.20 Lemma.** If \( \Gamma \vdash l \supseteq c \overset{\phi}{\Rightarrow} \), then \( \Gamma \vdash t: Uld \quad \Gamma \vdash code: Desc \)

**7.21 Example (Elaborating Tree).** Since our datatype has two constructors, there are two
instances of constructor elaboration.

**LET:** 1. \( \Gamma \) a valid context
2. \( A : \text{SET} \)

**SKETCH:** The first one to elaborate the leaf constructors:

\( \langle 1 \rangle \)

\( \Gamma \vdash \text{Tree } A \supseteq \text{leaf } \overset{\phi}{\Rightarrow} [\text{leaf } \mapsto 1] \)

By: Example 7.24 applied to rule (CONSTRUCTOR)

**SKETCH:** The second one to elaborate the node constructor:

\( \langle 1 \rangle 2. \quad \Gamma \vdash \text{Tree } A \supseteq \text{node } (a : A)(b : \text{Bool} \to \text{Tree } A) \)

\( \overset{\phi}{\Rightarrow} [\text{node } \mapsto \Sigma A \lambda \ldots (\Pi \text{Bool} \lambda \ldots \text{var}) \times 1] \)

By: Example 7.24 applied to rule (CONSTRUCTOR)

(7.22) Elaboration of arguments (Figure 7.2d.)

\[ \Gamma \vdash l \supseteq \text{args} \overset{\phi}{\Rightarrow} \text{code} \]

This judgment reads as: in context \( \Gamma \), a constructor’s arguments \( \text{args} \) defining the
datatype \( l \) elaborate to a description \( \text{code} \). The arguments are interpreted as telescopes
of \( \Sigma \)-types terminated by a unit type. Hence our translation to \( \Sigma \) and \( \times \) codes.

The rules (ARG-SIG) and (ARG-REC) are (purposely) ambiguous: \( T \) could either be a
proper type or a recursive argument. In the first case, this maps to a standard \( \Sigma \) code,
while in the second case, we elaborate the recursive arguments – via judgment \( \overset{\phi}{\Rightarrow} \) – and
continue with a non-dependent product. Once all arguments have been processed, we
conclude by generating the 1 code – via rule (ARG-END).

This translation is subject to the following soundness property:

**7.23 Lemma.** If \( \Gamma \vdash l \supseteq \text{args} \overset{\phi}{\Rightarrow} \text{code} \), then \( \Gamma \vdash \text{code}: \text{Desc} \)

**7.24 Example (Elaborating Tree).**

**LET:** 1. \( \Gamma \) a valid context
2. \( A : \text{SET} \)

**SKETCH:** Elaborating the arguments of the leaf constructor is trivial:

\( \langle 1 \rangle 1. \quad \Gamma \vdash \text{Tree } A \supseteq e \overset{\phi}{\Rightarrow} 1 \)

By: rule (ARG-END)

**SKETCH:** As for the node constructor, we obtain its code by elaborating \( (a : A) \) to a \( \Sigma \)
code, then elaborating the recursive argument, and terminating with the 1 code:

\( \langle 1 \rangle 2. \quad \Gamma \vdash \text{Tree } A \supseteq (a : A)(b : \text{Bool} \to \text{Tree } A) \overset{\phi}{\Rightarrow} \Sigma A \lambda \ldots (\Pi \text{Bool} \lambda \ldots \text{var}) \times 1 \)
7.1. Inductive Types

LET: \( a : A \)

\( \langle 2 \rangle.1. \vdash \text{Tree } A \ni (b : \text{Bool } \rightarrow \text{Tree } A) ^A \bowtie (\Pi \text{Bool } \lambda - . \text{var}) \times 1 \)

\( \langle 3 \rangle.1. \vdash \text{Tree } A \ni e ^\triangle \)

BY: rule (ARG-END)

\( \langle 3 \rangle.2. \text{ Q.E.D.} \)

BY: Example 7.27 and \( \langle 3 \rangle.1 \) applied to rule (ARG-REC)

\( \langle 2 \rangle.2. \text{ Q.E.D.} \)

BY: \( \langle 2 \rangle.1 \) applied to rule (ARG-SIG)

(7.25) Elaboration of recursive arguments (Figure 7.2e.)

\( \Gamma \vdash l \ni \text{rec } \mathcal{R} \)

This judgment reads as: in context \( \Gamma \), a recursive argument \( \text{rec} \) defining the datatype \( l \) elaborates to a description \( \text{code} \). There are three cases. First of all, we spot recursive arguments via rule (ARG-REC-VAR). In this case, we must make sure that the recursive call is valid – via the judgment \( T \triangleright l \triangleright \) – and, if so, we generate a \( \text{var} \) code. We also support constant exponentials of recursive arguments, translating them to the II code – via rule (ARG-REC-EXP). Similarly, we support coefficients of recursive arguments – via rule (ARG-REC-SIG). This translation is subject to the following soundness property:

7.26 Lemma. If \( \Gamma \vdash l \ni \text{rec } ^\mathcal{R} \bowtie \text{code} \), then \( \Gamma \vdash \text{code} : \text{Desc} \)

7.27 Example (Elaborating Tree). The leaf constructor has no recursive argument. There is therefore no elaboration step associated with it. As for the node constructor, we must elaborate its higher-order recursive argument:

LET: 1. \( \Gamma \) a valid context
2. \( A : \text{SET} \)
3. \( a : A \)

\( \langle 1 \rangle.1. \vdash \text{Tree } A \ni (b : \text{Bool } \rightarrow \text{Tree } A) ^\mathcal{R} \bowtie (\Pi \text{Bool } \lambda - . \text{var}) \)

LET: \( - : \text{Bool} \)

\( \langle 2 \rangle.1. \text{Tree } A \triangleright \text{Tree } A \)

BY: definition of \( - \triangleright - \)

\( \langle 2 \rangle.2. \vdash \text{Tree } A \ni \text{Tree } A ^\mathcal{R} \bowtie \text{var} \)

BY: \( \langle 2 \rangle.1 \) and rule (ARG-REC-VAR)

\( \langle 2 \rangle.3. \text{ Q.E.D.} \)

BY: \( \langle 2 \rangle.2 \) and rule (ARG-REC-EXP)

(7.28) Soundness. We can now prove the soundness of the whole translation: the elaboration of a datatype in a valid context \( \Gamma \) returns an extended context \( \Delta \) that is valid.

7.29 Theorem (Soundness of elaboration). If \( \Gamma \vdash \text{data } D (p : P) : \text{SET where choices } ^\mathcal{D} \Delta \), then \( \Delta \vdash \text{VALID} \).

Proof. First, we prove Lemma 7.26 by induction on the structure of recursive arguments. We then prove Lemma 7.23 by induction on the list of arguments. We obtain
Lemma 7.20. Applying this lemma to all constructors, we obtain Lemma 7.17. The soundness theorem follows.

7.30 Remark (Relative importance of matching labels correctly). The reader will have noticed that we did not prove any theorem about the relation $\triangleright$ that matches the labels with the user inputs. Indeed, this relation works purely at a syntactic level, to ensure that the definition entered by the user is consistent. At the type-theoretic level, a “wrong” definition of $\triangleright$ has little effect. For example, if our relation matching labels did not enforce the stability of parameters, the following definitions would elaborate to the same type-theoretic objects as before:

\[
\text{data } \text{Tree} \ [A:\text{SET}] : \text{SET} \ \text{where}
\]
\[
\begin{align*}
\text{Tree } B & \ni \text{leaf} \\
| \quad \text{node} (a:A)(b: \text{Bool} \rightarrow \text{Tree } A)
\end{align*}
\]
\[
\text{data } \text{Tree} \ [A:\text{SET}] : \text{SET} \ \text{where}
\]
\[
\begin{align*}
\text{Tree } A & \ni \text{leaf} \\
| \quad \text{node} (a:A)(b: \text{Bool} \rightarrow \text{Tree } B)
\end{align*}
\]

In effect, the relation $\triangleright$ guarantees that what the user sees is what the elaboration does. But the elaboration would do it irrespectively of a bogus definition of the label matching relation: it is but a syntactic safeguard.

(7.31) **Completeness.** While our soundness theorem gives some hint as to the correctness of our specification, we can obtain a stronger result by proving an equivalence between Coq’s Inductive definitions and the corresponding datatype declaration in our system. This equivalence amounts to proving the equivalence of the associated elimination forms, i.e. Fixpoint in Coq and induction in our system. However, since we do not know of any formal description of elimination principles generated from an Inductive definition, we shall use the simpler presentation given by Giménez [1995]. Because of its simplicity, Giménez’s model does not explicitly support inductive definitions such as mutually-inductive definitions, nested types, or nested fixpoints: as for descriptions (Remark 4.12), we can nonetheless express them through encodings.

7.32 Lemma. For any inductive definition $\text{Ind}(X: \text{SET})(C_0 | \ldots | C_n)$ in Coq, the corresponding inductive definition $\text{data } X : \text{SET} \ \text{where } X \ni C_0 | \cdots | C_n$ in our system elaborates to a code $D$ having an extensionally equal elimination principle.

\[\begin{align*}
\text{data } X : \text{SET} \ \text{where}
\end{align*}\]

**Proof.** To prove this result, we compute the elaboration of a constructor form $C$ (Definition 2.2, [Giménez, 1995]). This merely consists in applying the judgment $\lbrack - \rbrack$: we denote $\lbrack - \rbrack$ the result of this elaboration step. We proceed by induction over the syntax of a constructor form and obtain:

\[
\begin{align*}
\lbrack X \rbrack & \mapsto 1 \\
\lbrack (x:M) \rightarrow C \rbrack & \mapsto \Sigma M \lambda x. \lbrack C \rbrack \\
\lbrack (x:M \rightarrow X) \rightarrow C \rbrack & \mapsto (\Pi M \lambda x. \var) \times \lbrack C \rbrack
\end{align*}
\]

We thus get a translation from Giménez’s recursive type declarations to a code in our
universe:
\[ \text{Ind}(X : \text{SET}) \langle C_0 | \ldots | C_n \rangle \mapsto \Sigma (\text{Fin} n) \{ i \mapsto [C_i] \} \]

Having done that, it is then a straightforward symbol-pushing exercise to prove that Coq’s elimination rules (Section 3.1.1, Giménez [1995]) can be reduced to our generic elimination principle. The crux of the matter is in showing that the minor premises – defined by \( \mathcal{E}_1 \) in that paper – are extensionally equivalent to the inductive step – defined by \( \Box_{D} X \mapsto X \circ \text{in} \) in our system.

(7.33) A corollary of this lemma amounts to the completeness of our syntax of datatypes, \textit{i.e.} for a datatype \( X \), all the functions that can be written over \( X \) in one system are expressible in the other one.

7.34 Theorem (Completeness of elaboration). For an inductive type

\[ \text{Ind}(X : \text{SET}) \langle C_0 | \ldots | C_n \rangle \]

in Coq, any function introduced by a \texttt{Fixpoint} definition over \( X \) admits an extensionally equivalent definition in our system. Conversely, our generic elimination principle is accepted by Coq.

Proof. We must show that any \texttt{Fixpoint} definition can be implemented using our induction principle. To this end, we use Giménez reduction of \texttt{Fixpoint} definitions down to elimination rules. By Lemma 7.32, we have that Coq’s induction principle is equivalent to ours. Conversely, every definition in our system relies on the generic elimination principle, which we have shown equivalent to the one in Coq.

(7.35) Remark. Such a completeness result is only possible because the language of inductive definitions we consider corresponds exactly to Coq’s language. We have an alternative semantics to compare ours with, which allows us to verify our elaboration as we would verify a compiler against a reference semantics. However, in the indexed case, Coq does not support computation on indices. In this case, we escape Coq’s reach and elaboration \textit{defines} the semantics of our datatypes. This is why the soundness result is crucial.

7.2. Inductive Families

In this section, we extend our treatment of inductive definitions to inductive families. To do so, we add support for indices and computation on these indices. The resulting system subsumes the one presented in the previous section. We reuse most notations and rely on the intuition gained through this simpler system. Our syntax is strongly inspired by the one used by Agda and Coq. However, to support computation
7. Elaborating Inductive Definitions

Elaboration

\[
\begin{align*}
\text{⟨data⟩} & \ ::= \text{data} \ D \ \langle [p : \langle T \rangle] \rangle^* (\langle 'i : \langle T \rangle' \rangle)^* : \langle T \rangle \ \text{where} \ \langle \text{patterns} \rangle \\
\text{⟨patterns⟩} & \ ::= (\langle \text{choices} \rangle)^* (\langle \text{match} \rangle)^2 \\
\text{⟨choices⟩} & \ ::= \langle \text{pattern} \rangle \ \exists \langle \text{constructor} \rangle \ (\langle 'i : \langle T \rangle' \rangle)^* \\
\text{⟨match⟩} & \ ::= \langle \text{pattern} \rangle \ \Leftarrow \langle t \rangle \ \{ \langle \text{patterns} \rangle \} \\
\text{⟨pattern⟩} & \ ::= \ D \ (p)^* (\langle t \rangle) | (\langle 'i = \langle t \rangle' \rangle)^* \\
\text{⟨constructor⟩} & \ ::= \ \text{con} \ \langle \text{arguments} \rangle \\
\text{⟨arguments⟩} & \ ::= (\langle 'x : \langle T \rangle' \rangle)^* 
\end{align*}
\]

Figure 7.3.: Grammar of indexed inductive definitions

over indices, we add support for the Epigram-style by ( \Leftarrow \) gadget. Our language of
inductive definitions is therefore more complex, following the skeleton:

\[
\begin{align*}
\text{data} \ D \ \left[ p : \langle P \rangle \right][i : \langle I \rangle] : \text{SET} \ \text{where} \\
& \ D \ \overline{p} \ (i = t_0) \ \exists \ c_{0,0}(:,0,0 : \langle T_0,0 \rangle) \\
& \ \vdots \\
& \ D \ \overline{p} \ k_0 \ \Leftarrow \text{elim} \ i_m \\
& \ \vdots 
\end{align*}
\]

7.37 Definition (Grammar of inductive families). Formally, this notation is captured
by the grammar of Figure 7.3. Following convention, \( \langle t \rangle \) ranges over terms while \( \langle T \rangle \) ranges over types. We artificially segregate parameters from indices, putting param-
eters first. This is not strictly necessary since our syntax syntactically distinguishes
parameters, declared by writing \( \left[ p : \langle P \rangle \right] \), from indices, declared by writing \( \langle i : \langle I \rangle \rangle \). How-
ever, this choice simplifies the syntax and, later, its elaboration. Our definition of the
non-terminal \( \langle \text{patterns} \rangle \) follows the structure of tagged description (Definition 5.44): it
starts with \( \langle \text{choices} \rangle \) – the constructors available at any index – and ends with \( \langle \text{match} \rangle \) –
that introduces computations on indices.

7.38 Remark. This syntax is a superset of Agda’s and Coq’s definitions: by discarding
the non-terminal \( \langle \text{match} \rangle \), we obtain a syntax exactly equivalent to the one they offer.
If we were interested in reasoning about the inductive definitions of these systems,
we could simply remove \( \langle \text{match} \rangle \) from the grammar. With this minor adjustment, our
formalisation of elaboration gives a translation semantics for the inductive definitions
of these mainstream theorem provers.
7.2. Inductive Families

7.2.1. Description labels

In Section 7.1.1, we have introduced the notion of description labels. It was geared to deal with parametric definitions. For inductive families, we must adapt labels to account for indexing. Indices, unlike parameters, can be either unconstrained – denoted \( (i) \) for some index \( i \) – or constrained to some particular value – denoted \( (i = t) \). A description label is now a phantom type around \( \text{IDesc} I \), where \( I \) is the product of the datatype’s indices.

**7.40 Definition** (Indexed) description label. We extend the term language with a syntactic category of (indexed) labels \( \langle l \rangle \):

\[
\langle l \rangle ::= \text{D} (\langle T \rangle)^* \langle \text{indices} \rangle \quad (T) ::= \ldots \mid (l)
\]

\[
\langle \text{indices} \rangle ::= '(i: \langle T \rangle)' \langle \text{indices} \rangle \mid '('(i=t): \langle T \rangle)' \langle \text{indices} \rangle
\]

We then define the set former \( [l] \), the description label of the datatype \( l \) (Figure 7.4). Note that indices are presented in a Church-style: they carry a type. We also introduce a meta-operation \( [\cdot] D \) that computes the (dependent) product of these types. It computes the index of the description from the label:

\[
[D [p_k \ldots (i_i:T_j) \ldots (i_m = t_m:T_m) \ldots] D \mapsto (i_0:T_0) \times \ldots \times (i_m:T_m) \times \ldots]
\]

The type \( [l] \) is inhabited by a return constructor, which takes an enumeration of constructors and their codes. It is then eliminated by call\([l]\) that builds an indexed description.

**7.41 Remark** (Targetting indexed descriptions). For simplicity, our presentation elaborates to a poor man’s tagged indexed description: after a computation rule, we do not keep track of constructors – unlike the actual tagged indexed descriptions (¶ 5.44). Targeting a properly tagged description needlessly complicates our exposition.

7.2.2. Elaborating inductive families

As for inductive types, we shall present the elaboration process in a top-down manner. This presentation shares a few traits with the simpler elaboration of inductive types: we elaborate choices of constructors (Figure 7.5c), followed by individual constructors (Figure 7.5d), and finally process the telescope of arguments (Figure 7.5e).

The presence of indices introduces new steps. We constrain and compute over in-
7. Elaborating Inductive Definitions

<table>
<thead>
<tr>
<th>META-THEORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \Gamma \vdash D : \text{SET}_i ]</td>
</tr>
<tr>
<td>[ \Gamma \vdash \ell : \text{SET}_{i+1} ]</td>
</tr>
<tr>
<td>[ \Gamma \vdash E : \text{EnumU} ]</td>
</tr>
<tr>
<td>[ \Gamma \vdash T : \pi \lambda \ . \text{IDesc} \llbracket D \rrbracket ]</td>
</tr>
<tr>
<td>[ \Gamma \vdash \text{return } ET : \llbracket \ell \rrbracket ]</td>
</tr>
<tr>
<td>[ \text{call} \llbracket \ell \rrbracket (\text{return } ET) \implies (E, T) ]</td>
</tr>
</tbody>
</table>

Figure 7.4.: Description label (indexed)

Induces through a new top-level judgment (Figure 7.5b). Besides, we must translate the constraints to actual equalities (Figure 7.5h) and pass the correct indices when elaborating a recursive argument (Figure 7.5i).

7.43 Example. To give a better intuition of a rather intricate system, we illustrate every judgment with two examples. Our first example is the definition of vectors that relies on constraints to enforce the indexing discipline:

\[
\text{data Vec } [A : \text{SET}](n : \text{Nat}) : \text{SET} \text{ where } \\
\text{Vec } A \quad (n = 0) \ni \text{nil} \\
\text{Vec } A \quad (n = \text{suc } n') \ni \text{cons } (n' : \text{Nat})(a : A)(vs : \text{Vec } A n')
\]

Our second example consists of the alternative definition of vector, where we compute over the index to determine which constructor to offer:

\[
\text{data Vec } [A : \text{SET}](n : \text{Nat}) : \text{SET} \text{ where } \\
\text{Vec } A \quad n \leftarrow \text{Nat-case } n \\
\text{Vec } A \quad 0 \ni \text{nil} \\
\text{Vec } A \quad (\text{suc } m) \ni \text{cons } (a : A)(vs : \text{Vec } A m)
\]

(7.44) Elaboration of inductive families (Figure 7.5a.)

This judgment reads as: in context \( \Gamma \), the definition of datatype \( D \), with parameters \( \vec{p} \), indices \( \vec{i} \), and constructors \( \text{choices} \), elaborates to a context \( \Delta \) in which \( D \) is defined. The elaboration of an inductive definition sets up the environment to trigger the elaboration of the patterns of constructors. To do so, we first elaborate the telescope of parameters and indices – via type checking. We can then translate the patterns – via \( \text{Patts} \) – by elaborating against the label type corresponding to the given inductive type.
7.2. Inductive Families

\[ (\Gamma) \vdash (\text{data}) \overset{D}{\sim} (\Gamma) \]

\[ \Gamma \vdash \text{SET} \ni (p : P)(i : I) \rightarrow \text{SET} \quad \text{Chk} \]

\[ \Gamma ; p : P ; i : I \vdash D [p](i) \ni \text{patts} \overset{\text{D}}{\sim} \text{code} \]

\[ \Gamma \vdash \text{data} \overset{\text{D}}{\sim} (\text{patts} \overset{\text{D}}{\sim} \text{code}) \]

(a) Elaboration of definition

\[ (\Gamma) \vdash \langle l \rangle \ni \langle \text{patterns} \rangle \overset{\text{Patts}}{\sim} \langle t \rangle \]

\[ l \ni pt_i \overset{l_i}{\sim} l_i \]

\[ \Gamma \vdash l_i \ni cs_i \overset{\text{Cs}}{\sim} \{ \{ c_{ij} \} \rightarrow \{ e_{ij} \rightarrow as_{ij} \} \} \]

\[ l \ni pt_{i+1} \overset{l_{i+1}}{\sim} l_{i+1} \]

\[ \Gamma \vdash e \overset{\text{inm}}{\sim} e' \in (\langle x_k : X_k \rangle \rightarrow \langle l_k \rangle) \rightarrow \langle l_{i+1} \rangle \]

\[ \Gamma ; x_k : X_k \vdash l_k \ni p_k \overset{\text{Patts}}{\sim} \text{code}_k \]

(b) Elaboration of patterns

\[ \langle \Gamma \rangle \vdash \langle l \rangle \ni \langle \text{choices} \rangle \overset{\text{Cs}}{\sim} \langle \langle t \rangle \rightarrow \langle t \rangle \rangle \]

\[ \Gamma \vdash l \ni c \overset{\text{Cx}}{\sim} \{ t_i \rightarrow \text{code}_i \} \]

\[ \Gamma \vdash c_0 \ldots c_n \overset{\text{Cs}}{\sim} \{ \{ l_i \} \rightarrow \{ l_i \rightarrow \text{code}_i \} \} \]

(c) Elaboration of choices

Figure 7.5: Elaboration of inductive families
7. Elaborating Inductive Definitions

\[
\begin{align*}
\langle \Gamma \rangle \vdash \langle \text{constructor} \rangle \sim (t) \mapsto \langle t \rangle \\
\Gamma \vdash \text{Uld} \ni c \sim c' \\
\Gamma \vdash \text{lid} \ni \text{code} \sim \text{code} \\
\end{align*}
\]  
\[(\text{CONSTRUCTOR})\]

(d) Elaboration of constructor

\[
\begin{align*}
\Gamma \vdash \langle \text{lid} \rangle \ni (\text{arguments}) \sim \Delta (t)
\end{align*}
\]  

\[(\text{ARG-SIG})\]

\[
\begin{align*}
\Gamma \vdash \text{lid} \ni (x:T) \sim \Delta \sum \lambda x. \text{code} \Delta
\end{align*}
\]  

(AR-G-REC)

\[
\begin{align*}
\Gamma \vdash \text{lid} \ni \text{Eq} \\
\Gamma \vdash \text{lid} \ni \text{code} \Delta
\end{align*}
\]  

(AR-END)

(e) Elaboration of arguments

\[
\begin{align*}
\Gamma \vdash \langle \text{lid} \rangle \ni (\text{arguments}) \sim \Delta (t)
\end{align*}
\]  

\[(\text{ARG-REC-VAR})\]

\[
\begin{align*}
\Gamma \vdash \text{lid} \ni (x:T) \sim \text{var is}
\end{align*}
\]  

(f) Elaboration of recursive arguments

Figure 7.5.: Elaboration of inductive families
7.2. Inductive Families

\{(l) \supseteq (T) \overset{1}{\sim} (l)\}

\[
\begin{array}{c}
\Gamma \vdash D \overset{\sim}{\Downarrow} D \\
\Gamma \vdash T \overset{1}{\sim} l_T \\
l [p] \supseteq T \overset{p}{\sim} l_T [p] \\
l \vdash l \overset{1}{\sim} l_T (i: I) \\
l (i: I) \supseteq T (i = t: I) \overset{1}{\sim} l_T (i = t: I) \\
l (i = t: I) \supseteq T (i = t: I) \overset{1}{\sim} l_T (i = t: I)
\end{array}
\]

(g) Pattern validation

\[
\Gamma \vdash (l) \overset{E}{\sim} (l)
\]

\[
\begin{array}{c}
\Gamma \vdash \text{VALID} (\text{EQ-END}) \\
\Gamma \vdash D \overset{E}{\sim} 1 \\
\Gamma \vdash l \overset{1}{\sim} q \\
\Gamma \vdash l \overset{p}{\sim} q \\
\Gamma \vdash l (i: I) \overset{E}{\sim} q \\
\Gamma \vdash l (i = t: I) \overset{E}{\sim} q \\
\Gamma \vdash l (i = t: I) \overset{E}{\sim} q \\
\Gamma \vdash l (i = t: I) \overset{E}{\sim} q \\
\Gamma \vdash l (i = t: I) \overset{E}{\sim} q
\end{array}
\]

(h) Elaboration of constraints

\[
\begin{array}{c}
\Gamma \vdash (l) \supseteq (T) \overset{ls}{\sim} (t)
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash l \overset{ls}{\supseteq} t \overset{ls}{\sim} \text{is} \\
\Gamma \vdash D \overset{ls}{\supseteq} D \overset{ls}{\sim} \text{is} \\
\Gamma \vdash l \overset{ls}{\supseteq} t \overset{ls}{\sim} \text{is} \\
\Gamma \vdash \text{VALID} (\text{EQ-END}) \\
\Gamma \vdash e \overset{ls}{\supseteq} e \overset{ls}{\sim} *
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash l \overset{ls}{\supseteq} t \overset{ls}{\sim} \text{is} \\
\Gamma \vdash [p] l \overset{ls}{\supseteq} p t \overset{ls}{\sim} \text{is}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash l \overset{ls}{\supseteq} t \overset{ls}{\sim} \text{is} \\
\Gamma \vdash l \overset{ls}{\supseteq} t \overset{ls}{\sim} \text{is} \\
\Gamma \vdash l \overset{ls}{\supseteq} t \overset{ls}{\sim} \text{is}
\end{array}
\]

(i) Extraction of indices

Figure 7.5.: Elaboration of inductive families
7. Elaborating Inductive Definitions

7.45 Example (Vector, constrained). The elaboration of constraint-based vectors starts as follows:

**LET:** $\Gamma$ a valid context

\[ \Gamma \vdash \text{data } \mathbf{Vec} [A : \text{SET}](n : \text{Nat}) : \text{SET} \text{ where } [patts_\sim] \overset{D}{\sim} \]

\[ (A : \text{SET})(n : \text{Nat}) \rightarrow \text{SET} \]

**BY:** Example 7.51 applied to rule (DATA)

where

\[ patts_\sim \triangleq \mathbf{Vec} A \quad (n = 0) \ni \text{nil} \]
\[ \mathbf{Vec} A \quad (n = \text{suc } m) \ni \text{cons} (m : \text{Nat})(a : A)(\text{vs : Vec } A \; m) \]

\[ \text{code}_\sim \triangleq \text{return } \{ \text{'nil', 'cons} \}
\]
\[ \text{'nil} \mapsto \Sigma \; (n = 0) \; \lambda \rightarrow .1
\]
\[ \text{'cons} \mapsto \Sigma \; \text{Nat} \; \lambda m. \; \Sigma \; A \; \lambda \rightarrow . \text{var} (m, \ast) \times \Sigma \; (n = \text{suc } m) \; \lambda \rightarrow .1 \}

7.46 Example (Vector, computed). The same skeleton is used in the alternative definition of vectors, but the choices of constructors – and therefore the resulting code – are different:

\[ patts_{\rightarrow} \triangleq \mathbf{Vec} A \quad n \ni \text{Nat-case } n \]
\[ \mathbf{Vec} A \quad 0 \ni \text{nil} \]
\[ \mathbf{Vec} A \quad (\text{suc } m) \ni \text{cons} (a : A)(\text{vs : Vec } A \; m) \]

\[ \text{code}_{\rightarrow} \triangleq \text{return } \{ \text{'elim} \}
\]
\[ \text{'elim} \mapsto \text{Nat-case } n \; (\lambda n. \; \mathbf{Vec} [A](n : \text{Nat}))
\]
\[ \text{'elim} \mapsto \text{call} \{ \mathbf{Vec} [A] (0 : \text{Nat}) \} \{ \text{return } \{ \text{'nil} \mapsto 1 \} \}
\]
\[ \text{'elim} \mapsto \text{call} \{ \mathbf{Vec} [A] (\text{suc } m : \text{Nat}) \}
\]
\[ \text{return } \{ \text{'cons} \{ \text{'cons} \mapsto \Sigma \; A \; \lambda \rightarrow . \text{var} (m, \ast) \times 1 \} \}
\]

(7.47) Elaboration of patterns (Figure 7.5b.)

This judgment reads as: in context $\Gamma$, the index patterns $patts$ defining the datatype $l$ elaborate to a description $\text{code}$. This elaboration judgment is an extra step that was not necessary for inductive types. With inductive families, we can either constrain the index to some particular value or compute over the index to refine the choice of constructors. Hence, an inductive definition is a list of index patterns, potentially ending with a computation over the indices. Since the case where no index computation is performed is a special case of this rule, we save space and do not treat this case explicitly.

The elaboration of pattern choices consists in interpreting the datatype patterns of each constructor choice – via judgment $\overset{Cs}{\sim}$. The resulting labels are used to elaborate
7.2. Inductive Families

these constructors choices. If there is a computation over indices, we rely on elimination
with a motive [Goguen et al., 2006, McBride, 2002] – via judgment $\sim_{\text{eq}}$ – to generate
a type-theoretic term from the elimination principle provided by the user. We then
interpret each resulting subbranch as a pattern choice itself.

This elaboration step satisfies the following invariant:

7.48 Lemma. If $\Gamma \vdash I \ni \text{patts} \sim \text{code}$, then $\Gamma \vdash \text{code} : [I]$.

To prove this lemma, we need to show that pattern validation respects types:

7.49 Lemma. If $I \supseteq T \sim I_T$, then $\Gamma \vdash T = [I_T]_D$.

7.50 Remark. Note that we rely on the translation of datatype patterns $T$ into a label $l_T$ that carries the equations specified by $T$ – via judgment $I$; this device lets us postpone
the generation of the equality constraints until the end of the telescope of arguments,
for each argument. Indeed, we will be elaborating each individual constructor against
these labels: the rule (ARG-END) will trigger the generation of the equality constraints
at the end of each telescope of arguments.

7.51 Example (Vector, constrained). The elaboration of datatype patterns simply pro-
cceeds over the patterns of constructors, triggering the elaboration of patterns on nil and
cons:

LET: 1. $\Gamma$ a valid context
2. $A : \text{SET}$
3. $n : \text{Nat}$

\[\begin{array}{ll}
\langle 1 \rangle. & \vdash \text{Vec}[A](n : \text{Nat}) \ni \text{patts} \sim \text{code}\ \\
\langle 2 \rangle. & \text{Vec}[A](n : \text{Nat}) \supseteq \text{Vec}[A](n = 0) \sim \text{Vec}[A](n = 0 : \text{Nat}) \\
\text{BY: definition of } \sim_{\text{eq}}.
\end{array}\]

\[\begin{array}{ll}
\langle 2 \rangle. & \text{Vec}[A](n : \text{Nat}) \supseteq \text{Vec}[A](n = \text{suc } m) \sim \text{Vec}[A](n = \text{suc } m : \text{Nat}) \\
\text{BY: definition of } \sim_{\text{eq}}.
\end{array}\]

\[\langle 2 \rangle. \text{ Q.E.D.} \]

\[\text{BY: Example 7.55, } \langle 2 \rangle 1, \text{ and } \langle 2 \rangle 2 \text{ applied to rule (PATTERNS)}\]

where patts and code are the same as above.

7.52 Example (Vector, computed). For the other definition, the elaboration of patterns triggers the elaboration of a motive for the nil and cons patterns:

LET: 1. $\Gamma$ a valid context
2. $A : \text{SET}$
3. $n : \text{Nat}$

\[\begin{array}{ll}
\langle 1 \rangle. & \vdash \text{Vec}[A](n : \text{Nat}) \ni \text{patts} \sim \text{code}\ \\
\langle 2 \rangle. & \text{Vec}[A](n : \text{Nat}) \supseteq \text{Vec}[A](n = 0) \sim \text{Vec}[A](n = 0) \\
\text{BY: definition of } \sim_{\text{eq}}.
\end{array}\]

\[\begin{array}{ll}
\langle 2 \rangle. & \text{Nat-case } \lambda i_0. \text{Nat-case } i_0 \ \\
\text{BY: definition of } \sim_{\text{eq}}.
\end{array}\]

\[\langle 2 \rangle. \text{ Q.E.D.} \]

\[\text{BY: Example 7.55, } \langle 2 \rangle 1, \text{ and } \langle 2 \rangle 2 \text{ applied to rule (PATTERNS)}\]

where patts and code are the same as above.
7. Elaborating Inductive Definitions

BY: definition of \( \text{em} \)

SKETCH: In turn, this triggers the elaboration of constructor patterns for the \( \text{nil} \) and \( \text{cons} \) patterns:

\[
\langle 2 \rangle 3. \quad \vdash \text{Vec} [A] (0 : \text{Nat}) \supseteq \text{nil} \quad \text{Patts} \quad \text{return}\{ \text{'nil}' \} \quad \{ \text{'nil' } \mapsto \Sigma (n = 0) \lambda .1 \}\n\]

\[
\langle 3 \rangle 1. \quad \text{Vec} [A] (n : \text{Nat}) \supseteq \text{Vec} [A] (0 : \text{Nat}) \quad \text{LET}: \quad m : \text{Nat} \\
\vdash \text{Vec} [A] (suc m : \text{Nat}) \supseteq \text{Vec} [A] \quad \text{Patts} \quad \text{return}\{ \text{'cons}' \} \quad \{ \text{'cons' } \mapsto \Sigma A \lambda .\ \var (m, *) \times 1 \}\n\]

\[
\langle 3 \rangle 2. \quad \text{Q.E.D.} \quad \text{BY: Example 7.56 and } \langle 3 \rangle 1 \text{ applied to rule } \text{(PATTERNS)}
\]

\[
\langle 3 \rangle 3. \quad \text{Q.E.D.} \quad \text{LET}: \quad m : \text{Nat} \\
\vdash \text{Vec} [A] (suc m : \text{Nat}) \supseteq \text{Vec} [A] \quad \text{Patts} \quad \text{return}\{ \text{'cons}' \} \quad \{ \text{'cons' } \mapsto \Sigma A \lambda .\ \var (m, *) \times 1 \}\n\]

\[
\langle 3 \rangle 4. \quad \text{Q.E.D.} \quad \text{LET}: \quad m : \text{Nat} \\
\vdash \text{Vec} [A] (suc m : \text{Nat}) \supseteq \text{Vec} [A] \quad \text{Patts} \quad \text{return}\{ \text{'cons' \}} \quad \{ \text{'cons' } \mapsto \Sigma A \lambda .\ \var (m, *) \times 1 \}\n\]

This judgment reads as: in context \( \Gamma \), the sum of constructor \( \text{choices} \) elaborates to an enumeration of constructor names \( E \) and their respective codes \( T \). The elaboration of the choice of datatypes follows from the definition of inductive types (¶ 7.16): we merely collect the tag and code of each individual constructor and return them as enumerations.

This step is subject to the following soundness property:

**7.54 Lemma.** If \( \Gamma \vdash l \ni \text{choices} \sim [E \mapsto T] \), then \( \begin{cases} \Gamma \vdash E : \text{EnumU} \\ \Gamma \vdash T : \text{EnumT} \quad E \mapsto \text{iDesc} [I] D \end{cases} \)

**7.55 Example** (Vector, constrained). In the particular example of vector, there is only one choice of constructor when \( (n = 0) \) — namely \( \text{nil} \) — and when \( (n = \text{suc} m) \) — namely \( \text{cons} \). Therefore, we obtain the elaboration of choices from the elaboration of the unique constructor, in both cases:

\[
\langle 1 \rangle 1. \quad \vdash \text{Vec} [A] (n = 0 : \text{Nat}) \ni \text{nil} \sim [\{'nil' \} \mapsto \Sigma (n = 0) \lambda .1] \quad \text{BY: Example 7.59 applied to rule } \text{(CHOICES)}
\]
7.2. Inductive Families

\[
\vdash \text{Vec} [A] (n = \text{suc} m : \text{Nat}) \ni \text{cons} (m : \text{Nat}) (a : A) (\text{vs} : \text{Vec} A m)
\]

(1/2).

\[
\Gamma \vdash \text{Vec} [A] (n = \text{suc} m : \text{Nat}) \ni \text{cons} (m : \text{Nat}) (a : A) (\text{vs} : \text{Vec} A m)
\]

By: Example 7.59 applied to rule (CHOICES)

7.56 Example (Vector, computed). The same situation arises here: once we have determined which index we are dealing with, there is a single constructor available. Hence, we move from the elaboration of choices to the elaboration of the unique constructor:

LET: 1. \(\Gamma\) a valid context
2. \(A : \text{SET}\)
3. \(n : \text{Nat}\)

(1/1).

\[
\vdash \text{Vec} [A] (0 : \text{Nat}) \ni \text{nil} \vdash \text{Vec} A n
\]

By: Example 7.60 applied to rule (CHOICES)

(7.57) Elaboration of a constructor (Figure 7.5d.)

\[
\Gamma \vdash l \ni c \vdash [t \mapsto \text{code}]
\]

This judgment reads as: in context \(\Gamma\), the constructor \(c\) participating in the definition of datatype \(l\) is named \(t\) and its arguments are coded by \(\text{code}\). Again, the elaboration of a constructor follows the definition for inductive types (¶7.19).

It is subject to the following invariant:

7.58 Lemma. If \(\Gamma \vdash l \ni c \vdash [t \mapsto \text{code}]\), then \(\{\Gamma \vdash t : \text{UId} \}
\)

\(\{\Gamma \vdash \text{code} : \text{IDesc} [l] D\}\)

7.59 Example (Vector, constrained). Elaboration of the constructors is straightforward, simply switching to the elaboration of the arguments:

LET: 1. \(\Gamma\) a valid context
2. \(A : \text{SET}\)
3. \(n : \text{Nat}\)

(1/1).

\[
\vdash \text{Vec} [A] (n = 0 : \text{Nat}) \ni \text{nil} \vdash \text{Vec} A n
\]

By: Example 7.63 applied to rule (CONSTRUCTOR)

(1/2).

\[
\vdash \text{Vec} [A] (n = \text{suc} m : \text{Nat}) \ni \text{cons} (m : \text{Nat}) (a : A) (\text{vs} : \text{Vec} A m)
\]

By: Example 7.63 applied to rule (CONSTRUCTOR)

7.60 Example (Vector, computed). Similarly for the alternative definition, we have:

LET: 1. \(\Gamma\) a valid context
2. \(A : \text{SET}\)
3. \(n : \text{Nat}\)

(1/1).

\[
\vdash \text{Vec} [A] (0 : \text{Nat}) \ni \text{nil} \vdash [t \mapsto 1]
\]
7. Elaborating Inductive Definitions

BY: Example 7.64 applied to rule (CONSTRUCTOR)

LET: \( m : \text{Nat} \)

\[ \vdash \text{Vec} \, A \, (\text{suc} \, m : \text{Nat}) \ni \text{cons} \, (a : A) \, (\text{vs} : \text{Vec} \, A \, m) \]

(1)2. \[ \xi \, [\text{cons} \mapsto \Sigma \, A \, \lambda \, - \, \cdot \, \text{var} \, (m, \ast) \, \times \, 1] \]

BY: Example 7.64 applied to rule (CONSTRUCTOR)

Elaboration of arguments (Figure 7.5c.)

\[ \Gamma \vdash l \ni \text{args} \, A \, \triangleright \text{code} \]

This judgment reads as: in context \( \Gamma \), the telescope of arguments \( \text{args} \) participating in the definition of the datatype \( l \) elaborates to a description \( \text{code} \). The elaboration of arguments follows the same principle as for inductive types (¶ 7.22). However, after having encoded the telescope of arguments – via rule (ARG-SIG) and (ARG-REC), we must switch to translating the potential equality constraints – via rule (ARG-END). As hinted at in Remark 7.50, the equations are stored in the label \( l \).

This step is subject to the following soundness property:

7.62 Lemma. If \( \Gamma \vdash l \ni \text{args} \, A \, \triangleright \text{code} \), then \( \Gamma \vdash \text{code} : \text{IDesc} \, [l]_D \)

7.63 Example (Vector, constrained). We then elaborate the arguments by unfolding the telescope, at which point we switch to elaborating the constraints.

LET: 1. \( \Gamma \) a valid context

2. \( A : \text{SET} \)

3. \( n : \text{Nat} \)

SKETCH: We do so immediately in the \textit{nil} case:

(1)1. \[ \vdash \text{Vec} \, A \, (n = 0 : \text{Nat}) \ni e \, \triangleright \Sigma \, (n = 0) \, \lambda \, - \, . \, 1 \]

BY: Example 7.71 applied to rule (ARG-END)

SKETCH: While a few steps are necessary to elaborate the arguments in the \textit{cons} case, including the elaboration of the recursive call:

(1)2. \[ \vdash \text{Vec} \, A \, (n = \text{suc} \, m : \text{Nat}) \ni (m : \text{Nat}) \, (a : A) \, (\text{vs} : \text{Vec} \, A \, m) \]

\[ \triangleright \Sigma \, \text{Nat} \, \lambda \, m . \, \Sigma \, A \, \lambda \, - \, \cdot \, \text{var} \, (m, \ast) \, \times \, \Sigma \, (n = \text{suc} \, m) \, \lambda \, - \, . \, 1 \]

LET: \( m : \text{Nat} \)

(2)1. \[ \vdash \text{Vec} \, A \, (n = \text{suc} \, m : \text{Nat}) \ni (a : A) \, (\text{vs} : \text{Vec} \, A \, m) \]

\[ \triangleright \Sigma \, A \, \lambda \, - \, \cdot \, \text{var} \, (m, \ast) \, \times \, \Sigma \, (n = \text{suc} \, m) \, \lambda \, - \, . \, 1 \]

(3)1. \[ \vdash \text{Vec} \, A \, (n = \text{suc} \, m : \text{Nat}) \ni (\text{vs} : \text{Vec} \, A \, m) \]

\[ \triangleright \Sigma \, \text{var} \, (m, \ast) \, \times \, \Sigma \, (n = \text{suc} \, m) \, \lambda \, - \, . \, 1 \]

(4)1. \[ \vdash \text{Vec} \, A \, (n = \text{suc} \, m : \text{Nat}) \ni e \, \triangleright \Sigma \, (n = \text{suc} \, m) \, \lambda \, - \, . \, 1 \]

BY: Example 7.71 applied to rule (ARG-END)

(4)2. Q.E.D.

BY: Example 7.67, and (4)1 applied to rule (ARG-REC)

(3)2. Q.E.D.

BY: (3)1 applied to rule (ARG-SIG)
7.2. Inductive Families

(2)2. Q.E.D.

By: (2)1 applied to rule (ARG-SIG)

7.64 Example (Vector, computed). In the alternative definition, the process is similar:

LET: 1. $\Gamma$ a valid context
2. $A : \text{SET}$
3. $n : \text{Nat}$

\begin{align*}
\langle 1 \rangle 1. & A : \text{SET}, n : \text{Nat} \vdash \text{Vec} [A] (0 : \text{Nat}) \ni e \sim 1 \\
& \text{By: Example 7.72 applied to rule (ARG-END)}
\end{align*}

LET: $m : \text{Nat}$

\begin{align*}
\langle 1 \rangle 2. & \vdash \text{Vec} [A] (\text{suc} m) \ni (a : A) (\text{vs} : \text{Vec} A m) \sim \Sigma A \lambda \cdot \text{var} (m, *) \times 1 \\
\langle 2 \rangle 1. & \vdash \text{Vec} [A] (\text{suc} m) \ni (\text{vs} : \text{Vec} A m) \sim \text{var} (m, *) \times 1 \\
\langle 3 \rangle 1. & \vdash \text{Vec} [A] (\text{suc} m) \ni e \sim 1 \\
& \text{By: Example 7.72 applied to rule (ARG-END)}
\end{align*}

(3)2. Q.E.D.

By: Example 7.68, and (3)1 applied to rule (ARG-REC)

(2)2. Q.E.D.

By: (2)1 applied to rule (ARG-SIG)

(7.65) Elaboration of recursive arguments (Figure 7.5f.)

\[
\Gamma \vdash l \ni \text{arg} \sim R \text{code}
\]

This judgment reads as: in context $\Gamma$, the recursive argument $\text{arg}$ participating in the definition of the datatype $l$ elaborates to a description $\text{code}$. Again, the elaboration of recursive arguments follows the one for inductive types ($\S$ 7.25). However, when elaborating a recursive argument – via rule (ARG-REC-VAR), we must extract the indices for which that recursive step is taken – via judgment $\text{ids}$.

This step is subject to the following soundness property:

7.66 Lemma. If $\Gamma \vdash l \ni \text{args} \sim R \text{code}$, then $\Gamma \vdash \text{code} : \text{IDesc} [l]_D$

7.67 Example (Vector, constrained). There is no recursive argument in the $\text{nil}$ case. The elaboration of the recursive call in the $\text{cons}$ case is a direct application of (ARG-REC-VAR):

LET: 1. $\Gamma$ a valid context
2. $A : \text{SET}$
3. $n : \text{Nat}$
4. $m : \text{Nat}$

\begin{align*}
\langle 1 \rangle 1. & \vdash \text{Vec} [A] (n = \text{suc} m) : \text{Nat} \ni (\text{vs} : \text{Vec} A m) \sim \text{var} (m, *) \\
& \text{By: Example 7.75 applied to rule (ARG-REC-VAR)}
\end{align*}

7.68 Example (Vector, computed). In the alternative definition, the process is similar: no recursive argument in the $\text{nil}$ case and a direct appeal to (ARG-REC-VAR) in the $\text{cons}$ case:
7. Elaborating Inductive Definitions

LET: 1. Γ a valid context
2. A : SET
3. n : Nat
4. m : Nat

(1) 1. ⊢ Vec [A] (n = 0 : Nat) \sim (n = 0) \lambda \sim 1
2. ⊢ Vec [A] \sim 1

(3) 1. ⊢ Vec [A] \sim 1
   BY: rule (EQ-END)
2. Q.E.D.
   BY: (3) 1 applied to rule (EQ-PARAM)

(2) 2. Q.E.D.
   BY: (2) 1 applied to rule (EQ-CSTR)

LET: m : Nat

(1) 1. ⊢ Vec [A] (n = suc m : Nat) \sim (n = suc m) \lambda \sim 1
2. ⊢ Vec [A] \sim 1

(3) 1. ⊢ Vec [A] \sim 1
   BY: rule (EQ-END)
2. Q.E.D.
   BY: (3) 1 applied to rule (EQ-PARAM)

(2) 2. Q.E.D.
   BY: (2) 1 applied to rule (EQ-CSTR)

7.72 Example (Vector, computed). No equations are generated and, indeed, needed for
the alternative definition of vectors:

\[ \text{LET: } \begin{align*}
1. & \quad \Gamma \text{ a valid context} \\
2. & \quad A : \text{SET} \\
3. & \quad n : \text{Nat}
\end{align*} \]

\[ \begin{align*}
\langle 1 \rangle & \begin{align*}
1. & \vdash \operatorname{Vec} [A] (0 : \text{Nat}) \overset{\text{Eq}}{\sim} 1 \\
\end{align*} \\
\langle 2 \rangle & \begin{align*}
1. & \vdash \operatorname{Vec} [A] \overset{\text{Eq}}{\sim} 1 \\
\end{align*} \\
\langle 3 \rangle & \begin{align*}
1. & \vdash \operatorname{Vec} \overset{\text{Eq}}{\sim} 1 \\
\end{align*}
\]

\[ \begin{align*}
\text{BY: } \langle 3 \rangle \text{1 applied to rule (EQ-END)} \\
\langle 3 \rangle & \text{2. Q.E.D.} \\
\end{align*} \]

\[ \begin{align*}
\text{BY: } \langle 2 \rangle \text{1 applied to rule (EQ-PARAM)} \\
\langle 2 \rangle & \text{2. Q.E.D.} \\
\end{align*} \]

\[ \begin{align*}
\text{BY: } \langle 3 \rangle \text{1 applied to rule (EQ-PARAM)} \\
\text{BY: } \langle 2 \rangle \text{1 applied to rule (EQ-PARAM)} \\
\text{BY: } \langle 3 \rangle \text{1 applied to rule (EQ-PARAM)} \\
\text{BY: } \langle 2 \rangle \text{1 applied to rule (EQ-PARAM)} \\
\end{align*} \]

\[ \text{LET: } \begin{align*}
m : \text{Nat}
\end{align*} \]

\[ \begin{align*}
\langle 1 \rangle & \begin{align*}
1. & \vdash \operatorname{Vec} [A] (\text{suc } m : \text{Nat}) \overset{\text{Eq}}{\sim} 1 \\
\end{align*} \\
\langle 2 \rangle & \begin{align*}
1. & \vdash \operatorname{Vec} [A] \overset{\text{Eq}}{\sim} 1 \\
\end{align*} \\
\langle 3 \rangle & \begin{align*}
1. & \vdash \operatorname{Vec} \overset{\text{Eq}}{\sim} 1 \\
\end{align*}
\]

\[ \begin{align*}
\text{BY: } \langle 3 \rangle \text{1 applied to rule (EQ-END)} \\
\langle 3 \rangle & \text{2. Q.E.D.} \\
\end{align*} \]

\[ \begin{align*}
\text{BY: } \langle 2 \rangle \text{1 applied to rule (EQ-PARAM)} \\
\langle 2 \rangle & \text{2. Q.E.D.} \\
\end{align*} \]

\[ \begin{align*}
\text{BY: } \langle 3 \rangle \text{1 applied to rule (EQ-PARAM)} \\
\text{BY: } \langle 2 \rangle \text{1 applied to rule (EQ-PARAM)} \\
\text{BY: } \langle 3 \rangle \text{1 applied to rule (EQ-PARAM)} \\
\text{BY: } \langle 2 \rangle \text{1 applied to rule (EQ-PARAM)} \\
\end{align*} \]

\[(7.73) \quad \text{Extraction of indices (Figure 7.5i.)} \]

\[ \Gamma \vdash l \ni T \overset{\text{is}}{\sim} \text{is} \]

This judgment reads as: in context $\Gamma$, the recursive argument $T$ participating in the definition of datatype $l$ is indexed by $\text{is}$. This step extracts the indices at which the recursive argument is defined. To do so, we match the type definition with the datatype label. Parameters are checked for consistency, i.e. they must remain unchanged across recursive arguments. Indices are collected in a tuple terminated by the inhabitant of the unit type.

This ensures the following soundness property:

7.74 Lemma. If $\Gamma \vdash l \ni T \overset{\text{is}}{\sim}$, then $\Gamma \vdash \text{is} : \mathcal{I}$.  

7.75 Example (Vector, constrained). There is only one instance of recursive definition, in the \texttt{cons} case. Its elaboration goes as follows:

\[ \text{LET: } \begin{align*}
1. & \quad \Gamma \text{ a valid context} \\
2. & \quad A : \text{SET} \\
3. & \quad n : \text{Nat}
\end{align*} \]

\[ \begin{align*}
\langle 1 \rangle & \begin{align*}
1. & \vdash \operatorname{Vec} [A] (n = \text{suc } m : \text{Nat}) \ni \operatorname{Vec} A m \overset{\text{dx}}{\sim} (m, \ast) \\
\end{align*} \\
\langle 2 \rangle & \begin{align*}
1. & \vdash [A] (n = \text{suc } m : \text{Nat}) \ni A m \overset{\text{dx}}{\sim} (m, \ast)
\end{align*}
\]
7. Elaborating Inductive Definitions

(3) 1. ⊢ (n = suc m : Nat) ⊃ m \texttt{idx} (m, *)
(4) 1. ⊢ e \equiv e \texttt{idx}
   By: rule (EQ-END)
(4) 2. Q.E.D.
   By: (4) 1 applied to rule (EQ-CSTR)
(3) 2. Q.E.D.
   By: (3) 1 applied to rule (EQ-PARAM)
(2) 2. Q.E.D.
   By: (2) 1 applied to rule (EQ-START)

7.76 Example (Vector, computed). Similarly, the elaboration of the index for the recursive definition is as follows:

LET: 1. \( \Gamma \) a valid context
2. \( A : \text{SET} \)
3. \( n : \text{Nat} \)
4. \( m : \text{Nat} \)
(1) 1. \( \vdash \text{Vec} [A] (suc m : \text{Nat}) \triangleright \text{Vec} A m \texttt{idx} (m, *) \)
(2) 1. \( \vdash [A] (suc m : \text{Nat}) \triangleright A m \texttt{idx} (m, *) \)
(3) 1. \( \vdash (suc m : \text{Nat}) \triangleright m \texttt{idx} (m, *) \)
(4) 1. \( \vdash e \equiv e \texttt{idx} \)
   By: rule (EQ-END)
(4) 2. Q.E.D.
   By: (4) 1 applied to rule (EQ-CSTR)
(3) 2. Q.E.D.
   By: (3) 1 applied to rule (EQ-PARAM)
(2) 2. Q.E.D.
   By: (2) 1 applied to rule (EQ-START)

(7.77) Soundness. Having stated the soundness properties of each individual elaboration step, we can now state and prove the soundness of the elaboration of inductive families.

7.78 Theorem (Soundness of elaboration).

If \( \Gamma \vdash \text{data } D [p : P] (i : I) : \text{SET} \text{ where choices } D \triangleright \Delta \), then \( \Delta \vdash \text{VALID} \).

Proof. First, we prove that labels elaborate to a type-correct index (Lemma 7.74). We then prove that the constraints generated by interpreting the label are valid descriptions (Lemma 7.70). From these lemmas, we can prove the soundness of the elaboration of recursive arguments by induction on their structure (Lemma 7.66), followed by the soundness of arguments by induction over the telescope of arguments (Lemma 7.62). We then deduce the validity of the elaboration of a constructor (Lemma 7.58). Using that lemma over each constructor, we thus obtain the soundness of the elaboration of choices (Lemma 7.54). Using this result and assuming the soundness of elimination with a motive (if \( \Gamma \vdash e \text{\texttt{eqm}} e' \in T \), then \( \Gamma \vdash e' : T \)), we obtain Lemma 7.48. This gives the desired result.
(7.3) **Completeness.** As mentioned in Remark 7.38, our syntax is a strict superset of Coq’s syntax. Consequently, it is impossible to prove a completeness result for inductive families in the vein of Theorem 7.34. Nonetheless, if we drop our support of computation over indices, we then fall back in the realm of Coq’s inductive definitions. In this restricted domain, we conjecture the completeness of elaboration:

**7.80 Conjecture** (Completeness with respect to Coq). For an inductive family

\[ \text{Ind}(X:\vec{I} \to \text{SET}) \langle C_0 | \ldots | C_n \rangle \]

in Coq, any function introduced by a Fixpoint definition over \( X \) admits an extensionally equivalent definition in our system. Conversely, our generic elimination principle is accepted by Coq.

**7.81 Remark.** Ultimately, our elaboration rules (Figure 7.5, Figure 7.5, and Figure 7.5) define the semantics of inductive families by translation to indexed descriptions. As such, the soundness lemmas provide supporting evidences that these definitions are conform to our intents. Albeit not entirely satisfactory, this situation must be compared to current practice in Coq and Agda. In those systems, the semantics of inductive families is given de facto by the positivity checker, a large yet unspecified piece of software.

### 7.3. Reflections on Inductive Types

(7.82) Having described our elaboration infrastructure for inductive definitions, we now give an overview of the possibilities offered by such a system. Indeed, in a purely syntactic presentation of inductive definitions, we are stuck working at the meta-level of the type theory: if we want to provide support for meta-programming over inductive types, it must be implemented as an extension of the theorem prover, out of the type theory.

Because our type theory reflects inductive definitions in itself, the meta-theory of inductive types is no more than a universe. What used to be meta-theoretical constructions can now be implemented within type theory, benefiting from the various amenities offered by a dependently-typed programming language. By adequately extending the elaboration machinery, the user benefits from a convenient and high-level syntax to refer to these type-theoretic constructions. In this section, we present two examples of “reflections on inductive types”.

Our first example, reflecting constructions on constructors [McBride et al., 2004], will appeal to the implementers: we hint at the possibility of implementing key features of type theory within itself, a baby step toward bootstrapping. Our second example, providing a user defined deriving mechanism, should appeal to programmers: we illustrate how programmers could provide generic operations over datatypes and see them automatically integrated in their development.
7. Elaborating Inductive Definitions

7.83 Remark. For simplicity and conciseness, we shall define these mechanisms over our universe of inductive types, \( \text{Desc} \). Nonetheless, it is straightforward, but more verbose, to define these constructions for inductive families.

7.3.1. A few constructions on constructors, internalised

Model: Chapter7.Case, Chapter7.NoConfusion

McBride et al. [2004] describe a collection of lemmas that a theorem prover’s implementer would like to export with every inductive type. The authors first show how one can reduce case analysis and course-of-value recursion to standard induction. Then, they describe two lemmas over datatype constructors: the no confusion lemma – constructors are injective and disjoint – and the acyclicity lemma – we can automatically disprove equalities of the form \( x = t \) where \( x \) appears constructor-guarded in \( t \). However, since that paper works on the syntactic form of datatype definitions, its “definitions” are given over skeletons of inductive definitions filled with ellipsis. For example, the authors reduce case analysis to induction with no less than ten ellipsis.

7.85 Definition (Case analysis). In our system, we generically implement case analysis by a mere definition within type theory. To do so, we simply ignore the induction hypothesis from the generic induction principle:

\[
\text{case } (D: \text{Desc})(P: \mu D \rightarrow \text{SET})(\text{cases} : ((d: [D] (\mu D)) \rightarrow P(\text{ind}))(x: \mu D) : Px
\]

\[\text{case } D P \text{ cases } x \mapsto \text{induction } (\lambda d. \lambda - . \text{cases } d) x\]

To account for case analysis during elaboration, we simply add a judgment \( \Gamma \vdash \text{data } D (p:P) : \text{SET} \text{ where choices } \sim \Delta \)

This judgment reads as: in context \( \Gamma \), the datatype declaration of \( D \) elaborates to an extended context \( \Delta \), which defines the code of this declaration, denoted \( D\text{-Desc} \), together with a case analysis operator, denoted \( D\text{-case} \). The inference rule by itself is not surprising. We reuse the elaboration machinery to obtain the code of the datatype. We then specialise the \text{case} function to this datatype by partial application.

7.87 Remark. In Figure 7.6, we extend the context with a definition \( D\text{-Desc} \) containing the code of a description. This definition is useful on its own, independently of case analysis. One could easily extend the elaboration of datatypes (Figure 7.2a) to contain this definition directly. In the following, we should always assume that \( D\text{-Desc} \) is in context for a datatype definition \( D \).

Similarly, the authors specify and prove the “no confusion” lemma over the skeleton of an inductive definition. In our system, this result is internalised through two definitions. In the following, we will assume that \( D \) is a tagged description, i.e. \( D = \Sigma E T \) where \( E \) is the enumeration of constructor labels. An inhabitant of \( \mu D \) is therefore a
7.3. Reflections on Inductive Types

\[ \Gamma \vdash \text{data } D \vdash \text{SET} \]  
where \( \text{choices} \vdash \text{code} \)

\[
\begin{align*}
\Gamma \vdash & \text{SET} \ni (p : P) \rightarrow \text{SET} \\
\Gamma \vdash & (p : P) \rightarrow \text{SET Chk} \\
\Gamma \vdash & (p : P') \rightarrow \text{SET} \\
\Gamma \vdash & p : P' \ni \text{choices} \ni \text{code}
\end{align*}
\]

Figure 7.6.: Elaboration of case analysis

pair in \((c, a)\) with \(c\) representing the constructor name and \(a\) the tuple of arguments.

7.89 Lemma (“No confusion”, internalised). The NoConfusion lemma states that two propositionally equal terms \(x, y : \mu D\) must have the same constructor (otherwise, second case, we ask for the impossible) and their arguments must be equal (first case):

\[
\begin{align*}
\text{NoConfusion} & (x : \mu D) (y : \mu D) : \text{SET}_1 \\
\text{NoConfusion} & (\text{in} (c_x, a_x)) (\text{in} (c_y, a_y)) \mid \text{decideEq-EnumT } c_x c_y \\
\text{NoConfusion} & (\text{in} (c_x, a_x)) (\text{in} (c_y, a_y)) \mid \text{equal refl } \mapsto (P : \text{SET}) \rightarrow (a_x = a_y \rightarrow P) \rightarrow P \\
\text{NoConfusion} & (\text{in} (c_x, a_x)) (\text{in} (c_y, a_y)) \mid \text{not-equal } q \mapsto (P : \text{SET}) \rightarrow P
\end{align*}
\]

The function \(\text{decideEq-EnumT}\) decides whether two indices into an enumeration are equal or not. This is indeed a decidable property: an index is nothing but a natural number (bounded by the size of the enumeration). We put the result of \(\text{decideEq-EnumT } c_x c_y\) under the scrutiny of \(\text{NoConfusion}\) by using the “with” (\(|\)) gadget (¶ 3.36).

Proof. The proof of this lemma consists simply in deciding whether the constructor tag are equal or not, hence discriminating the constructors and deconstructing the equality:

\[
\begin{align*}
\text{noConfusion} & (x : \mu D) (y : \mu D) (q : x = y) : \text{NoConfusion } x y \\
\text{noConfusion} & (\text{in} (c_x, a_x)) (\text{in} (c_y, a_y)) q \mid \text{decideEq-EnumT } c_x c_y \\
\text{noConfusion} & (\text{in} (c_x, a_x)) (\text{in} (c_y, a_y)) q \mid \text{equal refl } \mapsto \lambda P. \lambda \text{rec. rec } q \\
\text{noConfusion} & (\text{in} (c_x, a_x)) (\text{in} (c_y, a_y)) q \mid \text{not-equal } \text{neq } \mapsto \lambda P. \text{0-elim } (\text{neq } q)
\end{align*}
\]

(7.90) We have proved this lemma generically, for all tagged descriptions. Hence, users can directly use it on their datatypes. For convenience, a subsequent elaboration phase can specialise this lemma for each datatype definition. Since the corresponding inference rule is similar to the one computing the datatype-specific case analysis (¶ 7.86), we do not dwell on it further.
7. Elaborating Inductive Definitions

7.3.2. Deriving operations on datatypes

MODEL: Chapter7.Derivable

Another extension of our system is a generic deriving mechanism. In the Haskell language, we can write a definition such as

```haskell
data Nat : SET where
  Nat ⊇ 0
  | suc (n : Nat)
deriving Eq
```

that generates automatically an equality test for the given datatype. Since datatypes are a meta-theoretical entity, this deriving mechanism has to be provided by the implementer and, template programming aside, it cannot be implemented by the programmers themselves. This issue was identified and treated in Haskell by several authors [Hinze and Jones, 2001, Magalhães et al., 2010], relying solely on multi-parameter type classes. By contrast, our solution is implemented entirely in type theory while the syntactic treatment of the deriving clause is handled by elaboration.

We now extend our elaborator with such a deriving mechanism. However, for this mechanism to work, we must restrict ourselves to decidable properties: for example, if the user asks to derive equality on a datatype that does not admit a decidable equality (e.g. Brouwer ordinals), it should fail immediately. To solve this issue, we add one level of indirection: while we cannot decide equality for every datatype, we can decide whether a datatype belongs to a subuniverse for which equality is decidable.

7.93 Definition (Derivable properties). To introduce a derivable property \( P \) in the system, one must populate the following record structure:

```
Derivable (P : Desc → SET) : SET₁
```

That is, one must specify a subuniverse of descriptions, \( \text{subDesc} \). The membership of a description to this subuniverse must be a decidable property, as witnessed by \( \text{membership} \). Finally, in order to actually derive the property \( P \), one must provide a \( \text{derive} \) function, taking an inhabitant of the subuniverse and providing a witness of \( P \).

7.94 Definition (Decision procedure). A decision procedure for a proposition \( P \) returns a positive answer if and only if \( P \) is true. It is said to be sound (the if direction) and complete (the only if direction). In type theory, a property \( P \) on a set \( X \) is decidable if we can implement a function

```
decide_P : (x : X) → Decidable (P x)
```
where `Decidable` is defined by:

```haskell
data Decidable [A : SET] : SET where
  Decidable A \equiv yes (a : A) \\
  \mid no (na : A \rightarrow \emptyset)
```

(7.95) Derivable properties range from generic lemmas – such as decision procedures for equality, or ordering – to generic functions – such as generic marshalling, or unmarshalling. The only requirements are that its type is itself of type `Desc \rightarrow SET`, and that the program is definable over a subuniverse of descriptions. We illustrate our approach with a decision procedure for equality.

### 7.96 Example (Derivable property: equality)

In the case of equality\(^1\), the programmer has first to provide a function `eqDesc : Desc \rightarrow SET_1`. One possible (perhaps simplistic, but valid) subuniverse consists only of products, finite sums, recursive call, and unit: it is enough to describe natural numbers and variants thereof.

One then implements a procedure

```
membershipEq : (D : Desc) \rightarrow Decidable (eqDesc D)
```

deciding whether a given `Desc` code fits into this subuniverse or not. It should be clear that the membership of a `Desc` code to our subuniverse of finite products and sums is decidable: we simply match on the code of `D`, returning `yes` when that code belongs to `eqDesc`, and `unlikely` otherwise.

Finally, one implements the key operation

```
deriveEq : eqDesc D \rightarrow (x y : \mu D) \rightarrow Decidable (x = y)
```

that decides equality of two objects, assuming that they belong to the subuniverse. This decision procedure is simply a matter of traversing a first-order structure and checking objects of finite cardinality for equality. Put together, this populates a record:

```
Eq : Derivable (\lambda D. (x y : \mu D) \rightarrow Decidable (x = y))
```

(7.97) While elaborating a datatype, it is then straightforward – and automatic – for us to generate its derivable property, or reject it immediately: we simply unfold `membership` on the specific code. If we obtain a positive witness, we pass that witness to `derive` and instantiate the property. If we obtain a negative response, the elaborated term simply fails to type check: in practice, one would certainly report a more informative error.

### 7.98 Example (Deriving equality: Natural numbers)

Since natural numbers fit into the `eqDesc` universe, the elaboration machinery can automatically generate the following

\(^1\)We refer our reader to `Chapter7.Derivable.Equality` for the implementation details.
7. Elaborating Inductive Definitions

\[ \Gamma \vdash \text{prop} \vdash \text{prop}' \in \text{Derivable P} \]

\[ \Gamma \vdash \text{data } D \vdash \text{SET where choices deriving prop} \]

\[ \Gamma \vdash \text{decide-P } \mapsto \text{prop'.derive (witness (prop'.membership D-Desc) *)} \]

\[ : (x:\mu D-Desc) \rightarrow P x \]

Figure 7.7.: Elaboration of deriving

decision procedure

\[
\begin{align*}
\text{Nat-eq } (x \cdot y: \text{Nat}) & : \text{Decidable } (x = y) \\
\text{Nat-eq } x \cdot y & \mapsto \text{deriveEq (witness (membershipEq NatD) *)}
\end{align*}
\]

without any input from the user but the deriving Eq clause. Here, \text{witness} is a library function that extracts the witness from a true decidable property:

\[
\begin{align*}
\text{witness } (w: \text{Decidable } A) \cdot (t: \text{True } w) & : A \\
\text{witness } (\text{yes } a) & \mapsto 1 \\
\text{witness } (\text{no } na) & \mapsto 0
\end{align*}
\]

Applied to \text{NatD}, the function \text{membershipEq} computes a positive witness that we can simply extract.

Having laid down the low-level machinery, we then support it with some high-level syntax. To this end, we extend the datatype definition with a deriving clause:

\[
\langle \text{data} \rangle ::= \text{data } D \langle \text{paramTel} \rangle : \langle T \rangle \text{ where choices deriving } \langle t \rangle
\]

We process these definitions through the judgment:

\[ \Gamma \vdash \text{data } D \vdash \text{SET where choices deriving } P \vdash \Delta \]

This judgment reads as: in context \(\Gamma\), the datatype definition deriving a property \(P\) elaborates to a context \(\Delta\), which either defines a function \(\text{decide-P}\) that decides \(P\), or is invalid – \textit{i.e.} \(\Delta \not\vdash \text{VALID}\). The inference rule (Figure 7.7) consists simply in extending \(\Gamma\) with a function extracting the decidability witness, as we did in Example 7.98, thus obtaining our decision procedure.

\textbf{7.100 Remark} (type checking and error management). We remark that the witness extraction might fail: if the decision procedure rejects the datatype, the \text{witness} function asks for an inhabitant of \(0\). Thus, our generated term, providing an inhabitant of \(1\), does not type check. This behavior is the expected one: we want to reject deriving state-
ments that cannot be decided. In an actual implementation, we would normalise the result of the decision procedure membership beforehand, and report a more convenient error message.
7. Elaborating Inductive Definitions

Conclusion

(7.101) In this chapter, we have given an elaboration of inductive definitions in type theory. We have adopted a relational approach, specifying the elaboration through a system of inference rules. We thus abstracted away the nitty-gritty operational details and focused on the translation of our syntactic constructions to the universe of codes. Overall, our presentation is conceptually simple and amenable to formal reasoning, as demonstrated by our correctness proof. Its relative simplicity (compared to, say, a positivity checker) together with the soundness proofs should convince the reader of its validity.

(7.102) From there, we believe that reasoning on inductive definitions can be liberated from the elusive ellipsis: proofs and constructions on inductive types ought to happen within the type theory itself. After Harper and Stone [2000], we claim that if the treatment of datatypes is conceptually straightforward then it ought to be technically straightforward and implemented as a generic program in type theory. For non-straightforward properties, our results should be reusable across calculi – such as the Calculus of Inductive Constructions – and not too rigidly tied to our universe of datatypes. Besides, we were careful to present elaboration as a relation rather than a mere program, making it more amenable to abstract reasoning.

(7.103) We also had a glimpse at two possible extensions of the elaboration process. We have seen how generic theorems on inductive types can be internalised as generic programs: besides the benefit of reducing the trusted computing base, their validity is guaranteed by type checking. Also, we have presented a generic deriving mechanism: with no extension to the type theory, we are able to let the user define subuniverses that support certain operations. These operations can then be automatically specialised to the datatypes that support them, without any user intervention.

Related work

(7.104) This chapter has been strongly influenced by the work of Harper and Stone [2000] on the formalisation of Standard ML in type theory. In their work, the authors gave a semantics of Standard ML – the full-blown language – through a translation of the high-level syntactic concepts – such as pattern-matching, data-type constructors, etc. – to type theory. This elaboration of a programming language down to a basic calculus was then pursued by Dreyer [2005] in his work on ML’s module system. Elaboration lets us bridge the gap between the theory – the basic calculus – and the implementation, without having to rely on a standard written in plain English, and therefore subject to misinterpretation.

While Standard ML is simply typed, this technique naturally applies in a dependent setting. The work of Giménez [1995] can be seen as an instance of this technique: to justify the Fixpoint definition in Coq, the author gave a translation of such definitions to an equivalent one using only induction on the datatype on which the fixpoint is constructed. However, this analysis remain at the meta-level: it aims at justifying the syntax, but the implementation does not benefit from this simplification.

McBride and McKinna [2004] took a step further and presented a programming lan-
7.3. Reflections on Inductive Types

guage that is entirely elaborated down to a basic calculus. In that setting, recursive functions are automatically translated to definitions by induction. Consequently, the implementer need only provide these elimination principles. However, inductive definitions were still treated at the meta-level, in a purely syntactic manner. The present chapter completes this work by elaborating the inductive fragment. This allows us to implement many conveniences for manipulating inductive types in type theory: for example, the constructions on constructors become mere (generic) programs, in a polytypical style reminiscent of the work of Hinze [2000a] in a non-dependent setting.

Elaboration is also used in Matita [Asperti et al., 2012] as a mean to enrich the language of expressions with a more convenient, type-directed syntax. As for us, having a bidirectional type checker is crucial in their work: types drive the refinement of the high-level syntax to a relatively simple core calculus. However, being stuck at the syntactic level, inductive definitions are poorly handled.
Part IV.

A Calculus of Structures

In this fourth and last part, we build upon our internalised presentation of inductive types. First, in Chapter 8, we adapt an universe of ornaments to our universe of indexed descriptions. Ornaments let us describe structure-preserving transformations of datatypes. We study their algebraic properties, hence developing an interesting "calculus of structures".

In Chapter 9, we put ornaments at work. While the universe of ornaments is restricted to transformations on datatypes, we present a universe of functional ornaments that lets us relate functions operating on similarly-structured datatypes.

Crucially, this last part does not require any support from the type theory, aside from a universe of datatypes. The theory of ornaments and functional ornaments is entirely developed within type theory. This illustrates, once more, the benefit of an internalised presentation of datatypes: the meta-theory has been swallowed by the theory.
Imagine designing a library for an ML-like language. For instance, we start with natural numbers and their operations, then we move to binary trees, then rose trees, etc. It is the garden of Eden: datatypes are data-structures, each coming with its unique and optimised set of operations. If we move to a language with richer datatypes, such as a dependently-typed language, we enter the Augean stables. Where we used to have binary trees, now we have complete binary trees, red-black trees, AVL trees, and countless other variants. Worse, we have to duplicate code across these tree-like datatypes: because they are defined upon this common binarily branching structure, a lot of computationally identical operations will have to be duplicated for the type checker to be satisfied.

Since the ML days, datatypes have evolved: besides providing an organising structure for computation, they are now offering more control over what is a valid result. With richer datatypes, we can enforce invariants on top of the data-structures. In such a system, programmers strive to express the correctness of programs in their types: a well-typed program is correct by construction, the proof of correctness being reduced to type checking. A simple yet powerful recipe to obtain these richer datatypes is to index the data-structure. We have described these objects as inductive families in Chapter 5. Inductive families made it to mainstream functional programming with GADTs. Refinement types [Freeman and Pfenning, 1991, Swamy et al., 2011] are another technique to equip data-structures with rich invariants. Atkey et al. [2012] have shown how refinement types relate to inductive families, through a generalisation of algebraic ornaments (Section 8.1.3).

When programming in type theory, we are in a sweet spot: because the programming language is also a logic, our datatypes are data-structures that comes naturally equipped with a data-logic. The structure captures the dynamic, operational behavior expected from the datatype. It corresponds to, say, the choice between a list or a binary tree, which is mostly governed by performance considerations. The logic, on the other hand, dictates the static invariants of the datatype. For example, by indexing lists by their length, thus obtaining vectors, we integrate a logic of length with the data. We can then take an $m \times n$ matrix to be a plainly rectangular $m$-vector of $n$-vectors, rather than a list of lists together with a proof that measuring each length yields the same result.

However, these carefully crafted datatypes are a threat to any library design: the same data-structure is used for logically incompatible purposes. This explosion of specialised datatypes is overwhelming: these objects are too specialised to fit in a global library. Yet, because they share this common recursive structure, many operations on them are extremely similar, if not exactly the same. The objective of this chapter is to
8. Ornaments

characterise, first intensionally and then categorically, these structure-preserving transformations of datatypes. Having such a characterisation, we are able to extend our datatypes on-demand with special-purpose logics without severing the structural ties between them.

To address this issue, McBride [2013] developed *ornaments*, which describe how one datatype can be enriched into another *with the same structure*. Let us give a few examples of ornamental transformations. We shall focus here on the result of the transformation: a new datatype, derived from another one. The ornament itself is the intensional object that describes these structure-preserving transformations, which we shall formally define and illustrate more concretely in the next section. Such transformations take two forms. First, we can *extend* the initial type with more information.

**8.4 Example** (Ornament: Extending the Booleans to the option type). We can extend the Booleans to the option type by attaching an \( a : A \) to the constructor `true`:

```plaintext
data Bool : SET where
  true : Bool
  false : Bool

data Maybe [A : SET] : SET where
  just (a : A) : Maybe A
  nothing : Maybe A
```

**8.5 Example** (Ornament: Extending numbers to lists). Or we can extend numbers to lists by inserting an \( a : A \) at each successor node:

```plaintext
data Nat : SET where
  0 : Nat
  suc (n : Nat) : Nat

data List [A : SET] : SET where
  nil : List A
  cons (a : A) (as : List A) : List A
```

**8.6 Example** (Ornament: Extending numbers to typed contexts). Extensions are not necessarily parametric. For example, the datatype `Context` used to define minimal logic (Example 3.51) is an extension of numbers:

```plaintext
data Nat : SET where
  0 : Nat
  suc (n : Nat) : Nat

data Context : SET where
  ϵ : Context
  (Γ : Context) ; (T : Type) : Context
```

Second, we can *refine* the indexing of the initial type, following a finer discipline. By refining the indices of a datatype, we make it logically more discriminating.

**8.7 Second, we can refine** the indexing of the initial type, following a finer discipline. By refining the indices of a datatype, we make it logically more discriminating.

**8.8 Example** (Ornament: Refining numbers to finite sets). We refine natural numbers to finite sets by indexing the number with an upper-bound:

```plaintext
data Nat : SET where
  0 : Nat
  suc (n : Nat) : Nat

data Fin (n : Nat) : SET where
  f0 (n′ : Nat)
  fsuc (n′ : Nat) (k : Fin n′)
```

The datatype `Fin n` describes the (finite) sets of cardinality 0 to \( n - 1 \).

We can also do both at the same time, as illustrated by the following example.

```
8.9 Example (Ornament: Extending numbers to typed contexts and refining). We can extend numbers to types by attaching an \( a : A \) to \( 0 \) and \( suc (n : Nat) \), following a finer discipline. By doing so, we make the types more discriminating.
```
8.9 Example (Ornament: Extending and refining numbers to vectors). We extend natural numbers to lists while refining the index to represent the length of the list thus constructed:

\[
\begin{align*}
\text{data } \text{Nat} & : \text{SET} \text{ where } \\
\text{Nat} & \ni 0 \\
| & \text{suc } (n : \text{Nat})
\end{align*}
\]

\[
\begin{align*}
\text{data } \text{Vec } [A : \text{SET}](n : \text{Nat}) & : \text{SET} \text{ where } \\
\text{Vec } A & (n = 0) \ni \text{nil} \\
| & \text{suc } (n' : \text{Nat}) \ni \text{cons } (n' : \text{Nat})(a : A)(vs : \text{Vec } A n')
\end{align*}
\]

(8.10) To complete our intuition of ornaments, let us present a non-example of ornament. We can spot that a datatype cannot ornament another by focusing on their recursive structure. If we cannot map the structure of one to the other, then this transformation is not in the realm of ornaments.

8.11 Non-example (Trees do not ornament numbers). Binary trees cannot ornament natural numbers: natural numbers have a linear structure – its constructors have arity 0 or 1 – while binary trees have a binarily-branching structure – its constructors have arity 0 or 2:

\[
\begin{align*}
\text{data } \text{Nat} & : \text{SET} \text{ where } \\
\text{Nat} & \ni 0 \\
| & \text{suc } (n : \text{Nat})
\end{align*}
\]

\[
\begin{align*}
\text{data } \text{Tree } [A : \text{SET}] & : \text{SET} \text{ where } \\
\text{Tree } A & \ni \text{leaf} \\
| & \text{node } (lb : \text{Tree } A)(a : A)(rb : \text{Tree } A)
\end{align*}
\]

(8.12) Because of their constructive nature, ornaments are not merely identifying similar structures: they give an effective recipe to build new datatypes from old, guaranteeing by construction that the structure is preserved. Hence, we can obtain a plethora of new datatypes with minimal effort.

8.13 Remark. Unlike the previous chapters, this chapter’s material is directly presented on our universe of indexed descriptions. Indeed, the point of ornaments is to relate indexed datatypes through their shared structure. On non-indexed types, ornaments are of lesser practical interest.

8.14 Remark (Meta-theoretical status). It is crucial to note that this chapter is built entirely within type theory. No change or adaptation to the meta-theory is required. In particular, the validity of our constructions is justified by mere type checking.

8.1. Universe of Ornaments

MODEL: Chapter8.Ornament

(8.15) Originally, McBride [2013, §4] presented the notion of ornament for a universe where the indexing discipline could only be enforced by equality constraints. Consequently, in that simpler setting, computing types from indices was impossible. We shall now adapt the original definition to our setting. Following Remark 5.6, we first focus on describing functors on \([\text{SET} / I, \text{SET}]\) (Definition 8.17). We shall then lift our definition...
pointwise to $[\text{SET}/I, \text{SET}]$ (Definition 8.20).

Our grammar of ornaments is similar to the original one: we can copy the base datatype (with the codes $1$, $\Pi$, $\Sigma$, $\times$, and $\sigma$), extend it by inserting sets (with the code $\text{insert}$), and refine the indexing subject to the relation imposed by $u$ (with the code $\text{var}$). However, we also have the $J$-index in our context: following Brady’s insight that inductive families need not store their indices [Brady et al., 2003], we can delete parts of the datatype definition as long as we can provide a witness.

8.17 Definition (Universe of ornaments). The universe of ornaments is presented in Figure 8.1. Its interpretation computes the description of the extended datatype. This consists in traversing the ornament code, turning the $\text{insert}$ codes into $\Sigma$ codes. In the $\text{delete}$ case, no $\Sigma$ code is generated: we use the witness to compute the extension of the rest of the ornament.

8.18 Remark (Notation). We overload the $\text{IDesc}$ constructor names in the definition of the $\text{Orn}$ universe. Indeed, the ornamental codes $1$, $\times$, $\sigma$, $\Pi$, and $\Sigma$ merely duplicate the underlying description. Similarly, the $\text{var}$ ornament takes an index $k$ that must be related to the underlying index $i$ through the function $u$. More generally, datatype’s constructors may freely be overloaded: for a bidirectional type checker, there is no ambiguity since constructors are checked against their type.

8.20 Definition (Ornament). An ornament is the lifting of an $\text{Orn}$ code to a $J$-indexed set:

$$\text{orn} (D : \text{func} K L) (u : I \rightarrow K) (v : J \rightarrow L) : \text{SET} \rightarrow \text{orn} (D (v j)) u \xrightarrow{\lambda j. [\sigma]_{\text{orn}}} \text{func} I J$$

8.21 Remark (Notation). We adopt an informal notation to describe ornaments conveniently. The idea is to simply mirror our data definition, adding from which datatype the ornament is defined. When specifying a constructor, we can then extend it with new information – using $[x : S]$ – or delete an argument originally named $x$ by providing a witness – using $[x \equiv s]$. We now rephrase our earlier examples using this language.

Just as the datatype declaration syntax was elaborated to $\text{IDesc}$ codes (Chapter 7), this high-level syntax elaborates to ornaments. A formal description of the translation is beyond the scope of this thesis\(^1\). Nonetheless, a few examples are enough to illustrate our notation and shall help us build some intuition for ornaments.

8.22 Example (Ornament: from Booleans to the option type). We obtain the option type

\(^1\)The treatment of inductive definitions in Chapter 7 was already rather dense. We believe that further simplification of our system is necessary before tackling the elaboration of ornaments.
8.1. Universe of Ornaments

data Orn \((D : \text{IDesc } K)[u : I \to K] : \text{SET}_1\) where

- Extend with \(S\):
  Orn \(D \ u \ni\) insert \((S : \text{SET})(D^+ : S \to \text{Orn } D u)\)

- Refine the index:
  Orn \((\text{var } k) \ u \ni\) var \((i : u^{-1} k)\)

- Copy the original:
  Orn \(1 \ u \ni\) 1

- Delete \(S\):
  | delete \((s : S)(T^+ : \text{Orn}(T s) u)\)

(a) Code

(b) Interpretation

Figure 8.1.: Universe of ornaments
8. Ornaments

from the Booleans by inserting an extra \( a : A \) in the \texttt{true} case:

\[
\text{data} \; \text{Maybe} \left[ A : \text{SET} \right] \; \text{from} \; \text{Bool} \; \text{where} \\
\text{Maybe} \; A \; \ni \; \text{just} \; [a : A] \\
\text{nothing}
\]

\[
\text{MaybeO} \left( A : \text{SET} \right) : \; \text{orn} \; \text{BoolD} \; \text{id} \; \text{id} \\
\text{MaybeO} \; A \; \mapsto \; \lambda* \cdot \Sigma \left\{ \begin{array}{l}
\text{true} \mapsto \text{insert} \; A \; \lambda \cdot \left[ . \text{var} \; 1 \right] \\
\text{false} \mapsto 1
\end{array} \right\}
\]

The reader will check that the interpretation of this ornament (by \( [[ - ]_{\text{orn}}] \)) followed by the interpretation of the resulting description (by \( [[ - ]] \)) yields the signature functor of the option type:

\[
[[[\text{MaybeO} \; A]_{\text{orn}}] \; X] \cong 1 + A
\]

8.23 Remark. The astute reader will have noticed that the constructor names of the \texttt{Maybe} type have been lost in the ornamentation. In the present system, its constructors are \texttt{false} : \texttt{Maybe} \; A and \texttt{true} : \( A \rightarrow \texttt{Maybe} \; A \). In practice, we want to be able to change the name of constructors during ornamentation. To do so, one could define a universe of ornaments specialised to tagged descriptions: the original type being in constructor-form, one would ask for an enumeration of constructor tags of the same size (and therefore, in bijection) to declare the constructors of the ornamented type.

8.24 Remark (Notation). We shall overload the interpretation of ornaments \( [[ - ]_{\text{orn}}] \) to denote both the description and the interpretation of that description. For instance, we write the above isomorphism as follows:

\[
[[\text{MaybeO} \; A]_{\text{orn}}] \; X] \cong 1 + A
\]

8.25 Example (Ornamenting natural numbers to lists). We obtain lists from natural numbers with the following ornament:

\[
\text{data} \; \text{List} \left[ A : \text{SET} \right] \; \text{from} \; \text{Nat} \; \text{where} \\
\text{List} \; A \; \ni \; \text{nil} \\
\text{cons} \left[ a : A \right] \left( as : \text{List} \; A \right)
\]

\[
\text{ListO} \left( A : \text{SET} \right) : \; \text{orn} \; \text{NatD} \; \text{id} \; \text{id} \\
\text{ListO} \; A \; \mapsto \; \lambda* \cdot \Sigma \left\{ \begin{array}{l}
0 \mapsto 1 \\
\text{suc} \mapsto \text{insert} \; A \; \lambda \cdot \text{var}
\end{array} \right\}
\]

We check that we obtain the signature functor of lists:

\[
[[\text{ListO} \; A]_{\text{orn}}] \; X] \cong 1 + A \times X
\]
8.26 Example (Ornamenting natural numbers to finite sets). We obtain finite sets by inserting a number \( n' : \text{Nat} \), constraining the index \( n \) to \( \text{suc} \ n' \), and – in the recursive case – indexing at \( n' \):

\[
\text{data } \text{Fin} \ (n : \text{Nat}) \text{ from } \text{Nat} \text{ where }
\begin{align*}
\text{Fin} \ n & \equiv f_0 \ [n' : \text{Nat}] | q : n = \text{suc} \ n' \\
& | \ fsuc \ [n' : \text{Nat}] | q : n = \text{suc} \ n' (k : \text{Fin} \ n')
\end{align*}
\]

\[\}

\[
\text{FinO} \ : \ \text{orn } \text{NatD} \ (\lambda n. \ast) (\lambda n. \ast) \text{ where }
\begin{align*}
\text{FinO} & \mapsto \lambda n. \text{insert } \text{Nat} \lambda n'.
\end{align*}
\]

\[
\begin{align*}
\Sigma & \begin{cases}
\cdot 0 & \mapsto 1 \\
\cdot \text{suc} & \mapsto \text{var } n'
\end{cases}
\end{align*}
\]

Again, the reader will check that this is indeed describing the signature of finite sets.

8.27 Example (Canonical lifting). A close inspection of the canonical lifting (Definition 5.23) reveals that it is a mere ornament of \( D \). We can define it by:

\[
\square_{(\text{Desc } K)} \ (xs : \llbracket D \rrbracket X) : \text{Orn } D \ \pi_0
\]

\[
\begin{align*}
\square_1 & \ast \mapsto 1 \\
\square_{\text{var } k} & x \mapsto \text{var } (\text{inv } (k, x)) \\
\square_{A \times B} & (a, b) \mapsto \square_A a \times \square_B b \\
\square_{E T} & (e, xs) \mapsto \text{delete } e (\square_{\text{switch } T e} xs) \\
\square_{\Sigma ST} & (s, xs) \mapsto \text{delete } s (\square_{\text{T s}} xs) \\
\square_{\Pi ST} & f \mapsto \Pi_{\text{As}}. \square_{\text{T s}} (f s)
\end{align*}
\]

This lifts pointwise to indexed descriptions, giving an ornament of type:

\[
\square_{(\text{func } K L)} : \text{orn } D \ (\pi_0 : (k : K) \times X k \to K) \ (\pi_0 : (l : L) \times \llbracket D \rrbracket X l \to L)
\]

We obtain the desired predicate transformer by first interpreting the ornament into a description, followed by the interpretation of the description into an indexed functor. To avoid clutter, we shall conflate notations: we write \( \square_D \) for the ornament, the corresponding description, and the resulting predicate transformer.

8.1.1. Brady’s optimisations, internally

MODEL: Chapter8.Brady

(8.28) In Remark 5.41, we introduced Brady’s forcing and detagging optimisations. We have explained how, thanks to our definition of descriptions as functions, we could (manually) craft datatypes in this form – such as detagged vectors or forced finite sets. Instead of a Henry Ford presentation, we gave an equivalent but less redundant definition. Seen as a datatype transformation, this operation is (obviously) structure-
8. Ornaments

In fact, these transformations are an instance of ornamentation. The key ingredient is the delete code that lets us delete parts of a definition using a witness extracted from the index. The deletion ornament lets us internalise the forcing and detagging optimisations into type theory. We can therefore craft our own Brady-optimised datatypes and benefit from them as early as at type checking.

8.29 Example (Detagging, by ornamentation). Our earlier definition of vectors (Example 3.47) mirrors Agda’s convention of constraining indices with equality. Our definition of ornaments lets us define a version of Vec that does not store its indices. Indeed, we can describe Vec by an ornament that matches the index \( n \) to determine which constructor to offer:

\[
\begin{align*}
\text{data } \text{Vec'} [A : \text{SET}] (n : \text{Nat}) \text{ from } \text{Vec} A n \text{ where} \\
\text{Vec'} A n & \equiv \text{Nat-case } n \\
\text{Vec'} A 0 & \ni \text{nil} \\
\text{Vec'} A (\text{suc } n) & \ni \text{cons } [n' \triangleq n] (a : A) (\text{vs } \text{Vec'} A n)
\end{align*}
\]

Such a definition was unavailable in the original presentation [McBride, 2013]. We have internalised detagging (Definition 5.43): the constructors of the datatype are determined by the index.

8.30 Example (Forcing, by ornamentation). The definition of finite sets given in Example 3.47 is also subject to an optimisation: by matching the index, we can avoid the duplication of \( n \) by deleting \( n' \) with the matched predecessor and trivialising the proofs. Hence, Fin can be further ornamented to the optimised Fin', which makes crucial use of deletion:

\[
\begin{align*}
\text{data } \text{Fin'} (n : \text{Nat}) \text{ from } \text{Fin} n \text{ where} \\
\text{Fin'} 0 & \ni [b : 0] \quad \text{– no constructor} \\
\text{Fin'} (\text{suc } n) & \ni \text{f0 } [n' \triangleq n] \\
& \mid \text{fsuc } [n' \triangleq n] (k : \text{Fin'} n')
\end{align*}
\]

Again, this definition was previously unavailable to us. We are making crucial use of the deletion ornament to avoid duplication. We have internalised forcing (Definition 5.42): the content of the constructors – here \( n' \) – is retrieved from the index, instead of being needlessly duplicated.

8.1.2. Ornamental algebra

MODEL: Chapter8.Ornament.Algebra

Following McBride [2013, §4], every ornament induces an ornamental algebra: an algebra that forgets the extra information introduced by the extensions, mapping the ornamented datatype back to its original form.

8.32 Definition (Cartesian morphism). For an ornament \( O : \text{Orn } D u \), there is a function – actually, a natural transformation – projecting the ornamented functor down to the
8.1. Universe of Ornaments

original, unornamented functor:

\[
\begin{align*}
\text{forgetNT} \ (O: \text{Orn} D u) \ (xs: [O]_{\text{orn}} (X \circ u)) & : [D] X \\
\text{forgetNT} \ (\text{insert} S D^+) \ (s, xs) & \mapsto \text{forgetNT} \ (D^+ s) \ xs \\
\text{forgetNT} \ (\text{var} (\text{inv} i)) \ xs & \mapsto xs \\
\text{forgetNT} \ 1 \ * & \mapsto * \\
\text{forgetNT} \ (O_1 \times O_2) \ (t_1, t_2) & \mapsto (\text{forgetNT} O_1 t_1, \text{forgetNT} O_2 t_2) \\
\text{forgetNT} \ (\sigma O) \ (k, xs) & \mapsto (k, \text{forgetNT} (\text{switch} O k) \ xs) \\
\text{forgetNT} \ (\Sigma O) \ (s, xs) & \mapsto (s, \text{forgetNT} (O s) \ xs) \\
\text{forgetNT} \ (\Pi O) \ f & \mapsto \lambda s. \text{forgetNT} (O s) (f s) \\
\text{forgetNT} \ (\text{delete} s O) \ xs & \mapsto (s, \text{forgetNT} O xs)
\end{align*}
\]

This function then lifts pointwise to a function on ornaments:

\[
\text{forgetNT}: (o: \text{orn} D u v) \rightarrow [o] (X \circ u) \rightarrow [D \circ v] X
\]

8.33 Remark. The choice of terminology is not innocent here: as we shall see in Section 8.2, this construction is related to the notion of Cartesian morphism of containers. These morphisms arise themselves from a fibration (Remark 8.71).

8.34 Definition (Ornamental algebra). Applied with \(\mu D\) for \(X\) and post-composed with the initial algebra \(\text{in}\), this Cartesian morphism induces the ornamental algebra:

\[
\begin{align*}
\text{forgetAlg} \ (o: \text{orn} D u u) \ (xs: [o] (\mu D \circ u) i) & : (\mu D \circ u) i \\
\text{forgetAlg} o \ xs & \mapsto \text{in} (\text{forgetNT} o xs)
\end{align*}
\]

8.35 Definition (Forgetful map). In turn, this algebra induces a forgetful map from the ornamented type to its original form:

\[
\begin{align*}
\text{forget} \ (o: \text{orn} D u u) & : \mu [o]_{\text{orn}} \rightarrow \mu D \circ u \\
\text{forget} o & \mapsto (\text{forgetAlg} o)
\end{align*}
\]

8.36 Example (From lists back to natural numbers). Applied to the ornament \(\text{ListO}\), the Cartesian morphism removes the extra information added through \(\text{insert}\), \(i.e.\) the inhabitant of \(A\). The resulting algebra thus takes \(\text{nil}\) to \(0\), and \(\text{cons} a\) to \(\text{suc}\). In turn, the forgetful map computes the length of the list.

8.37 Example (From finite sets back to natural numbers). Applied to the ornament \(\text{FinO}\), the Cartesian morphism removes the equations introduced by \(\text{insert}\) and forgets the indexing discipline enforced by the \(\text{var}\) code. The resulting forgetful map computes the cardinality – a natural number – of a finite set.
8. Ornaments

8.1.3. Algebraic ornaments

MODEL: Chapter8.AlgebraicOrnament

(8.38) An important class of datatypes is constructed by algebraic ornamentation over a base datatype. An algebraic ornament indexes an inductive type by the result of a catamorphism over its elements. From the code \( D : \text{func} \ K \ K \) and an algebra \( \alpha : \left[ D \right] X \rightarrow X \), we define an ornament \( D^\alpha \) indexed by \( (k : K) \times X \) satisfying the following coherence property:

\[
\mu D^\alpha (k, x) \cong (t : \mu D k) \times (\alpha t) \overset{\sim}{=} x
\]

Seen as a set comprehension, this states that \( \mu D^\alpha (k, x) \) is an inductive definition equivalent to the refinement type \( \{ t \in \mu D k \mid (\alpha t) \overset{\sim}{=} x \} \). A categorical presentation that explores the connection with refinement types is given by Atkey et al. [2012].

8.39 Example (Algebraic ornament: interval). For a given natural number \( m : \text{Nat} \), the addition \( m + - : \text{Nat} \rightarrow \text{Nat} \) is definable by folding the algebra:

\[
\begin{align*}
\text{add}_m (x n : [\text{NatD} \text{Nat}]) & : \text{Nat} \\
\text{add}_m \ ['0'] & \mapsto m \\
\text{add}_m \ ['\text{suc} \ n'] & \mapsto \text{suc} \ n
\end{align*}
\]

By algebraically ornamenting \( \text{Nat} \) by this algebra, we obtain the type of intervals \([m, -] : \text{Nat} \rightarrow \text{SET} \) that is characterised by the isomorphism:

\[
[m, n] \cong (k : \text{Nat}) \times (m + k = n)
\]

Put explicitly, the datatype computed by the algebraic ornament corresponds to

\[
data [[m : \text{Nat}], \text{-}] (n : \text{Nat}) : \text{SET} \text{ where} \\
[m, \text{-}] \quad (n = m) \quad \supseteq \ 0 \\
[m, \text{-}] \quad (n = \text{suc} \ n') \quad \supseteq \ \text{suc} \ (k : [m, n'])
\]

(8.40) The type-theoretic construction of \( D^\alpha \) was originally given by McBride [2013, §5]. We adapt it to our universe of types. The idea is to define – by ornamentation of \( D \) – a description whose fixpoint will satisfy the above coherence property. This ornament is called an algebraic ornament.

8.41 Definition (Algebraic ornament). The algebraic ornament of \( D \) by \( \alpha \) is a refinement of the former by the latter: for an index \((k, x)\), the algebraic ornament is built from any \( xs : [D] X k \), as long as we have \( \alpha xs = k \). We therefore insert such an \( xs \) and enforce the coherence (locally) by an equality constraint. We then use the canonical lifting of \( D \) at
index $xs$ to create a $D$-structure that follows the indexing discipline set by $xs$:

$$
\begin{align*}
(D : \text{func } K K) (\alpha[D] X \xrightarrow{x}) & : \text{orn } D \tau_0 \tau_0 \\
D^\alpha & \mapsto \lambda (k, x). \text{insert } ([D] X k) \lambda x s. \\
\text{insert } (\alpha xs = x) & \lambda - . \\
\square_D (k, xs)
\end{align*}
$$

**8.42 Remark.** In Section 8.3.2, we relate this type-theoretic definition with the categorical one given by Atkey et al. [2012]. We shall see that the two notions are in exact correspondence.

**8.43 Remark (Notation).** We shall indiscriminately write $D^\alpha$ to refer to the ornament and the resulting description.

**8.44 Example (Algebraic ornament: vectors).** Ornamenting natural numbers to lists, we obtain an ornamental algebra: the algebra computing the length of a list. We can therefore build the algebraic ornament of lists by the length algebra. This corresponds exactly to the datatype of vectors (Example 5.36).

Note that this operation generalises to all ornaments: any ornament induces an ornamental algebra. We can always build the algebraic ornament by the ornamental algebra. We shall study this operation in more details in Section 8.1.4.

**8.45 Example (Algebraic ornament: lifting).** As noticed by McBride [2013, §7], the canonical lifting of a description $D$ corresponds to $D^\text{in}$. Note that $\text{in}$ is an isomorphism (by Lambek’s lemma 4.35). The definition of $D^\text{in}$ does indeed reduce to $\square_D$: the inserted equality constraint being trivialised by the isomorphism. We shall come back to this point through a categorical angle in Section 8.3.2.

**8.46 Example (Algebraic ornament: indexing by semantics).** A typical use-case of algebraic ornaments is the implementation of semantic-preserving operations on syntax trees [McBride, 2013, §9]. For example, let us consider arithmetic expressions, whose semantics is given by interpretation in $\text{Nat}$:

```haskell
data Expr : Set where  
  const : n : Nat  \mapsto  Nat  
  add : d : Expr \mapsto  Nat  \mapsto  Nat  
            e : Expr \mapsto  Nat  

\begin{align*}
\text{evalAlg} (es : [\text{Expr-Desc}] \text{Nat}) & : \text{Nat} \\
\text{evalAlg} [\text{const } n] & \mapsto  n \\
\text{evalAlg} [\text{add } m n] & \mapsto  m + n
\end{align*}
```

Using the algebra $\text{evalAlg}$, we construct the algebraic ornament of $\text{Expr}$ and obtain expressions indexed by their semantics:

```haskell
data Expr^evalAlg (k : Nat) : Set where  
  Expr^evalAlg (k = n)  \mapsto  const (n : Nat) 
  Expr^evalAlg (k = m + n)  \mapsto  add (m : Nat) (d : Expr^evalAlg m) (e : Expr^evalAlg n)
```

We can now enforce the preservation of semantics by typing. For example, let us
optimise away all additions of the form \(0 + e\):

\[
\begin{align*}
optimise-0+ (e : \text{Expr}^{\text{evalAlg}} n) & : \text{Expr}^{\text{evalAlg}} n \\
optimise-0+ (\text{const } n) & \mapsto \text{const } n \\
optimise-0+ (\text{add } 0 \ n \ (\text{const } 0 \ e)) & \mapsto \optimise-0+ e \\
optimise-0+ (\text{add } m \ n \ d \ e) & \mapsto \text{add } m \ n \ d \ e
\end{align*}
\]

If the type checker accepts our definition, we have that, by construction, this operation preserves the semantics. We can then prune the semantics from the types using the forgetful map and retrieve the transformation on raw syntax trees. The \texttt{coherentOrn} theorem certifies that the pruned tree satisfies the invariant we enforced by indexing.

(8.47) Read constructively, the coherence property (§ 8.38) corresponds to two mutually inverse functions. From left to right, we obtain a proposition stating that forgetting an algebraic ornament gives a value that satisfies the algebraic predicate. From right to left, we have a function that builds an inhabitant of the algebraic ornament from an inhabitant of the base type, the index being computed by the algebraic predicate. These constructions arise naturally from the definition of algebraic ornaments.

(8.48) The direction \(\mu D^a(i, x) \rightarrow (t : \mu D \ i) \times \langle a \rangle \ t = x\) relies on the generic \texttt{forget} \(D^a\) function to compute the first component of the pair, and gives us the following theorem:

\[
\texttt{coherentOrn} : \forall t^a : \mu D^a(i, x). \langle a \rangle (\text{forget } D^a t^a) = x
\]

This corresponds to the \texttt{Recomputation} theorem of McBride [2013, §8].

(8.49) **Computational interpretation.** From an inhabitant of \(D^a\), we can extract its computational component with \texttt{forget}. Doing so, we forget the indexing (i.e. logical) information. The coherence theorem tells us that the resulting datatype satisfies the algebraic predicate that \(D^a\) was enforcing.

\textbf{8.50 Example} (Indexing by semantics). Applied to Example 8.46, we implement our semantics-preserving operation on the algebraic ornament, hence obtaining its soundness by construction. However, we are only interested in the operation of the non-indexed datatype, which we obtain by \texttt{forget}. The coherence theorem tells us that the non-indexed result is semantically valid.

(8.51) In the other direction, the isomorphism gives us a function of type:

\[
(t : \mu D \ i) \times \langle a \rangle t = x \rightarrow \mu D^a(i, x)
\]

Put in full and simplifying the equation, this corresponds to a function:

\[
D^a\text{-make} : (t : \mu D \ i) \rightarrow \mu D^a(i, \langle a \rangle t)
\]

This corresponds to the \texttt{remember} function of McBride [2013, §7].

(8.52) **Computational interpretation.** This operation lets us lift an inhabitant of \(D\) to its
algebraic refinement $D^\alpha$. Computationally, this function is an identity function: its role is purely logical, unfolding the index-level computation $\langle \alpha \rangle$ across the inhabitant of $D$.

8.1.4. Reornaments

MODEL: Chapter8.Reornament

In the next chapter, we are particularly interested in a special subclass of algebraic ornaments. As hinted at in Example 8.44, every ornament $o$ induces an ornamental algebra $\text{forgetAlg}\, o$, which forgets the extra information introduced by the ornament. Hence, given a datatype $D$ and an ornament $o_D$ of $D$, we can algebraically ornament $\llbracket o_D \rrbracket$ using the ornamental algebra $\text{forgetAlg}\, o_D$. McBride [2013, §6] calls this construction the algebraic ornament by the ornamental algebra.

8.54 Remark (Notation). We shall denote an ornament of $D$ by $o_D$ as $\llbracket o_D \rrbracket$. For brevity, we call it the reornament of $o_D$. As usual, we write $\llbracket o_D \rrbracket$ to denote both the ornament and the resulting description.

8.55 Example (Reornament: vectors). Following Example 8.44, a standard example of reornament is $\text{Vec}$: it is the reornament of $\text{ListO}$. Explicitly, a vector is the algebraic ornament of $\text{List}$ by the algebra computing its length, i.e. the ornamental algebra from $\text{List}$ to $\text{Nat}$. This corresponds to the definition of vectors by equality constraints:

```haskell
data Vec [A : SET](n : Nat) : SET where 
  Vec A (n = 0) ⊰ nil 
  Vec A (n = suc n') ⊰ cons (n' : Nat)(a : A)(vs : Vec A n')
```

8.56 Example (Reornament: indexed option type). In Example 8.22, we ornamented Booleans to the option type. We can thus reornament the option type with its Boolean status. Unfolding the definition of the reornament, we obtain the datatype:

```haskell
data IMaybe [A : SET](b : Bool) : SET where 
  IMaybe A (b = true) ⊰ just (a : A) 
  IMaybe A (b = false) ⊰ nothing
```

This datatype comes – by folding its ornamental algebra – with the forgetful map:

- $\text{forgetIMaybe} (\text{just} \, a) \mapsto (\text{just} \, a, \text{refl})$
- $\text{forgetIMaybe} \, \text{nothing} \mapsto (\text{nothing}, \text{refl})$

Reornaments are thus straightforwardly obtained through a two steps process: first, compute the ornamental algebra and, second, construct the algebraic ornament by this algebra. However, such a simplistic construction introduces a lot of spurious equality constraints and duplication of information. For instance, using this naive definition of reornaments, a vector indexed by $n$ is constructed as any list as long as it is of length $n$. 

195
Ornaments

Example (Reornamenting vectors, efficiently). We can adopt a more fine-grained approach yielding an isomorphic but better structured datatype. In our setting, where we can compute over the index, a finer construction of the Vec reornament is as follows:

- We retrieve the index, hence obtaining a number \( n : \text{Nat} \);
- By inspecting the ornament ListO, we obtain the exact information by which \( n \) is extended into a list: if \( n = 0 \), no supplementary information is needed and if \( n = \text{suc} \ n' \), we need to extend it with an \( a : A \). We call this the extension – denoted Extension – of ListO at \( n \);
- By inspecting the ornament ListO again, we obtain the recursive structure of the reornament by deleting the data already determined by the index and its extension, only to extract the refined indexing discipline: the tail of a vector of size \( \text{suc} \ n' \) is a vector of size \( n' \). The recursive structure is denoted Structure.

Let us generalise this construction to any ornament.

Definition (Reornament extension). By the coherence property, we know that for any index \( t : \mu \ D \), the reornament \( t^+ : \mu \lceil o_D \rceil \ t \) is isomorphic to the comprehension

\[
\{ t^+ : \mu \lceil o_D \rceil \ t \mid \text{forget } o_D \ t^+ = t \}
\]

Note that the equality constraints are introduced only to ensure that \( t^+ \) is in the inverse image of the forgetful map at \( t \). In our setting, we can enforce these constraints by construction: from the ornament \( o_D \) and the index \( t \), we can compute the set of valid extensions of \( t \) giving a \( t^+ \) in the inverse image of the forgetful map:

\[
\begin{align*}
\text{Extension } (O: \text{Orn } D \ u) (xs : [D] \ \mu \ D) & : \text{SET} \\
\text{– Do not duplicate the original data, it is already in } xs: \\
\text{Extension } (\var \ (\text{inv } i)) & t \mapsto 1 \\
\text{Extension } 1 & \ast \mapsto 1 \\
\text{Extension } (\Pi \ T^+) & f \mapsto (s : S) \mapsto \text{Extension } (T^+ s) (f s) \\
\text{Extension } (A^+ \times B^+) & (a, b) \mapsto \text{Extension } A^+ a \times \text{Extension } B^+ b \\
\text{Extension } (\sigma \ T^+) & (e, xs) \mapsto \text{Extension } (\text{switch } T^+ e) \ xs \\
\text{Extension } (\Sigma \ T^+) & (s, xs) \mapsto \text{Extension } (T^+ s) \ xs \\
\text{– Ask for freshly inserted data, it is the data missing from } xs: \\
\text{Extension } (\text{insert } S \ D^+) & xs \mapsto (s : S) \times \text{Extension } (D^+ s) \ xs \\
\text{– Deleted data must be consistent with } xs: \\
\text{Extension } (\text{delete } S \ T^+) & (s', xs) \mapsto (q : s = s') \times \text{Extension } T^+ \ xs
\end{align*}
\]

The next step consists in building the recursive structure of the reornamented type. Again, the recursive structure is entirely described by the ornament and the index: the ornament gives the recursive structure of \( t^+ \), while the index \( t \) specifies the indexing strategy of the subnodes: subnodes of \( t^+ \) must be indexed by the corresponding subnodes of \( t \).

Definition (Reornament structure). We obtain the recursive structure of \( t^+ \) by traversing the ornament definition while unfolding the index \( t \) along the way in order to reach its subnodes. On a variable, we index by the value specified by the ornament and
8.1. Universe of Ornaments

the subnode of \( t \) we have reached. We can delete \( \Sigma \) and insert codes to avoid information duplication: the information is already provided by the index in the case of \( \Sigma \), and by the extension in the case of insert. We proceed as follows:

\[
\text{Structure } (O: \text{Orn } D u) (xs: \mu D) (e: \text{Extension } O \hspace{1em} xs) : \text{Orn } \mu [O] \pi_0
\]

\[\quad \text{Extract index from the ornament and the index:}\]
\[
\text{Structure } (\text{var } (\text{inv } i)) \hspace{1em} t \quad * \quad \mapsto \text{var } (\text{inv } (i, t))
\]

\[\quad \text{Duplicate only the recursive structure:}\]
\[
\text{Structure } \ast \ast \mapsto 1
\]
\[
\text{Structure } ([T^+] f) \hspace{1em} ext \quad \mapsto \Pi \lambda s. \text{Structure } (T^+ s) (f s) (\text{ext } s)
\]
\[
\text{Structure } ((a, b) (\text{ext } a, \text{ext } b)) \quad \mapsto \text{Structure } (a \text{ ext } a) \times \text{Structure } (b \text{ ext } b)
\]
\[
\text{Structure } (\sigma T^+) (e, xs) \quad \text{ext} \quad \mapsto \text{delete } e (\text{Structure } (\text{switch } T^+ e) \hspace{1em} xs \hspace{1em} \text{ext})
\]
\[
\text{Structure } (\Sigma T^+) (s, xs) \quad \text{ext} \quad \mapsto \text{delete } s (\text{Structure } (T^+ s) \hspace{1em} xs \hspace{1em} \text{ext})
\]
\[
\text{Structure } (\text{insert } S D^+) \hspace{1em} xs \quad (s, \text{ext}) \quad \mapsto \text{delete } s (\text{Structure } (D^+ s) \hspace{1em} xs \hspace{1em} \text{ext})
\]
\[
\text{Structure } (\text{delete } T^+) (s, \hspace{1em} xs) \quad (\text{refl, ext}) \quad \mapsto \text{Structure } T^+ (\hspace{1em} xs \hspace{1em} \text{ext})
\]

8.62 Definition (Reornament). A reornament is thus the Extension of its index followed by the arguments specified by its recursive Structure. We define the reornament at index \( t = \in xs: \mu D \) by inserting the valid extensions of \( t \), followed by its recursive structure:

\[
\text{reornament } (o: \text{orn } D u) : \text{orn } [o] \pi_0 \pi_0
\]
\[
\text{reornament } o \mapsto \lambda (i, in \hspace{1em} xs). \text{insert } (\text{Extension } (o \hspace{1em} i) \hspace{1em} xs) \hspace{1em} \lambda e.
\]

8.63 Example (Reornament: vectors). Applied to the ornament ListO (Example 8.55), this construction gives the fully Brady-optimised – detagged and forced – version of Vec corresponding to the definition:

\[
\text{data } \text{Vec } [A: \text{SET}] (n: \text{Nat}) : \text{SET} \text{ where}
\]
\[
\hspace{1em} \text{Vec } A \quad 0 \quad \ni \text{nil}
\]
\[
\hspace{1em} \text{Vec } A \quad (\text{suc } n) \ni \text{cons } (a: A)(vs: \text{Vec } A \hspace{1em} n)
\]

That is, we determine which constructor of Vec is available by pattern-matching on the index. This is unlike the naive reornament, which relies on constraints to enforce the indexing discipline.

8.64 Example (Reornament: indexed option type). Under this definition, the reornament of MaybeO (Example 8.56) corresponds to the definition

\[
\text{data } \text{IMaybe } [A: \text{SET}] (b: \text{Bool}) : \text{SET} \text{ where}
\]
\[
\hspace{1em} \text{IMaybe } A \quad \text{true} \ni \text{just } (a: A)
\]
\[
\hspace{1em} \text{IMaybe } A \quad \text{false} \ni \text{nothing}
\]

where, similarly, constraints are off-loaded to computations on indices.

(8.65) Note that our ability to compute over indices is crucial for this construction to work.
8. Ornaments

Also, the datatypes we obtain are isomorphic to the datatypes one would have obtained by the algebraic ornament of the ornamental algebra. Consequently, the correctness property of algebraic ornaments is still valid: constructively, we get the \textit{coherentOrn} theorem in one direction and the $[\alpha_D] \cdot \text{make}$ function in the other.

8.66 Remark (Iterating reornamentation). Every ornament induces a reornament. A reornament is itself an ornament: it therefore induces yet another reornament. We are naturally led to wonder if this process ever stops, and if so when. For example, the ornament of natural numbers into lists reornaments to vectors. Reornamenting vectors, we obtain an inductive predicate representing the length function $\text{Length : Nat} \to \text{List } A \to \text{SET}$. Reornamenting $\text{Length}$ leads to an object with no computational content: all its information has been erased and is provided by the indices.

The same pattern arises in general: every chain of reornaments is bound to terminate on a computationally trivial object. We deduce this from the definition of reornaments based on \textit{Extension} and \textit{Structure}. We proceed by case analysis on the ornament. On a $1$, $\times$, $\Pi$, and \textit{var} code, the reornamentation proceeds purely structurally, hence introducing no information. The reornamentation \textit{deletes} $\Sigma$ codes, using the index information instead: this removes the information contained in the datatype. On a delete code, the reornament inserts an equality constraint, which contains no information per se: it is only enforcing the indexing discipline. Only on an insert code does the reornament introduce new information through a $\Sigma$ code.

In the next iteration of the reornament, the insert codes of the ornament (e.g. list from nat) turn into $\Sigma$ codes in its reornament (e.g. vector). In the subsequent iteration, these $\Sigma$ codes in the reornament are in turn deleted by the re-reornament (e.g. the \textit{Length} predicate). In the third iteration, there is nothing left in the code but equations: the resulting datatype is computationally trivial and is entirely determined by its indices.

In this section, we have adapted ornaments to our universe of datatypes. In doing so, we have introduced deletion ornaments, which use the indexing to remove duplicated information from the datatypes. This has proved useful to simplify the definition of reornaments. We shall see in Chapter 9 how this can be turned to our advantage when we transport functions across ornaments.

8.2. Categorical Semantics of Ornaments

In Chapter 5, we gave a presentation of inductive types in type theory and showed that these objects can faithfully be modelled by containers. In the previous section, we have adapted the notion of ornament to our theory of inductive types. A natural question is then whether ornaments correspond to a categorical entity in the theory of containers. As it turns out, they do: \textit{ornaments are Cartesian morphisms of containers}. In this section, we substantiate this claim.
8.2. Categorical Semantics of Ornaments

8.2.1. Cartesian morphisms of containers

MODEL:  
Chapter8.Container.Morphism  
Chapter8.Container.BaseChange  
Chapter8.Container.CobaseChange

In Section 5.3, we have recalled the definition of containers (Definition 5.53). In this chapter, we move on to studying the morphisms between containers, focusing our effort on the so-called Cartesian morphisms [Abbott et al., 2005, Gambino and Kock, 2013]. As we shall see in Section 8.2.2, Cartesian morphisms provide a categorical model for ornaments.

8.70 Definition (Cartesian morphism of containers). Let \( \text{Op}' \downarrow \text{Sort}' \downarrow \text{Ar}' : \text{ICont} I J \) and \( \text{Op} \downarrow \text{Sort} \downarrow \text{Ar} : \text{ICont} K L \) be two containers, indexed respectively by \((I, J)\) and \((K, L)\). Let \( u : I \to K \) and \( v : J \to L \) be two functions mapping the indices of the former to the latter.

A Cartesian morphism from \( \text{Op}' \downarrow \text{Sort}' \downarrow \text{Ar}' \) to \( \text{Op} \downarrow \text{Sort} \downarrow \text{Ar} \) framed by \( u \) and \( v \) corresponds to a triple

\[
\begin{align*}
\omega &: \text{Op}' j \to \text{Op} (v j) \\
\rho &: \forall \text{op}' : \text{Op}' j. \text{Ar} (\omega \text{op}') = \text{Ar}' \text{op}' \\
q &: \forall \text{op}' : \text{Op}' j. \forall \text{ar} : \text{Ar} (\omega \text{op}'). u (\text{Sort}' \text{ar}) = \text{Sort} \text{ar}
\end{align*}
\]

where \( \omega \) maps the operations of \( \text{Op}' \downarrow \text{Sort}' \downarrow \text{Ar}' \) to operations of \( \text{Op} \downarrow \text{Sort} \downarrow \text{Ar} \), while \( \rho \) enforces that the operations mapped through \( \omega \) have the same arity, and \( q \) enforces that the sorts of their arguments are related through \( u \). In general, we specify a Cartesian morphism by only giving its action on operations \( \omega \), leaving it to the reader to verify that the side-conditions \( \rho \) and \( q \) are satisfied.

Following 2-categorical conventions [Shulman, 2008], the hom-set of Cartesian morphisms (i.e. 2-cell) from \( \text{Op}' \downarrow \text{Sort}' \downarrow \text{Ar}' \) to \( \text{Op} \downarrow \text{Sort} \downarrow \text{Ar} \) framed by \( u \) and \( v \) is denoted

\[ \text{Op}' \downarrow \text{Sort}' \downarrow \text{Ar}' \Rightarrow_{u,v} \text{Op} \downarrow \text{Sort} \downarrow \text{Ar} \]

8.71 Remark. As suggested by the denomination of “Cartesian morphism”, these morphisms play a particular role in some fibration. We may think of the polynomial functors indexed by \( I \) and \( J \) as defining a subcategory of \([\text{SET}^I, \text{SET}^J]\) [Gambino and Kock, 2013, Proposition 2.4]. The category \( \text{SET}^I \) being pullback-complete, and \( \text{SET}^I \) having a terminal object \( 1 \), we may consider the fibration

\[
\begin{CD}
[\text{SET}^I, \text{SET}^I] @>>> 1 \\
\text{SET}^I \end{CD}
\]

The Cartesian morphisms we have just defined correspond exactly to the Cartesian natural transformations – natural transformations whose naturality square forms a
8. Ornaments

pullback – in the fibration [Gambino and Kock, 2013, Theorem 3.8]. This also suggests that there exists a more general notion of morphism of containers [Abbott et al., 2005, Gambino and Kock, 2013], which we shall not study in this thesis.

8.72 Example (From lists to natural numbers). We build a Cartesian morphism from ListCont_A (Example 5.56) to NatCont (Example 5.55) by mapping operations of ListCont_A to operations of NatCont:

\[ \omega : \text{ListCont}_A \rightarrow \text{NatCont} \quad \text{where} \]

\[ \omega(\text{op}_j : \text{Op}_{\text{List}} \ast) : \text{Op}_{\text{Nat}} \ast \]

\[ \omega(\text{inj}_j \ast) \mapsto \text{inj}_j \ast - \text{nil to } 0 \]

\[ \omega(\text{inj}_j a) \mapsto \text{inj}_j \ast - \text{cons } a \text{ to suc} \]

We are then left to check that arities are isomorphic: this is indeed true, since, in the nil/0 case, the arity is nil while, in the cons/suc case, the arity is one. The coherence condition is trivially satisfied, since both containers are indexed by 1. We shall relate this natural transformation to the function computing the length of a list in Example 8.93.

8.73 Definition (Interpretation of container morphism). A Cartesian container morphism interprets to a natural transformation from the first container to the second, simply mapping operations \(\text{Op}'\) to operations \(\text{Op}\) covariantly using \(\omega\):

\[ [(m : \text{Op}' \triangleleft \text{Sort}' \text{Ar}') \mapsto \text{op} \triangleleft \text{Sort} \text{Ar}] \in \text{Cont}_I \quad \text{where} \]

\[ \omega \circ \text{op} \triangleleft \text{Sort} \text{Ar} \]

\[ \omega \circ (\text{inj}_j \ast) \mapsto \text{inj}_j \ast - \text{nil to } 0 \]

\[ \omega \circ (\text{inj}_j a) \mapsto \text{inj}_j \ast - \text{cons } a \text{ to suc} \]

(8.74) For a pair of indices \(I, J : \text{SET}\), we can – quite unsurprisingly – organise \(I\text{Cont}_I J\) into a category [Abbott, 2003], with the morphisms defined to be the Cartesian morphisms framed by the identity (\(u \equiv \text{id}_I\) and \(v \equiv \text{id}_J\)).

More interestingly, we can organise \(I\text{Cont}\) itself into a 2-category [Gambino and Kock, 2013]. The objects are the indices, the category of 1-morphisms between indices \(I\) and \(J\) is the category \(I\text{Cont}_I J\). Thus, the 2-cells are the Cartesian morphisms between the containers indexed by \(I\) and \(J\).

However, this fails to capture the fact that indices have a life on their own: it makes sense to have morphisms between differently indexed functors. Indeed, morphisms between indices – the objects – induce 1-morphisms. Gambino and Kock [2013] have shown that polynomials (thus, containers) and their interpretations can be organised into framed bicategories [Shulman, 2008].

Besides, they have shown that the interpretation functor (Definition 5.58 and Definition 8.73) is an equivalence of framed bicategories [Gambino and Kock, 2013, Proposition 3.14]. We can therefore legitimately conflate containers and polynomial functors, the two notions being equivalent.

(8.75) Frame structure. By working with framed bicategories, we gain access to two (dual) operations that manipulate indexing of containers, the base-change and cobase-change [Gambino and Kock, 2013, §3.10]. Let us give their definition in the theory of containers.
8.76 Definition (Base-change container). Let $\text{Op} \triangleq \text{Sort} \, \text{Ar}$ be a container indexed by $K$ and $L$. Let $u : K \to I$ and $v : L \to J$ be two functions mapping $K$ and $L$ respectively to $I$ and $J$.

The base-change of $\text{Op} \triangleq \text{Sort} \, \text{Ar}$ by $u$ and $v$, written $(u, v)^* \text{Op} \triangleq \text{Sort} \, \text{Ar}$, is a container indexed by $I$ and $J$, defined as follows:

\[
(u, v)^* \text{Op} \triangleq \text{Sort} \, \text{Ar} \triangleq \begin{cases} 
\text{Op}^* (j : J) : \text{SET} \\
\text{Op}^* j \mapsto (l : v^{-1} j) \times \text{Op} l \\
\text{Ar}^* (op : \text{Op}^* j) : \text{SET} \\
\text{Ar}^* op \mapsto \text{Ar} (\pi_1 op) \\
\text{Sort}^* (ar : \text{Ar}^* op) : I \\
\text{Sort}^* ar \mapsto v (\text{Sort} ar)
\end{cases}
\]

(8.77) Intuition. In a sense, the base-change container $(u, v)^* \text{Op} \triangleq \text{Sort} \, \text{Ar}$ is the “closest” container to $\text{Op} \triangleq \text{Sort} \, \text{Ar}$ that is indexed by $I$ and $J$ instead of $K$ and $L$.

We can think of $u$ and $v$ as functions imposing a less precise indexing discipline since many $I$-indices may be mapped to the same $K$-index, and similarly from $J$ to $L$. The resulting base-changed container is thus describing a less finely-indexed data-structure.

8.78 Definition (Cobase-change container). Let $\text{Op} \triangleq \text{Sort} \, \text{Ar}$ be a container indexed by $I$ and $J$. Let $u : K \to I$ and $v : L \to J$ be two functions mapping $K$ and $L$ respectively to $I$ and $J$.

The cobase-change of $\text{Op} \triangleq \text{Sort} \, \text{Ar}$ by $u$ and $v$, written $(u, v)_! \text{Op} \triangleq \text{Sort} \, \text{Ar}$, is a container indexed by $K$ and $L$, defined as follows:

\[
(u, v)_! \text{Op} \triangleq \text{Sort} \, \text{Ar} \triangleq \begin{cases} 
\text{Op}_!(l : L) : \text{SET} \\
\text{Op}_! l \mapsto (op : \text{Op} (v l)) \times ((ar : \text{Ar} op) \mapsto (k : K) \times u k = \text{Sort} ar) \\
\text{Ar}_!(op : \text{Op}_! l) : \text{SET} \\
\text{Ar}_! op \mapsto \text{Ar} (\pi_0 op) \\
\text{Sort}_! (ar : \text{Ar}_! op) : K \\
\text{Sort}_! ar \mapsto \pi_0 ((\pi_1 op) ar)
\end{cases}
\]

(8.79) Intuition. While the base-change operation uses $u$ and $v$ to define a less precisely-indexed signature, the cobase-change has the opposite effect. Looking at $u$ from $K$ to $I$ this time, we can see the $I$-indices as more segregative than the $K$-indices, and similarly from $L$ to $J$.

The cobase-change container $(u, v)_! \text{Op} \triangleq \text{Sort} \, \text{Ar}$ is thus the more precisely-indexed container derived from $\text{Op} \triangleq \text{Sort} \, \text{Ar}$. To this end, we must carefully restrict its operations to the ones in the original container whose arities are preserved under the new (more segregative) indexing regime. An example of cobase-changed datatype is given in Example 8.109.

8.80 Remark. Interestingly, we can easily define a Cartesian morphism from a container to its base-change container and, dually, we have a Cartesian morphism from the cobase-change container to its original container. As we shall see in Section 8.3.3, this translates into a novel ornamental construction.
8. Ornaments

8.2.2. Ornaments are Cartesian morphisms

MODEL: Chapter8.Equivalence

(8.81) Relating the definition of ornaments (Definition 8.17) with our container-based reading of descriptions (Figure 5.4), we make the following remarks. Firstly, the ornament code lets us only insert – with the insert code – or delete – with the delete code – Σ codes, while forbidding deletion or insertion of either Π or var codes. In terms of containers, this translates to: operations can be extended, while arity must remain unchanged. Secondly, on the var code, the ornament code lets us pick any index in the inverse image of \( u \). In terms of container, this corresponds to the coherence condition: the initial indexing must commute with applying the ornamented indexing followed by \( u \).

(8.82) Concretely, for a container \( \text{Op} \langle \text{Sort} \rangle \text{Ar} \), an ornament can be modelled as an extension \( \text{ext} \), and a refined indexing \( \text{Sort}^+ \) subject to a coherence condition \( q \) with respect to the original indexing:

\[
\begin{align*}
\text{ext}: \text{Op}(v l) &\rightarrow \text{SET} \\
\text{Sort}^+: \text{ext op} &\rightarrow \text{Ar op} \rightarrow K
\end{align*}
\]

\[q: \forall \text{ar}: \text{Ar op}. u (\text{Sort}^+ e \text{ar}) = \text{Sort} \text{ar}\]

(8.83) Equivalently, the family of set \( \text{ext} \) can be understood as the inverse image of a function \( \omega: \text{Op}^+ l \rightarrow \text{Op}(v l) \). The function \( \text{Sort}^+ \) is then the arguments’ sort of a container with operations \( \text{Op}^+ \) and arities \( \text{Ar} \circ \omega \). Put otherwise, the morphism on operations \( \omega \) together with the coherence condition \( q \) form a Cartesian morphism from \( \text{Op}^+ \langle \text{Sort}^+ \rangle \text{Ar} \circ \omega \) to \( \text{Op} \langle \text{Sort} \rangle \text{Ar} \).

Example (The many faces of an ornament). To gain some intuition, the reader can revisit the Cartesian morphism of Example 8.72 as an ornament of container – by simply inverting the morphism on operations – and as an ornament of description – by relating it with the ornament ListO (Example 8.25).

We shall now formalise this argument by proving that ornaments and Cartesian morphisms are in fact the two sides of the same coin:

8.85 Proposition. Ornaments are an intensional characterisation of the Cartesian morphisms of containers, i.e. we have the isomorphism

\[
\text{orn} D u v \cong \text{ICont}(\text{\textsc{\-}}, \langle D \rangle)_{u,v}
\]

Proof. The first half of the isomorphism consists in mapping an ornament \( o \) of a description \( D \) to a Cartesian morphism from the container described by \( [o]_{\text{orn}} \) to the container described by \( D \). By definition of Cartesian morphisms, we simply have to give a map from the operations of \( [o]_{\text{orn}} \) to the operations of \( D \) (Figure 8.2).

We are then left to check (extensionally) that the arities are isomorphic and the indexing is coherent. This is indeed the case, even though proving it in type theory is cumbersome. The ornament does not introduce or delete any new \( \Pi \) or var: hence the arities of each operations are left unchanged. Concerning the sort of these arguments, we rely on \( u^{-1} k \) to ensure that the more precise indexing is coherent by construction.
8.2. Categorical Semantics of Ornaments

\( \phi (o : \text{orn } D) =^u \overset{v}u \) \( (D) \)

where

\[
\text{forget } (O : \text{Orn } D) (sh : \text{Opfunc } [O]_{\text{orn}}) : \text{Opfunc } D
\]

\[
\begin{align*}
\text{forget } (\text{insert } a D^+) & \quad (a, sh) \quad \mapsto \quad \text{forget } (D^+ a) sh \\
\text{forget } (\text{var } (\text{inv } j)) & \quad * \quad \mapsto \quad * \\
\text{forget } 1 & \quad * \quad \mapsto \quad * \\
\text{forget } (A^+ \times B^+) & \quad (a, b) \quad \mapsto \quad (\text{forget } A^+ a, \text{forget } B^+ b) \\
\text{forget } (\sigma T^+) & \quad (e, sh) \quad \mapsto \quad (e, \text{forget } (\text{switch } T^+ e) sh) \\
\text{forget } (\Pi T^+) & \quad f \quad \mapsto \quad \lambda a. \text{forget } (T^+ a) (f a) \\
\text{forget } (\Sigma T^+) & \quad (a, sh) \quad \mapsto \quad (a, \text{forget } (T^+ a) sh) \\
\text{forget } (\text{delete } s O) & \quad sh \quad \mapsto \quad (s, \text{forget } O sh)
\end{align*}
\]

Figure 8.2.: From ornament to Cartesian morphism

In the other direction, we are given a Cartesian morphism from \( F \) to \( G \). We return an ornament of the description of \( G \). For the isomorphism to hold, this ornament must map to the description of \( F \):

\[
\psi (m : F =^u \overset{v}u G) : \text{orn } (G)^{-1} u v
\]

\[
\psi (\text{forget}) \mapsto \lambda j. \Sigma sh. \text{insert } (\text{forget}^{-1} sh) \lambda ps. \text{var } (\text{inv } (n_F ps))
\]

Indeed, the description of \( G \) is a \( \Sigma \) of its operations, followed by a \( \Pi \) of its arities, terminated by a \( \text{var} \) at the desired sort. To ornament \( G \) into \( F \), we simply have to \text{insert} the inverse image of \text{forget}, i.e. the information that extends \( G \) to \( F \). As for the sorts, we can legitimately use \( F \)'s indexing function: the coherence condition of the Cartesian morphism ensures that it is indeed in the inverse image of the reindexing function.

We have carefully crafted \( \phi \) and \( \psi \) so that these functions are (extensionally) inverses of each other. Expressing these extensional evidences into an (intensional) theorem prover is arduous and sheds no new insight on the construction itself. Hence, we will not attempt to prove it in type theory here.

\[ \square \]

8.86 Remark (Relation with ornamental algebras (Section 8.1.2)). To define the ornamental algebra (Definition 8.34), we introduced the Cartesian morphism (Definition 8.32), a function taking an ornamented type to its unornamented form. This construction actually corresponds to our transformation \( \phi \), followed by the interpretation of the resulting Cartesian morphism. The Cartesian morphism \text{forgetNT} is indeed natural and Cartesian. This natural transformation was already present in the original presentation [McBride, 2013, §4], in the form of the helper function \text{erase}.

8.87 Remark (Terminology). We may now conflate the notions of ornament, Cartesian morphism, and Cartesian natural transformation. In particular, we shall say that “\( F \) ornaments \( G \)” when we have a Cartesian morphism from \( F \) to \( G \).
8. Ornaments

Let us now raid the container toolbox for the purpose of programming with ornaments. The next section shows the beginning of what is possible.

8.3. Tapping into the Categorical Structure


In the previous section, we have categorically characterised ornaments in terms of Cartesian morphisms. We now turn to the original ornamental constructions – such as the ornamental algebra and the algebraic ornament – and rephrase them in our categorical framework. Doing so, we extract the structure governing their type-theoretic definition and thus gain a finer understanding.

Next, we study the categorical structure of Cartesian morphisms and uncover novel and interesting ornamental constructions. We shall see how identity, composition, and the frame structure translate into ornaments. We shall also be interested in pullbacks in the category of containers and the functoriality of the derivative in that category.

8.3.1. Ornamental algebra

Following Remark 8.86, the forgetNT function that implements the ornamental algebra corresponds exactly to the Cartesian natural transformation described by the ornament. The ornamental algebra is thus a simple corollary of the very definition of ornaments as Cartesian morphisms.

**8.90 Corollary (Ornamental algebra).** From an ornament $o : F \xRightarrow{\mu} G$, we obtain the ornamental algebra forgetAlg $o : F (\mu G \circ v) \to \mu G \circ u$.

**Proof.** We apply the natural transformation $o$ at $\mu G$ and post-compose by $in$:

\[ \text{forgetAlg} o : F (\mu G \circ v) \xrightarrow{o \circ \mu G} (G \mu G) \circ u \xrightarrow{in} \mu G \circ u \]

\[\square\]

Going back to Section 8.1.2, we easily relate the type-theoretic definitions with their categorical model. As explained in Remark 8.86, the so-called Cartesian morphism forgetNT (Definition 8.32) is the interpretation of the Cartesian morphism. The ornamental algebra forgetAlg (Definition 8.34) is then obtained by composing the Cartesian morphism with the $in$ constructor.

8.91 **Computational interpretation.** Folding the ornamental algebra, we obtain a map from the ornamented type $\mu F$ to its unornamented version $\mu G$. In effect, the ornamental algebra describes how to forget the extra information introduced by the ornament.

**8.93 Example (Ornamental algebra: ListO).** The Cartesian morphism from list to natural numbers (Example 8.72) maps the nil constructor to 0, while the cons constructor is
8.3. Tapping into the Categorical Structure

mapped to \( \text{suc} \). Post-composing by \( \text{in} \), we obtain a natural number. This is the algebra computing the length of a list.

### 8.3.2. Algebraic ornaments

The notion of algebraic ornament was initially introduced by McBride [2013, §5]. A similar categorical construction, defined for any functor, was also presented by Atkey et al. [2012]. In this section, we reconcile these two works and show that, for a container, the refinement functor can itself be internalised as a container.

**Definition (Refinement functor [Atkey et al., 2012, §4.3]).** Let \( F \) be an endofunctor on \( \text{SET} \). Let \( (X: \text{SET}, \alpha: F X \to X) \) be an algebra over \( F \).

The refinement functor is defined by

\[
F^\alpha \triangleq \Sigma_{\alpha} \circ \hat{F}: \text{Set}^{(\cdot)} \times X \to \text{Set}^{(\cdot)} \times X
\]

where \( \hat{F} \) – the canonical lifting of \( F \) [Hermida and Jacobs, 1998, Fumex, 2012] – generalises the canonical lifting of descriptions (Definition 4.52) as follows

\[
\hat{F} (P:(i:I) \times X \to \text{SET}) (\text{ixs}:(i:I) \times F X i) : \text{SET} \\
\hat{F} \text{ixs} \mapsto (ps:F (\lambda-.(\text{ix}:(i:I) \times X i) \times P \text{ix})) \times F \pi_0 ps = \text{ixs}
\]

The idea, drawn from refinement types [Freeman and Pfenning, 1991], is that a function \( L^\alpha M: \mu F \to X \) can be thought of as a predicate over \( \mu F \). By integrating the algebra \( \alpha \) into the signature \( F \), we obtain a signature \( F^\alpha \) indexed by \( X \) that describes the \( F \)-objects satisfying, by construction, the predicate \( L^\alpha M \).

**Theorem (Coherence property of algebraic ornament [Atkey et al., 2012, Theorem 4.6]).** The fixpoint of the refinement functor \( P F^\alpha \) satisfies the isomorphism

\[
\mu F^\alpha i x \cong \Sigma_{\alpha} \mu F \\
\{t: \mu F i \mid (\alpha) t = x\}
\]

Informally, using a set-theoretic notation, this isomorphisms reads as

\[
\mu F^\alpha i x \cong \Sigma_{\alpha} \mu F \\
\{t: \mu F i \mid (\alpha) t = x\}
\]

That is, the algebraic ornament \( \mu F^\alpha \) at index \( i \) and \( x \) corresponds exactly to the pair of a witness \( t \) of \( \mu F i \) and a proof that this witness satisfies the indexing equation \( (\alpha) t = x \).

When \( F \) is a polynomial functor, we show that the refinement functor can be internalised and presented as an ornament of \( F \). In practice, this means that from a description \( D \) and an algebra \( \alpha \), we can compute an ornament code that describes the functor \( D^\alpha \). This should not come as a surprise: we defined algebraic ornaments (Definition 8.41) exactly this way, based on the original presentation of McBride [2013, §5]. The following theorem abstracts this original definition.

**Proposition.** Let \( \sigma \triangleq \text{Op}<\text{Sort Ar} \) be a container indexed by \( I \). Let \( (X, \alpha) \) be an algebra over \( F \), i.e. \( \alpha: [\sigma]_{\text{Cont}} X \to X \).
The refinement functor $\sigma^a$ is polynomial and ornaments $\sigma$.

**Proof.** We first show that $\hat{\sigma}$ ornaments $\sigma$, and then that $\Sigma_a$ ornaments the identity container. By horizontal composition, we thus get that $\sigma^a$ ornaments $\sigma$.

To show that $\hat{\sigma}$ ornaments $\sigma$, we remark that the canonical lifting of a container itself a container. Indeed, we define

\[
\begin{align*}
\mathbb{C} \sqsupseteq \sigma^a \colon & \text{ICont} ((i: I) \times X i) \times \llbracket \sigma \rrbracket_{\text{Cont}} X \ j) \\
\mathbb{C} \sqsupseteq \text{Op} \colon & \text{ICont} ((j: J) \times \llbracket \sigma \rrbracket_{\text{Cont}} X \ j) : \text{SET} \\
\mathbb{C} \sqsupseteq \text{Op}^{jxs} & : \text{SET} \\
\mathbb{C} \sqsupseteq \text{Ar}^l \colon & \text{SET} \\
\mathbb{C} \sqsupseteq \text{Ar}^{op} \colon & \text{SET} \\
\mathbb{C} \sqsupseteq \text{Sort}^l \colon & \text{SET} \\
\mathbb{C} \sqsupseteq \text{Sort}^{op} \colon & \text{SET}
\end{align*}
\]

We then verify that $\llbracket \mathbb{C} \sqsupseteq \sigma \rrbracket_{\text{Cont}} = \llbracket \hat{\sigma} \rrbracket_{\text{Cont}}$ and that there exists a container morphism from $\mathbb{C} \sqsupseteq \sigma$ to $\sigma$, which is suggested by the fact that both containers have the same arities.

Similarly, we can construct a container representing $\Sigma_a$ (along the lines of its definition as a description in the proof of Lemma 5.68), and build a Cartesian natural transformation from the container describing $\Sigma_a$ to the identity container indexed by $I$.

By horizontal composition of these two Cartesian natural transformations, we obtain a Cartesian natural transformation from $\sigma^a = \Sigma_a \circ \hat{\sigma}$ to $\text{id} \circ \sigma \cong \sigma$.

\[(8.101)\]

The categorical presentation of refinement functors (Definition 8.95) gives an interesting perspective on our definition of the algebraic ornament (Definition 8.41). We now understand the definition of $F^a$ as the composition of the canonical lifting, followed by the op-reindexing by $a$.

First, we explicitly appeal to the canonical lifting. Second, the op-reindexing is coded by inserting an $xs : \llbracket D \rrbracket \times X k$ satisfying the constraint $a \ xs = x$. In the internal language, we have:

\[
\Sigma_a : \text{SET} / \llbracket D \rrbracket \times X k \to \text{SET} / X
\]

\[
\Sigma_a \{ P_{xs} \mid x \in \llbracket D \rrbracket \times X \} \triangleq \left\{ \sum_{xs \in a^{-1}x} P_{xs} \mid x \in X \right\}
\]

\[
\triangleq \left\{ (xs : \llbracket D \rrbracket \times X) \times a \ xs = x \times P_{xs} \mid x \in X \right\}
\]

8.3.3. Categorical structures

\[(8.102)\]

**Identity.** A trivial ornamental construction is the *identity* ornament. Indeed, for every container, there is a Cartesian morphism from and to itself, introducing no extension and no refinement.

**8.103 Definition** (Identity ornament). In terms of ornament code, this construction simply consists in copying the code of the description: this is a generic program, taking a
8.3. Tapping into the Categorical Structure

description as input and returning the identity ornament:

\[
\text{idO} (D : \text{Desc} I) : \text{Orn} D \text{id}
\]

\[
\text{idO} (\text{var} i) \mapsto \text{var} (\text{inv} i)
\]

\[
\text{idO} 1 \mapsto 1
\]

\[
\text{idO} A \times B \mapsto \text{idO} A \times \text{idO} B
\]

\[
\text{idO} (\sigma E T) \mapsto \sigma \lambda e. \text{idO} (\text{switch} T e)
\]

\[
\text{idO} (\Pi S T) \mapsto \Pi \lambda s. \text{idO} T s
\]

\[
\text{idO} (\Sigma S T) \mapsto \Sigma \lambda s. \text{idO} T s
\]

This definition then lifts pointwise to ornaments:

\[
\text{idO} : (D : \text{func} I J) \rightarrow \text{orn} D \text{id}
\]

(8.104) **Vertical composition.** The next structure of interest is composition. Recall that an ornament corresponds to a (Cartesian) natural transformation. There are therefore two notions of composition. First, vertical composition lets us collapse chains of ornaments:

\[
\begin{array}{c}
\text{SET}/I \\
\downarrow o_1 \\
\downarrow o_2 \\
\end{array}
\quad G
\quad \begin{array}{c}
\text{SET}/J \\
\downarrow o_1 \\
\downarrow o_2 \\
\end{array}
= \begin{array}{c}
\text{SET}/I \\
\downarrow o_1 \\
\downarrow o_2 \\
\end{array}
\quad H
= \begin{array}{c}
\text{SET}/I \\
\downarrow o_1 \\
\downarrow o_2 \\
\end{array}
\quad H
\end{array}
\]

8.105 Example (Vertical composition of ornaments). We have seen that List ornaments \text{Nat}. We also know that Vec ornaments List. By vertical composition, we thus obtain that Vec ornaments \text{Nat}.

(8.106) **Horizontal composition.** Turning to horizontal composition, we have the following identity:

\[
\begin{array}{c}
\text{SET}/I \\
\downarrow o_1 \\
\downarrow o_2 \\
\end{array}
\quad G_1
\quad \begin{array}{c}
\text{SET}/J \\
\downarrow o_1 \\
\downarrow o_2 \\
\end{array}
\quad G_2
\quad \begin{array}{c}
\text{SET}/K \\
\downarrow o_1 \\
\downarrow o_2 \\
\end{array}
= \begin{array}{c}
\text{SET}/I \\
\downarrow o_1 \\
\downarrow o_2 \\
\end{array}
\quad G_1 \\
\quad \begin{array}{c}
\text{SET}/K \\
\downarrow o_1 \\
\downarrow o_2 \\
\end{array}
\quad G_2
\end{array}
\]

8.107 Example (Horizontal composition of ornaments). Let us consider the following
8. Ornaments

containers:

\[
\begin{align*}
\text{Square } X &\mapsto X \times X : \text{SET}/\emptyset \to \text{SET}/\emptyset \\
\text{Height } \{X_n | n \in \text{Nat}\} &\mapsto \{X_n \times X_{n+1} | n \in \text{Nat}\} \\
&\quad + \{X_n \times X_n | n \in \text{Nat}\} : \text{SET}/\text{Nat} \to \text{SET}/\text{Nat}
\end{align*}
\]

It is easy to check that VecCont (Example 5.57) ornaments ListCont (Example 5.56). In fact, Example 8.63 showed that vectors are a reornament of lists. Besides, Height ornaments Square. By horizontal composition of these ornaments, we obtain that VecCont \(\circ\) Height – describing a balanced binary tree – is an ornament of ListCont \(\circ\) Square – describing a binary tree. Thus, we obtain that balanced binary trees ornament binary trees.

\[(8.108)\] Frame structure. Finally, the frame structure (¶8.75) lets us lift morphisms on indices by ornamentation.

8.109 Example (Cobase-change ornament). Recall the arithmetic expressions indexed by their semantics (Example 8.46):

\[
\begin{align*}
\text{data } \text{Expr}^{\text{evalAlg}} (k : \text{Nat}) : \text{SET} \text{ where } \\
\text{Expr}^{\text{evalAlg}} (k = n) &\ni \text{const } (n : \text{Nat}) \\
\text{Expr}^{\text{evalAlg}} (k = m + n) &\ni \text{add } (m \ n : \text{Nat}) (d : \text{Expr}^{\text{evalAlg}} m) (e : \text{Expr}^{\text{evalAlg}} n)
\end{align*}
\]

One could be interested in a subset of these expressions: for example, we might want to manipulate those expressions that are strictly positive. Strictly-positive natural numbers are defined by

\[
\begin{align*}
\text{data } \text{Nat}^* : \text{SET} \text{ where } \\
\text{Nat}^* &\ni 1 \\
&\quad | \text{suc } (n : \text{Nat}^*)
\end{align*}
\]

and trivially embeds into natural numbers:

\[
\begin{align*}
\text{toNat } (n : \text{Nat}^*) &\ni \text{Nat} \\
\text{toNat } 1 &\ni \text{suc } 0 \\
\text{toNat } (\text{suc } n) &\ni \text{suc } (\text{toNat } n)
\end{align*}
\]

The container describing \(\text{Expr}^{\text{evalAlg}}\) is indexed by \(\text{Nat}\): we can therefore reindex it by \(\text{toNat}\). We obtain a container that describes the expressions denoting strictly-positive natural numbers. Put explicitly, the object thus computed is the following datatype:

\[
\begin{align*}
\text{data } \text{Expr}^{\text{evalAlg}^+} (k : \text{Nat}^*) : \text{SET} \text{ where } \\
\text{Expr}^{\text{evalAlg}^+} k &\ni \text{const } (n : \text{Nat}) (q : \text{toNat } k = n) \\
&\quad | \text{add } (m \ n : \text{Nat}^*) (d : \text{Expr}^{\text{evalAlg}^+} m) (e : \text{Expr}^{\text{evalAlg}^+} n) \\
&\quad | (q : \text{toNat } k = \text{toNat } m + \text{toNat } n)
\end{align*}
\]

For both constructors, the strict-positivity of inhabitants of \(\text{Expr}^{\text{evalAlg}^+}\) is enforced by the side-conditions \(q\): the (strictly-positive) index \(k\) must be equal to the denotation of
8.3. Tapping into the Categorical Structure

8.110 Remark. The identity, vertical and horizontal compositions, and frame structure illustrate the algebraic properties of ornaments: not only can we define ornaments, but we can also combine them. The categorical simplicity of ornaments gives us a finer understanding of datatypes and their relation to each other. This is demonstrated by Example 8.107 for instance.

8.3.4. Pullback of ornaments

So far, we have merely exploited the fact that containers form a (framed bi)category. However, it has a much richer structure. That extra structure can in turn be translated into ornamental constructions. We shall focus our attention on pullbacks, but we expect other categorical notions to be of programming interest.

8.112 Example (Pullback of ornament). Natural numbers can be ornamented to lists (Example 8.25) as well as finite sets (Example 8.26). Taking the pullback of these two ornaments, we obtain bounded lists that correspond to lists of bounded length, with the bound given by an index \( n : \text{Nat} \). Put explicitly, the object thus computed is the following datatype:

\[
\text{data BoundedList} [A : \text{SET}](n : \text{Nat}) : \text{SET} \text{ where}
\]

\[
\text{BoundedList}_A (n = \text{suc } n') \ni \text{nil} (n' : \text{Nat})
\]

\[
| \text{cons} (n' : \text{Nat})(a : A)(as : \text{BoundedList}_A n')
\]

This construction is not unique to this pair of ornaments. In fact, every pair of ornaments admits a pullback, as established by the following proposition.

8.114 Proposition ([Dagand and McBride, 2013a, Proposition 4]). The category of containers has all pullbacks.

8.115 Remark. The pullback construction is another algebraic property of ornaments: given two ornaments, both describing an extension of the same datatype (e.g. extending natural numbers to lists and extending natural numbers to finite sets), we can “merge” them into one having both characteristics (i.e. bounded lists). In type theory, Ko and Gibbons [2011] have experimented with a similar construction for composing indexing disciplines.

8.3.5. Derivative of ornament

Abbott et al. [2005] have shown that the Zipper [Huet, 1997] data-structure can be computed from the derivative of signature functors. Interestingly, the derivative is characterised by the existence of a universal arrow in the category of containers.

8.117 Definition (Differentiability [Abbott et al., 2005]). Let \( F \) be a container indexed by \( I : \text{SET} \).

\( F \) is differentiable in \( i : I \) if and only if, for any container \( G \), we have the following
8. Ornaments

bijection of morphisms

\[
\frac{\text{ICont}(G \times \pi_i, F)}{\text{ICont}(G, \partial_i F)}
\]

We denote \(\text{ICont}^\partial\) the class of containers differentiable in \(i\).

8.118 Remark (Intuition). A container can be thought of as describing a polynomial of multiple (in fact, \(I\)) variables. The above definition of a partial derivative is merely the multivariate counterpart to the derivative on (non-indexed) polynomials [Abbott et al., 2005], i.e. inductive types. Where the univariate derivative computes the one-hole context of some inductive type, the partial derivative in \(i\) computes the one-hole context of some inductive family at index \(i\), leaving the other variables unchanged. Following the mono-sorted setting, from the one-hole context of an inductive family, we generically obtain its (indexed) Zipper [Morris, 2007].

8.119 Example (Ornamentation of derivative). Let us consider binary trees. Balanced binary trees are an ornamentation of binary trees (Example 8.107). We also have that the derivative of balanced binary trees is an ornament of the derivative of binary trees. This observation carries over to any ornament: ornamentation is stable by derivation, as demonstrated by the following theorem.

8.121 Proposition. Let \(F\) and \(G\) be two containers in \(\text{ICont}^\partial\).

If \(F\) ornaments \(G\), then \(\partial_i F\) ornaments \(\partial_i G\).

Proof. The proof simply follows from the functoriality of \(\partial_i\) over \(\text{ICont}^\partial\) [Abbott, 2003, Section 6.4]. Intuitively, we have the following Cartesian morphism

\[
\partial_i F \times \pi_i \rightarrow^c F \rightarrow^c G
\]

where the first component is the unit of the universal arrow while the second component is the ornament from \(F\) to \(G\). By definition of differentiability, we therefore have the desired Cartesian morphism:

\[
\partial_i F \rightarrow^c \partial_i G
\]

8.122 Remark. The derivative is thus an example of an operation on datatypes that preserves ornamentation. Knowing that the derivative of an ornamented datatype is an ornamentation of the derivative of the original datatype, we get that the order in which we ornament or derive a datatype does not matter. This lets us relate datatypes across such transformations, thus preserving the structural link between them.
8.3. Tapping into the Categorical Structure

Conclusion

(8.123) In this chapter, we have adapted McBride’s ornaments to our universe of datatypes. This gives us the ability to compute over indices. Consequently, we have extended the original presentation with a deletion ornament. Deletion ornaments are a key ingredient for the internalisation of Brady’s optimisation [Brady et al., 2003] over inductive families. In particular, this gave us a simpler implementation of reornaments.

(8.124) We then gave a categorical presentation of ornaments. Doing so, we get to the essence of ornaments: ornamenting a datatype consists in extending it with new information, and refining its indices. Formally, this characterisation turns into an abstract representation of ornaments as Cartesian morphisms of containers. Having such an abstract understanding of ornaments, an entirely new design space opens up to us.

(8.125) We have reported some initial results of our explorations. We translate the type-theoretic ornamental toolkit to the categorical framework. Doing so, we gain a deeper understanding of the original definitions. Then, we have expressed the categorical structure of containers in terms of ornaments, discovering new constructions – identity, vertical composition, horizontal composition, and frame structure of ornaments – in the process. Last but not least, we have studied the algebraic structure of containers, obtaining the notion of pullback of ornaments and derivative of ornaments.

Related work

(8.126) Ornaments were initially introduced by McBride [2013] as a programming artefact. They were presented in type theory, with a strong emphasis on their computational contribution. Ornaments were thus introduced through a universe. Constructions on ornaments – such as the ornamental algebra, algebraic ornament, and reornament – were introduced as programs in this type theory, relying crucially on the concreteness of the universe-based presentation.

While this approach has many pedagogical benefits, it was also clear that more abstract principles were at play. For example, in this chapter, we have adapted the notion of ornaments to our universe of inductive families, whilst Ko and Gibbons [2011] explore datatype engineering with ornaments in yet another universe. This chapter gives such an abstract treatment. This focus on the theory behind ornaments thus complements the original, computational treatment.

Building upon that original paper, Ko and Gibbons [2011] also identify the pullback structure – called “composition” in their paper – as significant, giving a treatment for a concrete universe of ornaments and compelling examples of its effectiveness for combining indexing disciplines. The conceptual simplicity of our approach lets us subsume their type-theoretic construction as a mere pullback.

(8.127) Algebraic ornaments were also treated categorically by Atkey et al. [2012]: instead of focusing on a restricted class of functors, the authors described the refinement of any functor by any algebra. The constructions are presented in the generic framework of fibrations. The refinement construction described in their paper, once specialised to containers, corresponds exactly to the notion of algebraic ornament, as we have shown.
8. Ornaments

(8.128) Algebraic ornament are strikingly similar to the “upward incrementalisation” of Leather et al. [2011]. Upward incrementalisation consists of memoising the application of an algebra over a datatype. To do so, one defines another datatype whose operations are extended to store the result of the algebra. Algebraic ornaments are an indexed counterpart of this construction: no information is stored in the datatype, instead it flows through the indices. The arguments’ sorts contain the result of the algebra of the subnodes and the constructor’s resulting sort is computed by applying the algebra over the argument’s sorts.

It would be interesting to study the indexed counterpart of the “downward incrementalisation” described by the authors, and combinations of upward and downward incrementalisation. While we do not foresee any technical difficulty, the practical interest of such a definition is less clear. Indeed, when defining an indexing discipline, we normally focus on structural properties, i.e. properties that flows from the leaf of the data-structure to its root. Thus, when grafting subtrees together with a constructor, the indexing information flows quite naturally from the parts to define the type of the whole. A downward indexing discipline would propagate from the top down: when grafting subtrees, one would likely have to fix-up the type of the subnodes to satisfy the constraint arbitrarily set by the constructor.

(8.129) Finally, it is an interesting coincidence that Cartesian morphisms should play such an important role in structuring ornaments. Indeed, containers stem from the work on shapely types [Jay and Cockett, 1994]. In the shape framework, a few base datatypes were provided (such as natural numbers) and all the other datatypes were grown from these basic blocks by a pullback construction, i.e. an ornament. However, this framework was simply typed, hence no indexing was at play.
9. Functional Ornaments

(9.1) Thanks to ornaments (Chapter 8), we now have a good handle on the transformation of individual datatypes. However, we are still facing a major reusability issue: a datatype comes equipped with a set of operations. Ornamenting this datatype, we have to re-implement many similar operations for the ornamented version.

9.2 Example. The datatype $\text{Nat}$ comes with operations such as addition and subtraction. When defining $\text{List A}$ as an ornament of $\text{Nat}$, it seems natural to transport the structure-preserving functions of $\text{Nat}$ to $\text{List A}$, such as moving from addition of natural numbers to concatenation of lists:

\[
\begin{align*}
(m: \text{Nat}) + (n: \text{Nat}) : \text{Nat} & \quad (xs: \text{List A}) ++ (ys: \text{List A}) : \text{List A} \\
0 + n \mapsto n & \quad \text{nil} ++ ys \mapsto ys \\
(suc m) + n \mapsto suc (m + n) & \quad (\text{cons a xs}) ++ ys \mapsto \text{cons a (xs ++ ys)}
\end{align*}
\]

Or, from subtraction of natural numbers to dropping a prefix:

\[
\begin{align*}
(m: \text{Nat}) - (n: \text{Nat}) : \text{Nat} & \quad \text{drop} (xs: \text{List A}) (n: \text{Nat}) : \text{List A} \\
0 - n \mapsto 0 & \quad \text{drop} \; \text{nil} \; n \mapsto \text{nil} \\
m - 0 \mapsto m & \quad \text{drop} \; xs \; 0 \mapsto xs \\
(suc m) - (suc n) \mapsto m - n & \quad \text{drop} \; (\text{cons a xs}) \; (suc n) \mapsto \text{drop} \; xs \; n
\end{align*}
\]

(9.3) We want to generalise the notion of ornament to functions. In order to do this, we first need to be able to manipulate functions in type theory and, in particular, their types. To this effect, we define a universe of functions. With it, we are able to write generic programs over the class of functions captured by our universe. Using this technology, we define a functional ornament as a decoration over the universe of functions. The liftings implementing the functional ornament are related to the base function by a coherence property. To minimise the burden induced by coherence proofs, we expand our system with patches: a patch is the type of the functions that satisfy the coherence property by construction. Finally, and still writing generic programs, we show how we can automatically project the lifting and its coherence certificate out of a patch.

(9.4) First, we work through an example in Section 9.1. We explore the structural ties that link addition on natural numbers and concatenation of lists. Upon identifying some common structure, we cast it in terms of ornaments and their reornaments. Through this pedestrian presentation, we lay the foundations for the next sections: we introduce the vocabulary and the general intuitions.

(9.5) Then, we introduce functional ornaments by a universe construction in Section 9.2. Based on this universe, we establish the connection between a base function (such as
9. Functional Ornaments

\[
\begin{align*}
(m \colon \text{Nat}) < (n \colon \text{Nat}) & : \text{Bool} & \text{lookup} (m \colon \text{Nat}) (xs \colon \text{List} \ A) & : \text{Maybe} \ A \\
0 & < 0 \implies \text{false} & \text{lookup} m \nil & \mapsto \text{nothing} \\
0 & < \text{suc} n \implies \text{true} & \text{lookup} 0 (\text{cons} a xs) & \implies \text{just} a \\
\text{suc} m & < \text{suc} n \implies m < n & \text{lookup} (\text{suc} n) (\text{cons} a xs) & \implies \text{lookup} n xs
\end{align*}
\]

Figure 9.1.: Implementation of \(- < -\) and lookup

addition and subtraction) and its ornamented version (such as, respectively, \(- ++ -\) and drop). Within this framework, we redevelop the example of Section 9.1 with all the automation offered by our system.

9.6 In Section 9.3, we provide further support to drive the computer into lifting functions semi-automatically. As we can see from our examples above, the lifted functions often follow the same recursion pattern and return similar constructors: with a few constructions, we shall remove further clutter and code duplication from our libraries.

9.7 Remark (Meta-theoretical status). It is crucial to note that this chapter is built entirely within type theory. No change or adaptation to the meta-theory is required. In particular, the validity of our constructions is justified by mere type checking.

9.1. From Comparison to Lookup, Manually

9.8 There is an astonishing resemblance between the comparison function \(- < -\) on numbers and the list lookup function (Figure 9.1). Interestingly, the similarity is not merely at the level of types. It is also in their implementation: their definition follows the same pattern of recursion (first, case analysis on the second argument; then induction on the first argument) and they both return a failure value (respectively, \text{false} and \text{nothing}) in the first case analysis and a success value (respectively, \text{true} and \text{just}) in the base case of the induction.

9.9 This raises the question: what exactly is the relation between \(- < -\) and lookup? Also, could we use the implementation of \(- < -\) to guide the construction of lookup? First, let us work out the relation at the type level. To this end, we use ornaments to explain how each individual datatype has been promoted when going from \(- < -\) to lookup:

\[
\begin{align*}
\text{\text{- < -}} : \text{Nat} & \to \text{Nat} & \to \text{Bool} \\
\text{idO}_{\text{Nat}} & & \text{ListO} \ A & & \text{MaybeO} \ A \\
\text{lookup} & : \text{Nat} & \to \text{List} \ A & \to \text{Maybe} \ A
\end{align*}
\]

Note that the first argument is ornamented to itself, or put differently, it has been ornamented by the identity ornament.
Ornamental algebras. Each of these ornaments come with a forgetful map, computed from the ornamental algebra. These forgetful maps correspond to the following functions:

- \( \text{length} \ (as : \text{List} \ A) : \text{Nat} \)
- \( \text{isJust} \ (m : \text{Maybe} \ A) : \text{Bool} \)
- \( \text{length} \ \text{nil} \mapsto 0 \)
- \( \text{isJust} \ \text{nothing} \mapsto \text{false} \)
- \( \text{length} \ (\text{cons} \ a \ as) \mapsto \text{suc} \ (\text{length} \ as) \)
- \( \text{isJust} \ (\text{just} \ a) \mapsto \text{true} \)

Using these forgetful map we deduce a relation, at the operational level, between \( \text{−} < \text{−} \) and \( \text{lookup} \). This relation is uniquely determined by the ornamentation of the individual datatypes. This coherence property is expressed as follows:

\[
\forall n : \text{Nat}. \forall xs : \text{List} \ A. \ \text{isJust} \ (\text{lookup} \ n \ xs) = n < \text{length} \ xs
\]

Or, equivalently, using a commuting diagram:

\[
\begin{array}{cccc}
\text{Nat} \times \text{List} \ A & \xrightarrow{\text{lookup}} & \text{Maybe} \ A \\
\downarrow \text{id} \times \text{length} & & \downarrow \text{isJust} \\
\text{Nat} \times \text{Nat} & \xrightarrow{\text{−} < \text{−}} & \text{Bool}
\end{array}
\]

9.12 Remark (Vocabulary). We call the function we start with the base function (here, \( \text{−} < \text{−} \)), its type being the base type (here, \( \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Bool} \)). The richer function type built by ornamenting the individual pieces is called the functional ornament (here, \( \text{Nat} \rightarrow \text{List} \ A \rightarrow \text{Maybe} \ A \)). A function inhabiting this type is called a lifting (here, \( \text{lookup} \)). A lifting is said to be coherent if it satisfies the coherence property.

9.13 Remark (Coherence and functional ornament). It is crucial to understand that the coherence of a lifting is relative to a given functional ornament: the same base function ornamented differently would give rise to a different coherence property.

9.14 We now have a better grasp of the relation between the base function and its lifting. However, \( \text{lookup} \) remains to be implemented while making sure that it satisfies the coherence property. Traditionally, one would stop here: we would implement \( \text{lookup} \) and prove the coherence as a theorem. This works rather well in a system like Coq since it offers a powerful theorem proving environment. It does not work so well in a system like Agda that does not offer tactics to its users, forcing them to write explicit proof terms. It would not work at all in an ML language with GADTs, which has no notion of proof.

9.15 Remark (Methodology). We are not satisfied by this laborious approach: if we have dependent types, why should we use them only for proofs, as an afterthought? We should rather write a lookup function correct by construction: by implementing a more precisely indexed version of \( \text{lookup} \), the user can drive the type checker into verifying
9. Functional Ornaments

the necessary invariants. We believe that this is how it should be: computers should check proofs by computation; humans should drive computers. The other way around – where humans are coerced into computing for computers – may seem surreal, yet it reflects the current situation in most proof systems.

(9.16) To get the computer to work for us, we would rather implement the function \texttt{ilookup}

\[
\text{ilookup} \quad \texttt{(m:Nat)} \quad \texttt{(vs:Vec A n)} \quad \rightarrow \texttt{IMaybe A (m < n)}
\]

\[
\text{ilookup} \quad m \quad \texttt{nil} \quad \rightarrow \texttt{nothing}
\]

\[
\text{ilookup} \quad 0 \quad \texttt{(cons a vs)} \quad \rightarrow \texttt{just a}
\]

\[
\text{ilookup} \quad \texttt{(suc m)} \quad \texttt{(cons a vs)} \quad \rightarrow \texttt{ilookup m vs}
\]

where \texttt{Vec A} is the reornament of \texttt{List A} by its length (Example 8.55), and \texttt{IMaybe A} is the reornament of \texttt{Maybe A} by its truth as computed by \texttt{isJust} (Example 8.64).

Indeed, the accuracy afforded by \texttt{ilookup}'s type significantly constrains – and thus simplifies – its implementation. For instance, having determined that the vector is empty (first pattern), the return type leaves us no choice but to give the constructor \texttt{nothing}. In fact, in an interactive system such as Agda extended with Agsy [Lindblad and Benke, 2004], the user can drive the system to implement \texttt{ilookup} without typing a single term herself: we only need to setup the matching patterns, and then appeal to Agsy's automation to generate the (desired) returned terms.

(9.17) The rationale behind \texttt{ilookup} is to index the types of \texttt{lookup} by their unornamented version, i.e. the arguments and result of \texttt{−<−}. Hence, we can make sure that the result computed by \texttt{ilookup} respects the output of \texttt{−<−} on the unornamented indices: the result is correct by indexing! The type of \texttt{ilookup} is naturally derived from the ornamentation of \texttt{−<−} into \texttt{lookup} and is uniquely determined by the functional ornament we start with.

9.18 Remark (Vocabulary). Expanding further our vocabulary, we call coherent liftings these finely indexed functions that are coherent by construction.

9.19 We use reornaments to internalise the coherence requirements. From \texttt{ilookup}, we can extract both \texttt{lookup} and its proof of correctness without having written any proof term ourselves:

\[
\text{lookup} \quad \texttt{(m:Nat)} \quad \texttt{(xs:List A)} \quad \rightarrow \texttt{Maybe A}
\]

\[
\text{lookup} \quad m \quad \texttt{xs} \quad \rightarrow \pi_0(\texttt{forgetIMaybe (ilookup m (makeVec xs)))}
\]

\[
\text{cohLookup} \quad \texttt{(n: Nat)} \quad \texttt{(xs: List A)} \quad \rightarrow \texttt{isJust (lookup n xs) = n < length xs}
\]

\[
\text{cohLookup} \quad m \quad \texttt{xs} \quad \rightarrow \pi_1(\texttt{forgetIMaybe (ilookup m (makeVec xs)))}
\]

We rely on the function \texttt{makeVec : (xs: List A) \rightarrow Vec A (length xs)}, which turns a list into a vector of the corresponding length. It is derived automatically from the vector reornament (¶ 8.51). Operationally, it is an identity.

9.20 Remark (\texttt{ilookup vs. vlookup}). The function \texttt{ilookup} is very similar to the more fa-
miliar \texttt{vlookup} function:

\[
\begin{align*}
\text{vlookup } & (m: \text{Fin } n) \ (vs: \text{Vec } A \ n) : A \\
vlookup & \quad f0 \ (\text{cons } a \ xs) \mapsto a \\
vlookup & \quad (\text{fsuc } n) \ (\text{cons } a \ xs) \mapsto \text{vlookup } n \ xs
\end{align*}
\]

These two definitions are actually equivalent, thanks to the isomorphism:

\[
(m: \text{Nat}) \rightarrow \text{IMaybe } A (m < n) \cong \text{Fin } n \rightarrow A
\]

Intuitively, we can move the constraint that \(m < n\) from the result – where we return an object of type \(\text{IMaybe } A (m < n)\) – to the premise – where we expect an object of type \(\text{Fin } n\). Indeed, we can think of the type \(\text{Fin } n\) as the combination of a number \(m: \text{Nat}\) together with a proof that \(m < n\).

With this example, we have manually unfolded the key steps of the construction of a lifting of \(- < -\). Let us recapitulate each steps:

- Start with a base function, here \(- < - : \text{Nat } \rightarrow \text{Nat } \rightarrow \text{Bool}\)
- Ornament its inductive components as desired, here \(\text{Nat}\) to \(\text{List } A\) and \(\text{Bool}\) to \(\text{Maybe } A\) in order to describe the lifting of interest, here \(\text{lookup}: \text{Nat } \rightarrow \text{List } A \rightarrow \text{Maybe } A\) satisfying

\[
\forall n: \text{Nat}. \forall xs: \text{List } A. \ isJust (\text{lookup } n \ xs) \ = \ n < \text{length } xs
\]

- Implement a carefully indexed version of the lifting, here \(\text{ilookup}: (m: \text{Nat}) (vs: \text{Vec } A \ n) \rightarrow \text{IMaybe } A (m < n)\)

- Derive the lifting, here \(\text{lookup}\), and its coherence proof, without writing a proof! Besides, the implementation of \(\text{ilookup}\) is useful on its own: this function corresponds exactly to vector lookup, a function that one would have implemented anyway.

This manual unfolding of the lifting is instructive: it involves many constructions on datatypes (here, the datatypes \(\text{List } A\) and \(\text{Maybe } A\)) as well as on functions (here, the type of \(\text{ilookup}\), the definition of \(\text{lookup}\) and its coherence proof). Yet, it feels like a lot of these constructions could be automated. In the next section, we shall build the machinery to describe these transformations and obtain them within type theory.

9.2. A Universe of Functions and their Ornaments

We now generalise ornaments to functions. To do so, we first need to be able, in type theory, to manipulate functions and their types. We thus define a universe of functions (Section 9.2.1). With it, we will be able to write generic programs over the class of functions captured by this universe. We define a functional ornament as a decoration
over the universe of functions (Section 9.2.2). The liftings implementing the functional ornament are related to the base function by a coherence property.

To minimise the theorem-proving burden induced by coherence proofs, we expand our system with patches (Section 9.2.3): a patch is the type of the functions that satisfy the coherence property by construction. Finally, we show how we can project the lifting and its coherence certificate out of a patch (Section 9.2.4).

9.2.1. A universe of functions

**MODEL:** Chapter9.Functions

**Definition (Universe of types).** For clarity of exposition, we restrict our language of types to the bare minimum: a type can either be an exponential whose domain is an inductive type, or a product whose first component is an inductive type, or the unit type – used as a termination marker:

$$
\text{data \ Type: \ SET}_1 \ \text{where} \ \\
\text{Type } \ni \ \mu \{(D : \text{func } K K) (k : K) \rightarrow (T : \text{Type}) \} \ \\
\text{ } \mid \mu \{(D : \text{func } K K) (k : K) \} \times (T : \text{Type}) \ \\
\text{ } \mid 1
$$

This universe codes the function space from some (maybe none) inductive types to some (maybe none) inductive types. Concretely, the codes are interpreted as follows:

$$
\llbracket (T : \text{Type}) \rrbracket_{\text{Type}} \rightarrow \text{SET} \\
\llbracket \mu D k \rightarrow T \rrbracket_{\text{Type}} \rightarrow \mu D k \rightarrow \llbracket T \rrbracket_{\text{Type}} \\
\llbracket \mu D k \times T \rrbracket_{\text{Type}} \rightarrow \mu D k \times \llbracket T \rrbracket_{\text{Type}} \\
\llbracket 1 \rrbracket_{\text{Type}} \rightarrow 1
$$

The constructions we develop next could be adapted to a more powerful universe, such as one supporting non-inductive parameters, dependent quantifiers, or higher-order functions. However, this would needlessly complicate our exposition.

For instance, the treatment of non-inductive parameters would lead to further, but orthogonal, extensions of the functional ornaments; namely, inserting or deleting these quantifiers during functional ornamentation. To support higher-order functions, we are bound to make a distinction between covariant and contravariant ornamentations, which we can simply overlook in a first-order system.

**Example (Coding − < −).** Written in the universe of function types, the type of the comparison function is:

$$
type< : \text{Type} \\
type< \rightarrow \mu \{ \text{NatD } \} \rightarrow \mu \{ \text{NatD } \} \rightarrow \mu \{ \text{BoolD } \} \times 1
$$

The implementation of − < − is essentially the same as earlier, except that it must now
return a pair of a Boolean and an inhabitant of the unit type. To be explicit about the recursion pattern of this function, we make use of Epigram’s by (⇐) gadget:

\[
\begin{align*}
- & < - : \text{Type} \\
\text{Nat-case } n & \iff m < n \\
0 & < \text{suc } n \iff \text{false, } * \\
\text{suc } m & < \text{suc } n \iff m < n
\end{align*}
\]

That is, we first do a case analysis on \(n\) and then, in the successor case, we proceed by induction over \(m\).

**9.27 Example** (Coding \(- - + \)). In the universe of function types, the type of addition is given by:

\[
\text{type}+ : \text{Type} \\
\text{type}+ & \mapsto \mu \{ \text{NatD} \times \} \mapsto \mu \{ \text{NatD} \times \} \mapsto \mu \{ \text{NatD} \times \} \times 1
\]

Again, up to a trivial multiplication by \(1\), the implementation of \(- - +\) is left unchanged:

\[
\begin{align*}
- & + - : \text{Type} \\
\text{Nat-elim } m & \iff m + n \\
0 & + n \mapsto (n, *) \\
\text{suc } m + n & \mapsto (\text{suc } m + n, *)
\end{align*}
\]

That is, we proceed by induction over \(m\).

**9.2.2. Functional ornament**

MODEL: Chapter9.FunOrnament

(9.28) It is now straightforward to define functional ornaments: we traverse the function type and ornament the inductive types as we go. Note that it is always possible to leave an object unornamented: we ornament by the identity, which simply copies the original description.

**9.29 Definition** (Universe of functional ornaments). Following this intuition, we define functional ornaments by the following grammar:

\[
\text{data FunOrn } (T : \text{Type}) : \text{SET} \exists 1 \text{ where} \\
\text{FunOrn } (\mu[D k \to T]) & \triangleright \mu^+ \{ (D u u) (i : u \mapsto k) \to (T^+ : \text{FunOrn } T) \\
\text{FunOrn } (\mu[D k \times T]) & \triangleright \mu^+ \{ (D u u) (i : u \mapsto k) \times (T^+ : \text{FunOrn } T) \\
\text{FunOrn } \exists 1 & \triangleright \exists 1
\]

**9.30 Definition** (Lifting type). We get the type of the liftings by interpreting the orna-
9. Functional Ornaments

ments as we traverse the functional ornament:

\[
\begin{align*}
\{ (T^+ : \text{FunOrn } T) \}_\text{FunOrn} & : \text{SET} \\
\mu + \{ o \text{ (inv } i \text{) } \} T^+ \mu \text{orn} k \rightarrow \{ T^+ \}_\text{FunOrn} \\
\mu + \{ o \text{ (inv } i \text{) } \} T^+ \mu \text{orn} k \rightarrow \{ T^+ \}_\text{FunOrn} \\
\{ 1 \}_\text{FunOrn} & \rightarrow 1
\end{align*}
\]

9.31 We want our ornamented function to be *coherent* with respect to the base function we started from: for a function \( f : \mu D \rightarrow \mu E \), the ornamented function

\[
f^+ : \mu \{ o_D \}_\text{orn} \rightarrow \mu \{ o_E \}_\text{orn}
\]

is said to be coherent with \( f \) if the following diagram commutes:

\[
\begin{array}{ccc}
\mu \{ o_D \}_\text{orn} & \xrightarrow{f^+} & \mu \{ o_E \}_\text{orn} \\
\text{forget } o_D & \downarrow & \text{forget } o_E \\
\mu D & \xrightarrow{f} & \mu E
\end{array}
\]

Or, equivalently in type theory:

\[
\forall x^+ : \mu \{ o_D \}_\text{orn} i. f \ (\text{forget } o_D x^+) = \text{forget } o_E \ (f^+ x^+)
\]

This captures our intuition that the lifted function \( f^+ \) behaves similarly to the base function \( f \), only that it also carries the extra-information introduced by the ornament \( o_D \) over to the ornament \( o_E \). Coherence states that this extra-step does not interfere with its core operational behavior, which is specified by \( f \).

9.32 Definition (Coherence). This definition of coherence generalises to any arity. We define it by induction over the code of functional ornaments:

\[
\begin{align*}
\text{Coherence}(T^+ : \text{FunOrn } T)(f : \{ T \}_\text{Type}) & (f^+ : \{ T^+ \}_\text{FunOrn}) : \text{SET} \\
\text{Coherence} (\mu + \{ o \text{ (inv } i \text{) } \} T^+) & \rightarrow f^+ \rightarrow \\
\forall x^+ : \mu \{ o \}_\text{orn} k. \text{Coherence } T^+ (f \ (\text{forgetOrn } x^+)) (f^+ x^+) \\
\text{Coherence} (\mu + \{ o \text{ (inv } i \text{) } \} T^+) & \rightarrow \\
x = \text{forgetOrn } x^+ \times \text{Coherence } T^+ x^+ xs^+
\end{align*}
\]

9.33 Example (Ornamenting *type< to describe lookup*). In Section 9.1, we have identified the ornaments taking the type of \( \_ < \_ \) to the type of *lookup*. We ornament \( \text{Nat} \) to \( \text{List} A \) (Example 8.25), and \( \text{Bool} \) to *Maybe* \( A \) (Example 8.22). From there, the functional
ornament describing the type of the \texttt{lookup} function is as follows:

\begin{verbatim}
  typeLookup : FunOrn type<
  typeLookup \mapsto \mu + {\text{id}_{\text{Nat}}} * \mapsto \mu + {\text{ListO} A * } \mapsto \mu + {\text{MaybeO} A * } \times 1
\end{verbatim}

The reader checks that \([\text{typeLookup}]_{\text{FunOrn}}\) gives us the type of the \texttt{lookup} function, up to a multiplication by 1. Also, unfolding the coherence condition gives the desired property:

\begin{equation}
\text{Coherence typeLookup } (\_ < \_ ) \leadsto \lambda f^+ : [\text{typeLookup}]_{\text{FunOrn}}.
\forall n : \text{Nat}. \forall xs : \text{List A}. \text{isJust} \ (f^+ n xs) = n < \text{length xs}
\end{equation}

\textbf{9.34 Remark.} This equation is not specifying the \texttt{lookup} function: it is only establishing a computational relation between \_ < \_ and a candidate lifting \(f^+_+\), for which \texttt{lookup} is a valid choice. However, one could be interested in other functions satisfying this coherence property and they would be handled by our system just as well.

\textbf{9.35 Example} (Ornamenting \texttt{type}+ to describe \_ ++ \_). The functional ornament of \texttt{type}+ relies solely on the ornamentation of \texttt{Nat} into \texttt{List A}:

\begin{verbatim}
  type++ : FunOrn type+
  type++ \mapsto \mu + {\text{ListO} A * } \mapsto \mu + {\text{ListO} A * } \mapsto \mu + {\text{ListO} A * } \times 1
\end{verbatim}

Again, we check that \([\text{type++}]_{\text{FunOrn}}\) is indeed the type of \_ ++ \_ while the coherence condition \text{Coherence type++ } (\_ + \_ ) correctly captures our requirement that appending lists preserves their lengths. As before, the list append function is not the only valid lifting: one could for example consider a function that reverses the first list and appends it to the second one.

\textbf{9.2.3. Patches}

\textbf{MODEL:} Chapter9.Patch

(9.36) By definition of a functional ornament, the lifting of a base function \(f : [T]_{\text{Type}}\) is a function \(f^+\) of type \([T^+]_{\text{FunOrn}}\) satisfying the coherence property \text{Coherence} \(T^+ f\). To implement a lifting that is coherent, we might ask the user to first implement the lifting \(f^+\) and then prove its coherence. However, as discussed in Remark 9.15, this fails to harness the power of dependent types when implementing \(f^+\), this weakness being then paid off by tedious proof obligations. To overcome this limitation, we define the notion of \text{Patch} as the type of all the functions that are coherent by construction.

\textbf{9.37 Remark.} We are looking for an isomorphism here: we define patches in such a way that they are in bijection with the liftings satisfying a coherence property. Put otherwise, we want that:

\begin{equation}
\text{Patch} \; T \; T^+ f \cong (f^+ : [T^+]_{\text{FunOrn}}) \times \text{Coherence} \; T^+ f \; f^+
\end{equation}
In this chapter, we constructively use this bijection in the left-to-right direction: having implemented a patch $f^+$ of type $\text{Patch } T^+ f$, we show how we can extract a lifting together with its coherence proof.

**9.39 Example (Patching $-<-$).** Before giving the general construction of a $\text{Patch}$, let us first work through our running example. After having functionally ornamented $-<-$ with $\text{typeLookup}$, the lifting function $f^+$ and coherence property can be represented by the following commuting diagram:

\[
\begin{array}{ccc}
\text{Nat} & \times & \text{List } A \\
\downarrow \text{id} & & \downarrow \text{length} \\
\text{Nat} & \times & \text{Nat} \\
\end{array}
\xrightarrow{f^+} 
\begin{array}{c}
\text{Maybe } A \\
\downarrow \text{isJust} \\
\text{Bool} \\
\end{array}
\]

In type theory, this is written as:

\[
(f^+: \text{Nat} \times \text{List } A \rightarrow \text{Maybe } A) \times \\
\forall m: \text{Nat}. \forall as: \text{List } A. m < \text{length } as = \text{isJust } (f^+ m as)
\]

Applying intensional choice, this is equivalent to:

\[
\cong (m: \text{Nat}) \times (n: \text{Nat}) \times (as: \text{List } A) \times \text{length } as = n \\
(ma: \text{Maybe } A) \times \text{isJust } ma = m < n
\]

Now, by definition of reornaments, we have that:

\[
(\text{as: } \text{List } A) \times \text{length } as = n \cong \text{Vec } A n \\
(\text{ma: } \text{Maybe } A) \times \text{isJust } ma = b \cong \text{IMaybe } A b
\]

Applying these isomorphisms, we obtain the following type, which we call the $\text{Patch}$ of the functional ornament $\text{typeLookup}$:

\[
\cong (m: \text{Nat}) \times (n: \text{Nat}) \times (vs: \text{Vec } A n) \rightarrow \text{IMaybe } A (m < n)
\]

This last type is thus equivalent to a pair of a lifting and its coherence proof.

Intuitively, the $\text{Patch}$ construction consists in turning the vertical arrows of the commuting diagram (9.2) into the equivalent reornaments. In type-theoretic terms, it turns the pairs of datatypes and their algebraically defined constraints into the equivalent reornaments. The coherence property of reornaments tells us that projecting the orna-
mented function down to its unornamented components gives back the base function.

By turning the projection functions into inductive datatypes, we enforce the coherence property directly by the index: we introduce a fresh index for the arguments (in Example 9.39, introducing \( m \) and \( n \)) and index the return types by the result of the unornamented function (in Example 9.39, indexing \( \text{IMaybe } A \) by the result \( m < n \)).

**9.41 Definition (Patch type).** We define the Patch type generically by induction over the functional ornament. Upon an argument (i.e. a code \( \mu \{ \circ \} \rightarrow \cdot \)), we introduce a fresh index and the reornament of \( o \). Upon a result (i.e. a code \( \mu \{ \circ \} \times \cdot \)), we ask for a reornament of \( o \) indexed by the result of the base function.

\[
\text{Patch} \quad (T : \text{Type}) \quad (T^+ : \text{FunOrn } T) \quad (f : [\!\! T \!\!]_{\text{Type}}) : \text{SET}
\]

\[
\text{Patch} \quad (\mu[D \{ u \ i \} \rightarrow T] \ (\mu^+\{ o \ \text{inv} \ i \} \rightarrow T^+)) \quad f \quad \mapsto (x : \mu D \{ u \ i \}) \rightarrow \mu \{ o \} \ (i, x) \rightarrow \text{Patch } T T^+ \ (f \ x)
\]

\[
\text{Patch} \quad (\mu[D \{ u \ i \} \times T] \ (\mu^+\{ o \ \text{inv} \ i \} \times T^+)) \quad (x, xs) \quad \mapsto \mu \{ o \} \ (i, x) \times \text{Patch } T T^+ \ xs
\]

\[
\text{Patch} \quad 1 \quad 1 \quad * \quad \mapsto \mathbb{1}
\]

**9.42 Example (Patch of typeLookup).** The type of the coherent liftings of \( - < - \) by typeLookup, as defined by the Patch of \( - < - \) by typeLookup, unfolds to:

\[
(m : \text{Nat})(m^+ : \mu \{ \text{idO} \text{Nat} \} \ m) \rightarrow (n : \text{Nat})(\text{vs} : \mu \{ \text{List } A \} \ n) \rightarrow \mu \{ \text{Maybe } A \} \ (m < n) \times \mathbb{1}
\]

**9.43 Remark.** \( \mu \{ \text{idO} \text{Nat} \} \ n \) is isomorphic to \( \mathbb{1} \): all the content of the datatype has been forced – the recursive structure of the datatype is entirely determined by its index – and detagged – the choice of constructors is entirely determined by its index, leaving no actual data in it. Hence, we discard this argument as computationally uninteresting. On the other hand, \( \{ \text{List } A \} \) and \( \{ \text{Maybe } A \} \) are, respectively, the previously introduced vectors and indexed option types.

**9.44 Example (Patch of type+).** Similarly, the Patch of \( - + - \) by type+ unfolds to the type of the vector append function

\[
(m : \text{Nat})(xs : \{ \text{List } A \} \ m) \rightarrow (n : \text{Nat})(ys : \{ \text{List } A \} \ m) \rightarrow \{ \text{List } A \} \ (m + n) \times \mathbb{1}
\]

where, again, the datatype \( \{ \text{List } A \} \) corresponds exactly to vectors.

**9.45 Lemma.** Following our Remark 9.37, we have that a Patch is isomorphic to the pair of a lifting and its coherence proof:

\[
\forall T : \text{Type}. \forall T^+ : \text{FunOrn } T. \forall f : [\!\! T \!\!]_{\text{Type}}. \quad \text{Patch } T T^+ \ f \cong (f^+ : [\!\! T^+ \!\!]_{\text{FunOrn}}) \times \text{Coherence } T^+ \ f \ f^+
\]

That is, our definition of the Patch type enforces that its inhabitants are exactly those liftings that are coherent by construction.
9. Functional Ornaments

Proof. For clarity, we shall only write the proof for arity one. The generalisation to multiple input and output arities is straightforward but laboriously verbose. So, from a base function $f$, we start with its lifting and the associated coherence property:

$$
(f^+: \mu [o_A]_{orn} \rightarrow \mu [o_B]_{orn}) \times \\
\forall a^+: \mu [o_A]_{orn}. \text{forget} o_B (f^+ a^+) = f (\text{forget} o_A a^+)
$$

Applying intensional choice, we obtain the following equivalent type:

$$
\cong (a: \mu A) \times (a^+: \mu [o_A]_{orn}) \times \text{forget} o_A a^+ = a \rightarrow (b^+: \mu [o_B]_{orn}) \times \text{forget} o_B b^+ = fa
$$

Then, we can simply use the characterisation of a reornament to turn every pair $(x^+: \mu [o_X]_{orn}) \times \text{forget} o_X x^+ = t$ into the equivalent inductive type $\mu [o_X] t$

$$
\cong (a: \mu A) \times \mu [o_A] a \rightarrow \mu [o_B] (f a)
$$

which corresponds to the Patch type of this functional ornament.

\[\Box\]

9.46 Remark (When to index?). While these precisely indexed functions relieve us from the burden of theorem proving, this approach is not always applicable. For instance, if we were to implement a length-preserving list reversal function, our patching machinery would ask us to implement $vrev$:

$$
vrev (xs: \text{Vec } A n) : \text{Vec } A n
vrev \hspace{1em} \text{nil} \rightarrow \text{nil}
\hspace{1em} vrev \hspace{1em} \text{(cons } a \text{ vs}) \rightarrow \{(vrev \hspace{1em} vs) \hspace{1em} ++(\text{cons } a \text{ nil}) : \text{Vec } A (1 + n)\}
$$

To complete this goal calls for some proving in order to match up the types: we must appeal to the equational theory of addition. Here, the term we put in the hole has type $\text{Vec } A (n + 1)$ while the expected type is $\text{Vec } A (1 + n)$. The commutativity of addition is beyond the grasp of our type checker, which can only decide definitional identities.

Unless the type checker works up to equational theories, as done in CoqMT [Strub, 2010], the programmer is certainly better off using our machinery to generate the coherence condition (Section 9.2.2) and implement the lifting and its coherence proof manually, rather than using patches. However, this example gives a hint as to what can be seen as a “good” coherence property: because we want the type checker to do all the proving, the equations we rely on at the type level need to be definitionally true, either because our logic decides a rich definitional equality, or because we rely on operations that satisfy these identities by definition.
9.2. A Universe of Functions and their Ornaments

9.2.4. Patching and coherence

At this stage, we can implement the `ilookup` function exactly as we did in Section 9.1. From there, we now want to obtain the `lookup` function and its coherence certificate. More generally, having implemented a function satisfying the `Patch` type, we want to extract the lifting and its coherence proof. Perhaps not surprisingly, we obtain this construction by looking at the isomorphism of the previous section (Lemma 9.45) through our constructive glasses: indeed, since the `Patch` type is isomorphic to the set of liftings satisfying the coherence property, we effectively get a function taking every `Patch` to a lifting and its coherence proof. More precisely, we obtain the lifting by generalising the reornament-induced `forget` functions to functional ornaments while we obtain the coherence proof by generalising the reornament-induced `coherentOrn` theorem.

9.48 Definition (Patching). We call patching the action of projecting the coherent lifting from a `Patch` function. Again, it is defined by induction over the functional ornament. When ornamented arguments are introduced (i.e. with the code `µ + {o} →`), we simply patch the body of the function. This is possible because from `x + : µ [[0D]orn]`, we can forget the ornament to compute `f (forgetOrn x +)`, and we can also make the reornament to compute `f ++ (makeAlgOrn x +)`. When an ornamented result is to be returned (i.e. with the code `µ + {o} × `), we simply forget the reornamentation computed by the coherent lifting:

\[
patch (T + : FunOrn T) (f : T [Type]) (p : Patch T T + f) : T + [FunOrn] \\
patch (µ + {o (inv i)}) → T + f f ++ \mapsto \\
\lambda x +. patch (f (forgetOrn x +)) \\
(f ++ (forgetOrn x +) (makeAlgOrn x +)) \\
patch (µ + {o (inv i)} × T +) (x, xs) (x +, x +) \mapsto \\
(forgetOrn x +, patch T + xs x s +) \\
patch 1 * * \mapsto *
\]

9.49 Definition (Coherence of a patch). Extracting the coherence proof follows a similar pattern. We introduce arguments as we go, just as we did with patch. When we reach a result, we have to prove the coherence of the result returned by the patched function, which is a straightforward application of the `coherentOrn` theorem:

\[
coherence (T + : FunOrn T) (f : T [Type]) (p : Patch T T + f) : Coherence T + f p (patch T + p) \\
coherence (µ + {o (inv i)}) → T + f p \mapsto \\
\lambda x +. coherence T + (f (forgetOrn x +)) \\
(p (forgetOrn x +) (makeAlgOrn x +)) \\
coherence (µ + {o (inv i)} × T +) (x, xs) (x +, p) \mapsto \\
(coherentOrn x +, coherence T + xs p) \\
coherence 1 * * \mapsto *
\]
9. Functional Ornaments

9.50 Example (Obtaining \texttt{lookup} and its coherence, for free). This last step is a mere application of the \texttt{patch} and \texttt{coherence} functions. Hence, we define \texttt{lookup} as follows:

\[
\begin{align*}
\text{lookup} & : \texttt{typeLookup} \texttt{FunOrn} \\
\text{lookup} & \mapsto \texttt{patch typeLookup} (\sim \sim) \texttt{ilookup}
\end{align*}
\]

And we get its coherence proof, here spelled in full:

\[
\begin{align*}
\text{cohLookup} (n : \texttt{Nat}) (xs : \texttt{List A}) : \texttt{length} (\text{lookup} n \text{\textit{xs}}) = n < \texttt{length} \text{\textit{xs}} \\
\text{cohLookup} n xs \mapsto \texttt{coherence typeLookup} (\sim \sim) \texttt{ilookup} n xs
\end{align*}
\]

9.51 Remark (Code readability). The \texttt{lookup} function thus defined is rather daunting, especially for a potential user of that piece of code. However, we must bear in mind that \texttt{lookup} is in fact entirely specified by \texttt{ilookup}: there is no point in inspecting the definition of \texttt{lookup}. In a programming environment, we could imagine a syntactic device akin to our informal syntax for ornaments. For instance, we would state that \texttt{lookup} is a functional ornamentation of \(\sim \sim\). This would lead us to – transparently – implement \texttt{ilookup} in lieu of \texttt{lookup}.

9.52 Example (Obtaining \(\sim + + \sim\) and its coherence, for free). Assuming that we have implemented the coherent lifting \texttt{vappend}, we obtain concatenation of lists and its coherence proof by simply running our generic machinery:

\[
\begin{align*}
++ & : \texttt{type++} \texttt{FunOrn} \\
++ & \mapsto \texttt{patch type++} (\sim + \sim) \texttt{vappend} \\
\text{coh++} (xs : \texttt{List A}) (ys : \texttt{List A}) : \texttt{length} (\text{\textit{xs}} ++ \text{\textit{ys}}) = (\text{\texttt{length}} \text{\textit{xs}}) + (\text{\texttt{length}} \text{\textit{ys}}) \\
\text{coh++} xs ys & \mapsto \texttt{coherence type++} (\sim + \sim) \texttt{vappend} \text{\textit{xs}} \text{\textit{ys}}
\end{align*}
\]

(9.53) Looking back at the pedestrian construction of Section 9.1, we can measure the progress we have made: while we had to entirely duplicate the type signature of \texttt{lookup} and its coherence proof, we can now write down a functional ornament and these are generated for us. This is not just convenient: a functional ornament establishes a strong connection between two functions. By pinning down this connection with the universe of functional ornaments, we turn this knowledge into an effective object that can be manipulated and reasoned about within type theory.

We make use of this concrete object when we construct the \texttt{Patch} induced by a functional ornament: this is again a construction that is generic now, while we had to tediously (and perhaps painfully) construct it in Section 9.1. Similarly, we get patching and extraction of the coherence proof for free now, while we had to manually fiddle with several projection and injection functions.

(9.54) We presented \texttt{Patch} as the type of the liftings coherent by construction. As we have seen, its construction and further projection down to a lifting is now entirely automated, hence effortless. This is a significant step forward: we could either implement \texttt{lookup}
and then prove it coherent, or we could go through the trouble of manually defining carefully indexed types and write a function correct by construction.

We have now made this second alternative just as accessible as the first one. And, from a programming perspective, the second approach is much more appealing. In a word, we have made an appealing technique extremely cheap!

9.55 Remark (No meta-theory). We must reiterate that none of the above constructions involve extending the type theory: using our universe of datatypes, functional ornaments are internalised as a few generic programs and inductive types.

For systems such as Agda, Coq, or an ML with GADTs, this technology would need to be provided at the meta-level. However, the fact that our constructions type check in our system suggests that adding them at the meta-level is consistent with a pre-existing meta-theory.

9.3. Lazy Programmers, Smart Constructors

MODEL: Chapter9.Lift

(9.56) In our journey from \(\prec\prec\) to lookup, we had to implement the ilookup function. It is instructive to put \(\prec\prec\) and ilookup side-by-side (Figure 9.2). First, both functions follow the same recursion pattern: case analysis over \(n/\text{vs}\) followed by induction over \(m\). Second, the returned constructors are related through the Maybe ornament: knowing that we have returned true or false when implementing \(\prec\prec\), we deduce which of just or nothing will be used in ilookup. Interestingly, the only unknown, hence the only necessary input from the user, is the \(a\) in the just case: this is precisely the information that has been introduced by the Maybe ornament.

(9.57) In this section, we are going to leverage our knowledge of the definition of the base function – such as \(\prec\prec\) – to guide the implementation of the coherent lifting – such as ilookup: instead of re-implementing ilookup by duplicating most of the code of \(\prec\prec\), the user indicates what to transport and only provides the strictly-necessary inputs. We are primarily interested in transporting two forms of structure:

Recursion pattern: if the base function is a catamorphism \((a)\) and the user provides us with a coherent algebra \(\hat{a}\) of \(a\), we construct the coherent lifting \((\hat{a})\) of \((a)\);
9. Functional Ornaments

**Returned constructor:** if the base function returns a constructor \( C \) and the user provides us with a coherent extension \( \hat{C} \) of \( C \), we construct the coherent lifting of \( C \).

We shall formalise what we understand by being a coherent algebra and a coherent extension below. The key idea is to identify the strictly-necessary inputs from the user, helped in that by the ornaments. It is then straightforward to build the lifted objects, automatically and generically.

9.3.1. Transporting recursion patterns

When transporting a function, we are very unlikely to change the recursion pattern of the base function. Indeed, the very reason why we can do this transformation is that the lifting uses exactly the same underlying structure to compute its results. Hence, most of the time, we could just ask the computer to use the recursion pattern induced by the base function: the only task left to the user will be to give an algebra.

**Example (Lifting a catamorphism).** To understand how we transport recursion patterns, let us look again at the coherence property of liftings, but this time specialising to a function that is a catamorphism:

\[
\begin{array}{cc}
\mu \eval{D} & \mu \eval{E} \\
\mu \eval{D} \circ \忘 \ & \mu \eval{E} \\
\end{array}
\]

By the fold-fusion theorem [Bird and de Moor, 1997], it is sufficient (but not necessary) to work on the algebras, where we have the following diagram:

\[
\begin{array}{cc}
\mu \eval{D} \circ \忘 \ & \mu \eval{E} \circ \忘 \\
\mu \eval{D} \circ \忘 \ & \mu \eval{E} \circ \忘 \\
\end{array}
\]

Now, we would like to find an algebra \( \hat{\alpha} \) such that its catamorphism gives us a function of the Patch type.

**Example (Lifting isSuc).** To illustrate our approach, let us work through a concrete example: we derive \( \text{hd} : \text{List} A \to \text{Maybe} A \) from \( \text{isSuc} : \text{Nat} \to \text{Bool} \) by transporting the
algebra. For the sake of argument, we artificially define \textit{isSuc} by a catamorphism:

\[
\text{isSuc} \ (n : \text{Nat}) : \text{Bool} \\
\text{isSuc} \ n \mapsto ((\text{isSucAlg}) \ n) \\
\]

where

\[
\text{isSucAlg} \ (xs : [\text{NatD} \ \text{Bool}]) : \text{Bool} \\
\text{isSucAlg} \ ['0] \mapsto \text{false} \\
\text{isSucAlg} \ ['\text{suc} \ xs] \mapsto \text{true}
\]

Our objective is thus to define the algebra for \textit{hd}, which has the following type

\[
\text{hdAlg} : [\text{ListD} \ A] \ \text{Maybe} \ A \rightarrow \text{Maybe} \ A
\]

such that its catamorphism is coherent. By the fold-fusion theorem, it is sufficient for \text{hdAlg} to satisfy the following condition:

\[
\forall ms : [\text{ListD} \ A] \ \text{Maybe} \ A . \ \\
\text{isJust} \ (\text{hdAlg} \ ms) = \text{isSucAlg} \ (\text{forgetNT} \ (\text{ListO} \ A) \ (\text{ListD} \ A \ \text{isJust} \ ms))
\]

Following the same methodology we applied to define the \textit{Patch} type, we can massage the type of \text{hdAlg} and its coherence condition to obtain an equivalent definition enforcing the coherence by indexing. In this case, we obtain the type:

\[
\text{lift}_{\text{Alg}} \text{hdAlg} \triangleq [\text{VecD} \ A] \ (\lambda n'. \ \text{IMaybe} \ A \ (\text{isSuc} \ n')) \ n \rightarrow \text{IMaybe} \ A \ (\text{isSuc} \ n)
\]

This construction generalises to any functional ornament.

\textbf{9.62 Definition (Coherent algebra).} Let \(\alpha\) be an algebra

\[
\alpha : (k : K) \rightarrow [D] \ (\lambda - . \ [T]_{\text{Type}}) \ k \rightarrow [T]_{\text{Type}}
\]

together with an ornament \(\alpha D : \text{orn} \ D \ u \ u\) and a functional ornament \(T^+ : \text{FunOrn} \ T\).

We define the \textit{coherent algebras} for \(\alpha\) to be the inhabitants of the type:

\[
\text{lift}_{\text{Alg}} \alpha \triangleq [\text{VecD} \ A] \ (\lambda n'. \ \text{IMaybe} \ A \ (\text{isSuc} \ n')) \ n \rightarrow \text{IMaybe} \ A \ (\text{isSuc} \ n)
\]

\textbf{9.63 Definition (Lifting of coherent algebra).} Constructively, we get that coherent algebras induce coherent liftings, by the catamorphism of the coherent algebra:

\[
\text{lift-fold} \ (\alpha : (k : K) \rightarrow [D] \ (\lambda - . \ [T]_{\text{Type}}) \ k \rightarrow [T]_{\text{Type}}) \\
(\hat{\alpha} : \text{lift}_{\text{Alg}} \alpha) : \text{Patch} \ (\mu \{D \ (u \ i) \rightarrow T\} \ (\alpha) \ (\mu^+ \{o \ i \rightarrow T^+\}) \\
\text{lift-fold} \hat{\alpha} \hat{\alpha} \mapsto \lambda x . \lambda x^{++} . (\hat{\alpha}) \ x^{++}
\]
9. Functional Ornaments

9.64 Example (Transporting the recursion pattern of isSuc). We can now apply our generic machinery to transport isSuc to hd: using a high-level notation, we write the command of Figure 9.3(a) (Page 235). To this command, an interactive system would respond by automatically generating the algebra, as shown in Figure 9.3(b). In the low-level type theory, this would elaborate to the following term:

\[
\text{ihd} \left(\text{vs : Vec A n}\right) : \text{IMaybe} A (\text{isSuc n})
\]

where

\[
\text{ihdAlg} \left(\text{vs : \left[\text{VecD A}\right] (\lambda n'. \text{IMaybe} A (\text{isSuc n'}) n)\right) : \text{IMaybe} A (\text{isSuc n})
\]

\[
\text{ihdAlg} \left[\text{nil} \mapsto \{?\}
\]

\[
\text{ihdAlg} \left[\text{cons a xs} \mapsto \{?\}
\]

9.65 Remark (High-level notation). Formalising the elaboration process from the high-level notation to the low-level type theory is beyond the scope of this thesis. The readers will convince themselves that the high-level notation contains all the information necessary to reconstruct a low-level term. We use the high-level syntax to describe our transformations, with the understanding that it builds a low-level, well-typed term.

The treatment of induction is essentially the same, as hinted at by the fact that induction can be reduced to a catamorphism [Hermida and Jacobs, 1998, Fumex, 2012]. We first define the coherent inductive step and deduce an operation lifting induction principles:

9.66 Definition (Coherent inductive step). Let \(\alpha\) be an inductive step

\[
\alpha : (k : K) (xs : J D (\mu D) k) \rightarrow \Box D (\lambda - . [T]_{\text{Type}}) xs \rightarrow [T]_{\text{Type}}
\]

together with an ornament \(o : D \rightarrow [o]\) and a functional ornament \(T^+ : \text{FunOrn} T\).

We define the coherent inductive step for \(\alpha\) to be the inhabitants of the type:

\[
\text{lift}_{\text{IH}}^\alpha \triangleq (i : I)(t : \mu D (u i))(xs : [o]) (\mu [o]) (i, t)) \rightarrow
\]

\[
\Box [o] (\lambda (i, t). \text{Patch} T ((\text{induction } \alpha) t T^+) xs \rightarrow \text{Patch} T ((\text{induction } \alpha) t T^+ T^+)
\]

9.67 Definition (Lifting of inductive step). As for algebras, a coherent inductive step \(\hat{\alpha}\) induces a coherent lifting, by merely applying the induction:

\[
\text{lift-ind} (\hat{\alpha} : \text{IH} : (k : K) (xs : \Box D (\mu D) k) \rightarrow \Box D (\lambda - . [T]_{\text{Type}}) xs \rightarrow [T]_{\text{Type}})
\]

\[
(\hat{\alpha} \text{lift}_{\text{IH}}^\alpha) : \text{Patch} (\mu D (u i) \rightarrow T) ((\text{induction } \alpha) (\mu^+[o i] \rightarrow T^+))
\]

\[
\lambda x. \lambda x^+ \text{iduction } \hat{\alpha} x^+
\]

9.68 Definition (Lifting of case analysis). Lifting case analysis is trivial, since it is derivable from induction by stripping out the induction hypotheses (Section 7.3.1).
9.3. Lazy Programmers, Smart Constructors

lift-case \((\alpha : (k : K)(xs : [D]) (\mu D) k) \rightarrow [T]_{Type}\)

\((\hat{\alpha} : lift\_lh) \in Patch\ (\mu D (u i) \rightarrow T)\)

(iinduction \((\lambda xs \rightarrow . \alpha xs)\))

\((\mu^+ \{o i\} \rightarrow T^+))\)

lift-case \(\alpha \hat{\alpha} \mapsto \rightarrow\) lift-ind \((\lambda xs \rightarrow . \alpha xs)\) \((\lambda xs \rightarrow . \hat{\alpha} xs)\)

9.69 Example (Transporting the recursion pattern of \(- < -\)). To implement ilookup, we use lift-case to transport the case analysis on \(n\):

\[
\begin{align*}
ilookup : & \text{Patch type} < \text{typeLookup} \rightarrow < - \\
\text{ilookup} & m \text{ im} \ n \ vs \ \llift \ \text{lift-case} \\
\text{ilookup} & m \text{ im} \ 0 \ \text{nil} \ \{?\} \\
\text{ilookup} & m \text{ im} (\text{suc} \ n) (\text{cons} \ a \ vs) \ \{?\}
\end{align*}
\]

Followed by lift-ind to transport the induction over \(m\):

\[
\begin{align*}
ilookup : & \text{Patch type} < \text{typeLookup} \rightarrow < - \\
\text{ilookup} & m \text{ im} \ n \ vs \ \llift \ \text{lift-case} \\
\text{ilookup} & m \text{ im} \ 0 \ \text{nil} \ \{?\} \\
\text{ilookup} & m \text{ im} (\text{suc} \ n) (\text{cons} \ a \ vs) \ \llift \ \text{lift-ind} \\
\text{ilookup} & 0 \ 0 \ 0 \ \text{nil} \ \{?\} \\
\text{ilookup} (\text{suc} \ m) (\text{suc} \ im) \ 0 \ \text{nil} \ \{?\}
\end{align*}
\]

The interactive nature of this construction is crucial: the user types in the \(\llift\) command together with the action to be carried out, while the computer does all the heavy lifting and generates the resulting patterns.

9.70 Example (Transporting the recursion pattern of \(- + -\)). In order to implement the concatenation of vectors, we can also benefit from our generic machinery. We simply have to instruct the machine that we want to lift the case analysis used in the definition of \(- + -\) and the computer comes back to us with the following goals:

\[
\begin{align*}
vappend : & \text{Patch type} + \text{type} ++ \rightarrow + - \\
vappend & m \ xs \ n \ ys \ \llift \ \text{lift-case} \\
vappend & 0 \ \text{nil} \ n \ ys \ \{?\} \\
vappend (\text{suc} \ m) (\text{cons} \ a \ xs) \ n \ ys \ \{?\}
\end{align*}
\]

9.3.2. Transporting constructors

(9.71) Just as the recursive structure, the returned values frequently mirror the original definition: we are often in a situation where the base function returns a given constructor and we would like to return its ornamented counterpart. Informing the computer that
we simply want to lift that constructor, it should fill in the parts that are already
determined and ask only for the missing information, i.e. the data newly introduced by the
ornament.

Recall that, when implementing the coherent lifting, we are working on the reorna-
mants of the lifting. Hence, when returning a constructor-headed value, we are simply
building an inhabitant of a reornament. When defining reornaments in Section 8.1.4,
we have shown that, thanks to deletion ornaments, a reornament can be decomposed
in two components:

- The extension that contains all the extra information introduced by the ornament;
- The recursive structure of the refined datatype that specifies the sort of its argu-
ments.

And no additional information is required: all the information provided by indexing
with the unornamented datatype is optimally used in the definition of the reornament.
Thus, there is absolutely no duplication of information.

**9.73 Definition** (Lifting of constructor). This clear separation of concerns is a blessing
for us: when lifting a constructor, we only have to provide the extension and the argu-
ments of the datatype, nothing more. In terms of implementation, this is as simple as:

```plaintext
lift-constructor (e:Extension (o i) xs)
  (a:[Structure (o i) xs e]orn (μ [o])))
  (t++:Patch T t T+)
: Patch (μ(D (μ i) }× T) (in xs,t) (μ+{o i}× T+)
```

**9.74 Remark.** The implementation of the lifting of constructors is straightforward: it is
merely packing the various components in pairs. We might wonder whether such a
trivial construction is worth considering. In fact, we achieve such a level of simplicity
thanks to a careful choice of definitions: we have crucially defined reornaments as a
product of an extension and a structure. An alternative definition would have made
this construction much more involved. By carefully choosing the “right” definition, we
obtain a simple but just as interesting lifting function.

**9.75 Example** (Transporting the constructors of isSuc). Let us finish the implementation
of hd from isSuc. Our task is simply to transport the true and false constructors along
the Maybe ornament. In a high-level notation, we would write the command shown
in Figure 9.3(c). The interactive system would then respond by generating the code
of Figure 9.3(d). The unit goals (1) are trivially solved, probably automatically by the
system. The only information the user has to provide is a value of type A, which is
required by the just constructor.

**9.76 Example** (Transporting the constructors of − < −). As for ilookup, we want to lift
the constructors true and false to the Maybe ornament. In a high-level notation, this
9.3. Lazy Programmers, Smart Constructors

would be represented as follows:

\[
\begin{align*}
ilookup & : \text{Patch type} \prec \text{typeLookup} \prec - \\
ilookup & (m \ im \ n \ vs) \xrightarrow{\text{lift}} \text{lift-case} \\
ilookup & (m \ im \ 0 \ nil) \xrightarrow{\text{lift}} \text{nothing} \star \\
ilookup & (m \ im \ (\text{suc} \ n) \ (\text{cons} \ a \ vs)) \xrightarrow{\text{lift}} \text{lift-ind} \\
ilookup & (0 \ 0 \ (\text{suc} \ n) \ (\text{cons} \ a \ vs)) \xrightarrow{\text{lift}} \text{just} \{? : a\} \ A \star \\
ilookup & (\text{suc} \ m) \ (\text{suc} \ im) \ (\text{suc} \ n) \ (\text{cons} \ a \ vs) \xrightarrow{?}
\end{align*}
\]

As before, in an interactive setting, the user would instruct the machine to execute the command \(\xrightarrow{\text{lift}}\) and the computer would come back with the skeleton of the expected inputs. Finishing the implementation of \(\text{ilookup}\) is now one baby step away:

\[
\begin{align*}
ilookup & : \text{Patch type} \prec \text{typeLookup} \prec - \\
ilookup & (m \ im \ n \ vs) \xrightarrow{\text{lift}} \text{lift-case} \\
ilookup & (m \ im \ 0 \ nil) \xrightarrow{\text{lift}} \text{nothing} \star \\
ilookup & (m \ im \ (\text{suc} \ n) \ (\text{cons} \ a \ vs)) \xrightarrow{\text{lift}} \text{just} \{? : a\} \ A \star \\
ilookup & (\text{suc} \ m) \ (\text{suc} \ im) \ (\text{suc} \ n) \ (\text{cons} \ a \ vs) \xrightarrow{?}
\end{align*}
\]

9.77 Example (Transporting the constructors of \(\prec + \prec\)). We can also benefit from the automatic lifting of constructors to fill out the \(\text{cons}\) case of vector append. We instruct the system that we want to lift the \(\text{suc}\) constructor, which results in the following goals:

\[
\begin{align*}
vappend & : \text{Patch type} \prec \text{type} \prec \prec \prec \prec \\
vappend & (m \ xs \ n \ ys) \xrightarrow{\text{lift}} \text{lift-case} \\
vappend & (0 \ nil \ n \ ys) \xrightarrow{\text{lift}} \text{cons} \{?\} \\
vappend & (\text{suc} \ m) \ (\text{cons} \ a \ xs) \ n \ ys \xrightarrow{\text{lift}} \text{cons} \{? : A\} \ [?] \\
vappend & (\text{suc} \ m) \ (\text{cons} \ a \ xs) \ n \ ys \xrightarrow{\text{lift}} \text{cons} \ a \{vappend \ m \ xs \ n \ ys\}
\end{align*}
\]

9.78 Remark (About an interactive system). To convey our message to the reader, we have used a very high-level language of liftings, without giving much information about its implementability, or even a formal specification. It would certainly be inter-
testing to elaborate on such a language extension. However, more easily implementable alternatives are worth considering. For instance, one could choose to implement a semi-decision procedure à la Agsy [Lindblad and Benke, 2004] that attempts to automatically lift a function. The lifting operations we have described thus serve as a precise language in which to express the lifting problem, and over which to compute its solution. Our high-level syntax is highly anticipative, and may remain unimplemented. However, it is only necessary for the purpose of conveying high-level intuitions to the reader, and to keep us from flooding these pages with lambda terms.
9.3. Lazy Programmers, Smart Constructors

(a) Request lifting of algebra (user input):

\[
\begin{align*}
&\text{ihd} \quad (\text{vs} : \text{Vec } A \ n) : \text{IMaybe } A \ \text{isSuc } n \\
&\text{ihd} \quad \text{lift} \quad \text{lift-fold} \\
&\{ ? \}
\end{align*}
\]

(b) Result of lifting the algebra (system output):

\[
\begin{align*}
&\text{ihd} \quad (\text{vs} : \text{Vec } A \ n) : \text{IMaybe } A \ (\text{isSuc } n) \\
&\text{ihd} \quad \text{lift} \quad \text{lift-fold} \quad \text{where} \\
&\quad \text{ihdAlg} \quad (\text{vs} : \text{[VecD]} \ (\lambda n'. \text{IMaybe } A \ (\text{isSuc } n')) \ n) : \text{IMaybe } A \ (\text{isSuc } n) \\
&\quad \text{ihdAlg} \quad '\text{nil} \quad \text{lift} \quad \{ ? \} \\
&\quad \text{ihdAlg} \quad ('\text{cons } a \ \text{xs}) \quad \text{lift} \quad \{ ? \}
\end{align*}
\]

(c) Request lifting of constructors (user input):

\[
\begin{align*}
&\text{ihd} \quad (\text{vs} : \text{Vec } A \ n) : \text{IMaybe } A \ (\text{isSuc } n) \\
&\text{ihd} \quad \text{lift} \quad \text{lift-fold} \quad \text{where} \\
&\quad \text{ihdAlg} \quad (\text{vs} : \text{[VecD]} \ (\lambda n'. \text{IMaybe } A \ (\text{isSuc } n')) \ n) : \text{IMaybe } A \ (\text{isSuc } n) \\
&\quad \text{ihdAlg} \quad '\text{nil} \quad \text{lift} \quad \{ ? \} \\
&\quad \text{ihdAlg} \quad ('\text{cons } a \ \text{xs}) \\
\end{align*}
\]

(d) Result of lifting constructors (system output)

\[
\begin{align*}
&\text{ihd} \quad (\text{vs} : \text{Vec } A \ n) : \text{IMaybe } A \ (\text{isSuc } n) \\
&\text{ihd} \quad \text{lift} \quad \text{lift-fold} \quad \text{where} \\
&\quad \text{ihdAlg} \quad (\text{vs} : \text{[VecD]} \ (\lambda n'. \text{IMaybe } A \ (\text{isSuc } n')) \ n) : \text{IMaybe } A \ (\text{isSuc } n) \\
&\quad \text{ihdAlg} \quad '\text{nil} \quad \text{lift} \quad \text{nothing} \quad \{ ? : A \} | \{ ? : 1 \} \\
&\quad \text{ihdAlg} \quad ('\text{cons } a \ \text{xs}) \quad \text{lift} \quad \text{just} \quad \{ ? : A \} | \{ ? : 1 \}
\end{align*}
\]

(e) type checked term (automatically generated from (d)):

\[
\begin{align*}
&\text{ihd} \quad (\text{vs} : \text{Vec } A \ n) : \text{IMaybe } A \ (\text{isSuc } n) \\
&\text{ihd} \quad \text{vs} \quad \text{lift-fold} \quad \text{isSucAlg} \ \text{ihdAlg} \quad \text{where} \\
&\quad \text{ihdAlg} \quad (\text{vs} : \text{[VecD]} \ (\lambda n'. \text{IMaybe } A \ (\text{isSuc } n')) \ n) : \text{IMaybe } A \ (\text{isSuc } n) \\
&\quad \text{ihdAlg} \quad '\text{nil} \quad \rightarrow \text{lift-constructor} \quad '\text{nil} \quad \text{lift} \quad \{ ? : 1 \} | \{ ? : 1 \} \\
&\quad \text{ihdAlg} \quad ('\text{cons } a \ \text{xs}) \quad \rightarrow \text{lift-constructor} \quad ('\text{suc } n) \quad \text{lift} \quad \{ ? : A \} | \{ ? : 1 \}
\end{align*}
\]

Figure 9.3.: Guided implementation of \( \text{ihd} \)

235
9. Functional Ornaments

Conclusion

In this chapter, we have generalised ornaments to functions: from a universe of function types, we define a functional ornament as the ornamentation of each of its inductive components. A function of the resulting type will be subject to a coherence property, akin to the ornamental forgetful map of ornaments. We have constructively presented this object by building a small universe of functional ornaments.

Using functional ornaments, we can tackle the question of transporting a function to its ornamented version in such way that the coherence property holds. Instead of asking our user to write cumbersome proofs, we define a Patch type as the type of all the functions that satisfies the coherence property by construction. Hence, we make extensive use of the dependently-typed programming machinery offered by the environment: in this setting, the type checker, that is the computer, is working with us to construct a term, not waiting for us to produce a proof.

Having implemented a function correct by construction, we then get, for free, the lifting and its coherence certificate. This is a straightforward application of the isomorphism between the Patch type and the set of coherent functions. These projection functions have been implemented in type theory by programming over the universe of functional ornaments.

To further improve code reuse, we provide two smart constructors to implement a Patch type: the idea is to use the structure of the base function to guide the implementation of the coherent lifting. Hence, if the base function uses a specific induction principle or returns a specific constructor, we make it possible for the users to specify that they want to lift this element one level up. This way, the function is not duplicated: only the new information, as determined by the ornament, is necessary.

We believe that this is a first yet interesting step towards code reuse for dependently-typed programming systems. With ornaments, we were able to organise datatypes by their structure. With functional ornaments, we are now able to organise functions by their structure-preserving computational behaviour. Besides, we have developed some appealing automation to assist the implementation of functional ornaments, without any proving required, hence making this approach even more accessible.

Related work

In their work on realisability and parametricity for Pure Type Systems, Bernardy and Lasson [Bernardy and Lasson, 2011] have shown how to build a logic from a programming language. In such a system, terms of type theory can be precisely segregated based on their computational contribution and their logical contribution. In particular, the idea that natural numbers realise lists of the corresponding length appears in this system under the guise of vectors, the reflection of the realisability predicate. The strength of the realisability interpretation is that it is naturally defined on functions: while McBride [2013] and Atkey et al. [2012] only consider ornaments on datatypes, their work is the first, to our knowledge, to capture a general notion of functions realising – i.e. ornamenting – other functions.
Following the steps of Bernardy, Ko and Gibbons [2011] adapted the realisability interpretation to McBride’s universe of datatypes and explored the other direction of the Patch isomorphism, using reornaments to generate coherence properties: they describe how one could take list append together with a proof that it is coherent with respect to addition and obtain the vector append function. Their approach would shift neatly to our index-aware setting, where the treatment of reornaments is streamlined by the availability of deletion.

However, we prefer to exploit the direction of the isomorphism that internalises coherence: we would rather use the full power of dependent types to avoid explicit proof. Hence, in our framework, we simultaneously induce list append and implicitly prove its coherence with addition just by defining vector append. Of course, which approach is appropriate depends on one’s starting point. Moreover, our universe of functions takes a step beyond the related work by supporting the mechanised construction of liftings, leaving to the user the task of supplying a minimal patch. Our framework could easily be used to mechanise the realisability predicate constructions of Bernardy and Lasson [2011], Ko and Gibbons [2011].
10. Conclusion

(10.1) In this thesis, we have designed a type theory for generic programming. It is based on a low-budget Martin-Löf type theory, with $\Pi$-types, $\Sigma$-types, and finite sets. We left equality, both definitional and propositional, underspecified so as to be more widely applicable. We obviously have an eye for observational type theory, or a homotopic treatment of equality. But none of these powerful equalities are required: they are just bonus.

We have extended this type theory with inductive types and families, following a universe-based approach. A universe enables an intensional characterisation of inductive families. It also implies that our presentation of inductive types is closed. No datatype is ever “introduced” in the type theory, it already implicitly existed: we merely made it explicit by naming its code.

(10.2) The code of this universe is itself an inductive type: we are therefore able to describe it into itself. To do so, we have delineated the minimal amount of meta-theory necessary to obtain inductive types: their fixpoint. The rest is bootstrapped. We obtain a lightweight meta-theory of inductive types.

Consequently, we were able to treat the code of inductive definitions just like any other datatype: in particular, we use the elimination principle of inductive types to compute over the grammar of datatypes. This enables generic programming, with no special support but this particular setup of the type theory. We built upon this calculus to give a few examples of generic programs.

(10.3) The use of codes instead of inductive definitions could scare away a potential user. While we certainly have a system for generic programming, we also want specific programming to remain natural and free from low-level encodings. We have seen how, with a blend of bidirectional type checking and elaboration, we can grow a programming language on top of our austere calculus. Consequently, we expect specific programming in our system to be just like programming in a non-generic system, such as Coq and Agda.

(10.4) We have adapted ornaments to our presentation of datatypes. We obtain a syntactic object characterising structure-preserving operations on inductive types. Doing so, we take advantage of our categorical understanding of ornaments to give a more abstract presentation of the type-theoretic constructions. We have generalised ornaments to functions, thus relating structure-preserving functions. This opens up interesting applications for reusing code across domain-specific datatypes. We gave a few examples of automatic code transformations, lifting a function across its ornamentation. Crucially, none of these constructions require extending the type theory. The theory of ornaments and functional ornaments is entirely developed within type theory. Its validity is thus
10. Conclusion

simply justified by type checking.

Finally, we have striven to give a categorical model to the objects studied in this thesis. Our objective was to develop a mathematical toolkit with which to carry more abstract reasoning than is possible in type theory. We have thus established an equivalence between our universe of datatypes and containers. We also proved an equivalence between ornaments and Cartesian morphisms of containers. As a result, we were able to explore the realm of ornaments using the more abstract Cartesian morphisms. In the process, we have discovered a few interesting objects (Section 8.3), such as the cobase-change ornament (Example 8.109) and the pullback of ornaments (Example 8.112).

10.1. Further Work

A very natural next step is to extend our universe of datatypes. Let us explore a few possibilities. Firstly, we have only considered the inductive fragment: as we introduced a least fixpoint operator \( \mu \), we could introduce a greatest fixpoint \( \nu \). We would then equip this set former with its terminal algebra semantics. However, it seems that introducing greatest fixpoint would put some more constraints on our notion of equality [McBride, 2009], beyond what we were willing to consider in this thesis.

Secondly, we could extend our universe with codes for internal fixpoints, as did Morris et al. [2004]. The external fixpoint operator approach we have taken here makes datatypes such as rose trees, i.e. a tree with lists of subtrees, more cumbersome than they should be. Moreover, if we allow the alternation of least and greatest fixpoints, we expect to gain types that are not readily encoded with one external fixpoint.

Thirdly, it would be fascinating to extend our universe with dedicated support for syntax with binding. So far, we have worked with endofunctors on \( \text{SET} \) and slices of \( \text{SET} \). Following Fiore et al. [1999], we are interested in representing endofunctors on the presheaf of finite sets. Such a binding structure would thus come equipped with a monadic substitution operation as described by Altenkirch and Reus [1999]. By building the type theory around this universe, we might be able to obtain a definitional equality working modulo substitution.

Finally, it is very tempting to move to inductive-recursive definitions [Dybjer and Setzer, 2001], allowing us to interleave the definition and interpretation of data in intricate and powerful ways. This interleaving is particularly useful when reflecting the syntax of dependent type systems. The potential for generic programming on such objects seems limitless.

Much work remains to be done on the side of elaboration. For instance, we must extend the elaboration of datatypes to the elaboration of ornaments. Also, we would like to adapt our syntax to the definition of coinductive types. Finally, an interesting challenge would be to internalise the elaboration process itself in type theory, hence obtaining a correct-by-construction translation. Overall, despite our attempt at systematising the elaboration process, it feels extremely laborious. We would like to develop a general framework for type-directed elaboration.

In our study of ornaments, we have barely scratched the surface of containers: most
of its structure remains unexploited. Pursuing this exploration might lead to novel and interesting ornamental constructions. Also, our definition of ornaments in terms of containers might be limiting. One can wonder if a more abstract criterion could be found for a larger class of functors. For instance, the functor \( -1_C : [C, D] \to D \) is a fibration for \( D \) pullback-complete and \( C \) equipped with a terminal object \( 1_C \). Specialised to the slices of an LCCC, the Cartesian morphisms are exactly our ornaments. What about the general case?

Finally, there has been much work recently on homotopy inductive types [Awodey et al., 2012]. Coincidentally, the formalism used in these works is based on W-types, i.e. the fixpoint of containers. It would be interesting to study what ornaments we could express in this framework.

Concerning functional ornaments, we have deliberately chosen a simple universe of functions, which we would like to extend in various directions. Adding support for higher-order functions, type dependency (\( \Pi \)-types and \( \Sigma \)-types) but also non-inductive sets is a necessary first step. Inspired by Bernardy and Lasson [2011], we would like to add a parametric quantifier: in the implementation of \( \text{lookup} \), we would mark the index \( A \) of \( \text{Vec} \ A \) and \( \text{IMaybe} \ A \) as parametric so that in the \( \text{cons} \ a \) case, the \( a \) could be automatically carried over.

The universe of functional ornaments could be extended as well, especially once the universe of functions has been extended with dependent quantifiers. For instance, we want to consider the introduction and deletion of quantifiers, as we are currently doing on datatypes. Whilst we have only looked at least fixpoints, we also want to generalise our universe with greatest fixpoints and the lifting of co-inductive definitions.

Further, our framework relies crucially on the duality between a reornament and its ornament presentation subject to a proof. We cross this isomorphism in both directions when we project the lifting from the coherent lifting. In practice, this involves a traversal of each of the input datatypes and a traversal of each of the output datatypes. However, computationally, these traversals are identities: the only purpose of these terms is at the logical level, for the type checker to fix the types. We are looking at transforming our library of smart constructors into a proper domain-specific language (DSL). This way, implementing a coherent lifting would consist in working in a DSL for which a compiler could optimise away the computationally irrelevant operations.

10.2. Implementation Work

This thesis has grown out of the Epigram2 system. A prototype of Epigram2 implemented an early universe of datatypes, upon which this thesis is based. It was organised around a bidirectional type checker. It also featured a bootstrapped presentation of inductive definitions. However, much to our regret, this prototype had to be abandoned and the implementation effort disrupted. Consequently, the constructs presented in this thesis have been developed in an Agda model. In the future, we would like to implement a language that supports the programming paradigm we advocate.

A first direction of development is the elaboration of inductive definitions. In Epi-
gram2, Peter Morris had implemented a tactic elaborating an earlier form of inductive definition down to our universe of codes. We would like to pursue this implementation, adapted to our setting. We would extend it to support a deriving mechanism. Also, we will have to generate the specialised induction principles – such as case analysis and course-of-value recursion – as well as constructions on constructors.

A second area of experimentation concerns the lifting of function across ornaments. Informally, we have presented a syntax that describes the lifting of constructors and recursors. We could try to formally describe this syntactic toolkit by extending the set of gadgets defined by McBride and McKinna [2004]. Alternatively, and perhaps more pragmatically, we could provide a decision procedure à la Agsy [Lindblad and Benke, 2004]. Users would pass their definition to a tool that would return a skeleton definition.

Finally, much legwork remains to be done on the front of usability: for convenience, we gave a relational specification of various elaborators. We are left with implementing these systems. Another usability question is whether or not we can protect our users from leaks of encoding during datatype-specific programming. Again, early experiments with Epigram2 suggest that it is, in principle, possible. Integrating these features without making generic programming prohibitively difficult is another question.

10.3. Epilogue

First and foremost, this thesis attempts to swallow inductive types in type theory. By reducing inductive definitions to type-theoretic objects, we follow the design principle of type theory: reflection. Indeed, if we believe in type theory as a complete mathematical system, we should strive to express the mathematical concepts we manipulate – here, inductive definitions – in that system itself. In terms of implementation, this amounts to implementing most of the inductive fragment in the type theory.

From a programming language perspective, this corresponds to bootstrapping. Here, we have focused on the inductive fragment. However, we are confident that type theory will eventually be fully bootstrapped [Barras, 2013]. This thesis gives a glimpse of tomorrow’s type theory, which would provide:

- a native support for generic programming, not necessarily restricted to inductive types;
- an elaboration of expressions, programs, and inductive definitions, implemented in type theory;
- a reflected representation of types, going beyond our first-order, simply-typed universe of functions;
- a framework for symbolically manipulating and defining terms, pushing further our reflections on inductives and the lifting library.

Being at the interface between type theory and category theory, this thesis was meant for both communities. To the type theorist, we offer a more semantic account of inductive types and constructions over them. We use the intuition thus gained to introduce new type-theoretic constructions. To the category theorist, we present a type theory, i.e.
Our approach can be summarised as *categorically-structured programming*. For practical reasons, we do not work on categorical objects directly: instead, we materialise these concepts through universes, thus reifying categorical notions through computational objects. Descriptions, ornaments, and functional ornaments are merely an instance of that interplay between categorical concepts and effective, type-theoretic universes. To help bridge the gap between type theory and category theory, we have provided the type theorist with concrete examples of the categorical notions and the category theorist with the computational intuition behind the type-theoretic objects.

Finally, this thesis was an exercise in generic programming. This exercise was guided by the idea that generic programming is *just* programming, as it should be. Indeed, generic programming simply consists in programming over some special mathematical structures, be they containers (datatype-generic programming), or their Cartesian morphisms (ornaments and functional ornaments). By internalising these structures in type theory, generic programs become first-class citizens.

This reduction of generic programming to programming with mathematical structures was made possible by a deeper understanding of the mathematics at play. This is a direct result of the dialogue we have established between type theory and category theory. This interaction has two components. On one hand, we have striven to create mathematical *knowledge* from operational phenomenons observed in type theory. On the other hand, we have turned the abstract understanding thus gained into novel programming *artefacts*.

Programming is a fascinating activity during which virtual constructions are created out of thin air. This process consists in describing, in a language intelligible by a computer, how to compute a result. On the other hand and paraphrasing Wigner [1960], mathematics is an *unreasonably effective* system for explaining *why* we have carved our objects in one way and not another. We have constructed this thesis around this dialogue between programs – the how? – and mathematics – the why?
# A. Overloaded Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="not1.png" alt="Notation" /></td>
<td><img src="desc1.png" alt="Description" /></td>
<td><img src="int1.png" alt="Interpretation" /></td>
</tr>
<tr>
<td><img src="not2.png" alt="Notation" /></td>
<td><img src="desc2.png" alt="Description" /></td>
<td><img src="int2.png" alt="Interpretation" /></td>
</tr>
<tr>
<td><img src="not3.png" alt="Notation" /></td>
<td><img src="desc3.png" alt="Description" /></td>
<td><img src="int3.png" alt="Interpretation" /></td>
</tr>
<tr>
<td><img src="not4.png" alt="Notation" /></td>
<td><img src="desc4.png" alt="Description" /></td>
<td><img src="int4.png" alt="Interpretation" /></td>
</tr>
<tr>
<td><img src="not5.png" alt="Notation" /></td>
<td><img src="desc5.png" alt="Description" /></td>
<td><img src="int5.png" alt="Interpretation" /></td>
</tr>
<tr>
<td><img src="not6.png" alt="Notation" /></td>
<td><img src="desc6.png" alt="Description" /></td>
<td><img src="int6.png" alt="Interpretation" /></td>
</tr>
</tbody>
</table>

```
\[ (D:\text{Desc}) : (X:\text{SET}) \rightarrow \text{SET} \]
\[ (D:\text{IDesc } i) : (I \rightarrow \text{SET}) \rightarrow \text{SET} \]
\[ (D:\text{func } i j) : (X : I \rightarrow \text{SET}) \rightarrow J \rightarrow \text{SET} \]
```

```
\[ \Box (D:\text{Desc}) : (P : X \rightarrow \text{SET}) (xs : [D] X) \rightarrow \text{SET} \]
\[ \Box (D:\text{func } i j) : (P : (i : I) \times \mu D i \rightarrow \text{SET}) (xs : [D] (\mu D) i) \rightarrow \text{SET} \]
```

```
\[ (\mu (D:\text{Desc}) : \text{SET} \]
\[ \mu (D:\text{func } i l) : I \rightarrow \text{SET} \]
```
Bibliography


Bibliography


B. Barras. Semantical Investigations in Intuitionistic Set Theory and Type Theories with Inductive Families. Habilitation à diriger les recherches, Université Paris 7 - Denis Diderot, 2013.


Bibliography


Bibliography


Bibliography


255
Bibliography


Index

\( \mu \), see Least fixpoint

Acyclicity, 172
Algebra, 75
   initial, 75
Algebraic ornament, 192
   coherence property, 205
Algebraic ornament by the ornamental algebra, see Reornament

Base function, 215
Base type, 215
Base-change container, 200
Bidirectional type checking, 47

Canonical lifting, 80, 189
   indexed, 92
Cartesian morphism
   container, 199
   ornament, 190
Cartesian natural transformation, 200
Catamorphism, 75
   generic, 126
   indexed, 91
Category
   algebra, 75
   Kleisli, 130
   slice, 87
Cobase-change container, 201
Coherent algebra, 229
Coherent inductive step, 230
Coherent lifting, 216
Constructions on constructors, 172
Container, 100
   interpretation, 102
   natural transformation, 200
Context validity, 27
Conversion
   typed, 30
   untyped, 41

Datatype-generic programming, 71
Datatype-specific programming, 84
Decidability of type checking, 44
Decision procedure
   type theory, 174
Derivable properties, 174
Deriving mechanism, 174
Described functor, 103
Description, 68
   indexed, 88
   tagged, 73
   tagged indexed, 97
Description label, 147
   indexed, 157
Detagging, see also Forcing, 190
Differentiability, 209

Elaboration, 48
   arguments, 152, 166
   choices, 164
   constraints, 168
   constructor, 84, 151, 165
   constructor choices, 151
   enumeration, 52
   enumeration elimination, 52
   enumeration index, 53
   indices, 169
   inductive definition, 148
   inductive families, 158
Index

patterns, 162
recursive arguments, 153, 167
tuple, 51
Enumeration, 37
elimination, 40
index, 38
Equality
judgmental, 30
propositional, 33
Equivalence
descriptions and containers, 106
ornaments and Cartesian morphisms, 202
Expressions, 48
Fold, 75
Forcing, see also Detagging, 190
Frame structure, 200
base-change, 200
cobase-change, 201
Free monad, 127, 128
indexed, 131
Functional ornament, 215
cohesion, 220
Functor
terminal object, 83
Generic operation, 129
Henry Ford principle, 96
Hierarchy of types, 137
Hole, 55
Identity type, 35
Induction principle, 79
indexed, 92
Inductive definitions
grammar, 146
Inductive families
grammar, 156
Inverse image, 104
Judgmental equality, 30
enumeration, 38
enumeration index, 39
tag, 36
Kleisli category, 130
Lambek’s lemma, 77
Least fixpoint, 77
indexed, 91
Least reflexive relation, 35
Let binding, 27
Level-parametric definition, 138
Levitation, 113
description, 120, 121
enumeration, 114
indexed description, 123
Lifting, 215
case analysis, 230
coherent, 215
coherent algebra, 229
constructor, 232
inductive step, 230
Lifting map, 82
indexed, 93
Lifting type, 219
Local definition, 27
Multi-sorted signature, 101
Mutually-inductive definition, 95
Next universe operator, 137
No confusion, 172
NuPRL, 34
Ornament, 186
algebraic, 192
Cartesian morphism, 190
cohesion, 194
derivative, 210
forgetful map, 191
frame structure, 208
horizontal composition, 207
identity, 206
pullback, 209
vertical composition, 207
Ornamental algebra, 191, 204
Patch type, 223
Index

Patching, 225
Polarity, 30
Polynomial functor, 102
Predicate transformer, 81
Recomputation, 194
Refinement type, 183
Remember, 194
Reornament, 195, 197
  extension, 196
  structure, 196
Set polymorphism, 117
Sigmas-of-sigmas, 67
Small II-type, 40
Strictly-positive family, 89
Strictly-positive functor, 70
Strong normalisation, 44
Structure, 183
Substitution, 26
Sum, 41
Sum-of-products, 67
Surjective pairing, 32
Tagged description, 73
Tags, 36
Terminal object functor, 83
Terms, 25
  enumeration, 37
  enumeration index, 38
  label, 55
  tag, 36
Type
  negative, 31
  positive, 31
Type checking, 49
Type synthesis, 48
Typing
  Church-style, 30
  Curry-style, 47
Typing judgment, 27
  enumeration, 37
  enumeration index, 38
  sum, 41
  tag, 36
UIP, see Unicity of identity proof
Unicity of identity proof, 34
Universe
  description, 68
  enumeration, 36
  functional ornament, 219
  indexed description, 89
  ornaments, 186
  types, 218
W-types, 136
Wellorderings, 136
Zipper data-structure, 209
Acknowledgements

I remember quite vividly my interview for this PhD position: my supervisor-to-be, Conor McBride, presented this project as “a walk in the minefield”. This was enough to convince me to embark on this journey. I must therefore thank Conor for his guidance and for my two legs. I also greatly appreciate the freedom he gave me to pursue my research interests and to pick my own tools. I’m particularly thankful to him for having accepted and supported my two internship applications.

Coincidentally, his words upon receiving my final draft were: “you’ll get a concentrated blast [of feedback] from me.” And a blast it was: I must thank him for his attention to detail, which have tremendously improved my dissertation. Knowing his taste for extended metaphors, I cannot shake the idea that our conversation has set a new record.

I would like to thank my examiners, Ralf Hinze and Clemens Kupke, for taking the time to read this dissertation and for providing many suggestions for improvement.

This dissertation is the fruit of an intellectual and human journey. Back in 2009, I was in ETH-Zürich, compiling domain-specific languages for the Barrelfish operating system. Little did I know about Martin-Löf type theory, inductive families, or generic programming, which were to be my daily diet in Glasgow. Along the way, I have met some truly amazing people. I am deeply indebted to them and this thesis reflects their influences upon me. I would like to take the following lines to thank those that made this adventure so enjoyable.

In the MSP circles, I must give special credit to Stevan Andjelkovic and Stuart Hannah for, respectively, their South Scandinavian and West Scottish sense of humour. This last year spent in Glasgow was absolutely hilarious, in large part thanks to them. I want to thank Lorenzo Malatesta for putting the paper of Gambino and Kock into my hands: this was absolutely instrumental for my research. I am also thankful to Guillaume Allais for his insightful feedback on my papers and the many pointers he gave me. I have learned a lot from discussions with our neighbours in Edinburgh, Ben Kavanagh and Ohad Kammar. Edwin Brady, James Chapman, Peter Morris, and Wouter Swierstra have also played a significant role in this thesis. They guided me to make the most of my PhD and provided me with a wealth of motivating examples for my research.

Finally, I am immensely grateful to Peter Hancock for his thought-provoking comments on my work but also for supporting and guiding me toward the completion of my dissertation. I thank him for these multiple occasions where he selflessly defused a mine I had stepped on. Last but not least, no words can express my gratitude to Clément Fumex: he has been an extremely supportive colleague and a great friend. I have learned a lot from him and he has strongly influenced my research. I have learned
Acknowledgements

from him most – if not all – of the category theory found in the previous pages. More
than a fellow PhD student, he has been a patient and insightful teacher.

This journey has also been influenced by friends from my “previous lives” at EPFL
and ETH. In particular, I owe to Ruzica Piskac and Damien Zufferey to have made
me realise that the MSP group, an unusually small pond, was perhaps a special place.
They planted that seed at the most crucial moment, which led me to visit the Barrelfish
team in Cambridge. I am most thankful to my Master’s supervisor Timothy Roscoe,
and to Andrew Baumann and Oriana Riva for their support to fly me off to the Pacific
Northwest. I am particularly indebted to Oriana for her insistence, and to Andrew for
the zeal with which he had polished my application.

This interlude at Microsoft Research Redmond was an amazing experience. I want
to thank my manager, Nikhil Swamy, and the F∗ team: Juan Chen, Cédric Fournet, Ben
Livshits, and Pierre-Yves Strub. I have very fond memories of my fellow interns too: it
was refreshing to be amongst such a high concentration of brains. My hiking buddies –
Louis Jachiet, Jason König, Magnus Madsen, and Mehdi Bouaziz – deserve a special
mention, for they hurled me to the top of the Cascades every week-end. Thanks so
much for the oxygen. Nik and Mehdi have made a lasting impression on me: their
patience, kindness, and enthusiasm is an inspiration for me.

From one Microsoft Research, I jumped to the next, in Cambridge this time. There, I
had the privilege to work with Nick Benton, Andrew Kennedy, and Jonas Jensen, writ-
ing x86 programs in Coq. In three months, I was (happily!) force-fed enough Coq,
SSReflect, and separation logic to implement and prove the correctness of a regular
expression compiler. Needless to say, my fellow interns were as impressive as the ones
I had left in Redmond. I owe special thanks to Aws Albarghouthi and Thomas Ströder
for making fun of my workaholism (they will be delighted to learn that my thesis is fi-
nally done), to Richard Eisenberg, Nicolas Frisby, Markus Rabe, and Gordon Stewart for
many interesting discussions, and to Sadia Ahmed for being such a perky neighbour. I
was also a great pleasure to reminisce with Akhilesh Singhania.

The present document has been written in Paris, from February to April 2013. Every
now and then, I would take a break from the redaction by visiting a nearby research
group. To work out, I would walk up to the LIX, in Saclay. I thank Assia Mahboubi for
her kind welcome and very useful advices. I also thank Stéphane Graham-Lengrand,
Bruno Barras, and Dale Miller for their time.

I would also regularly visit my neighbours at PPS. I would like to thank Pierre Boutil-
lier for exchanging his office in Paris for mine in Glasgow, and to Hugo Herbelin and
Pierre-Louis Curien to have permitted that trade. In a month, I have learned so much: I
am indebted to Matthieu Sozeau, and to my temporary office mates Guillaume Munch-
Maccagnoni, Pierre-Marie Pédrot, Matthias Puech, and Stéphane Zimmermann for this
glimpse at a fascinating world.

Finally, I am looking forward to start my post-doc in the Gallium team. I would like
to thank Didier Rémy and François Pottier for having supported my application, and
for their help in making it happen on a very short notice. I am also very grateful to
Thibaut Balabonski for his guidance and advice, and to Barbara Petit for having put us
in touch.
This dissertation is also the result of countless interactions at conferences or by email. For better or for worse, the initial impulse for Chapter 7 was provided by Peter Dybjer in Marseille, with the final kick provided by Derek Dreyer in Copenhagen. I thank them for these discussions. I also thank Andrea Vezzosi for these Agda files he would send me every now and then: he would prove a theorem I had postulated, or disprove it with a counter-example. I am indebted to Gabor Greif, for his feedback on my LICS paper, and to those madmen that have read parts of my thesis, (mostly) of their own free will: Clément, Fredrik Nordvall Forsberg, Guillaume, Hank, Lorenzo, Matteo Maffei, Stevan, and Vincent Siles.

My final thanks go to José Pedro Magalhães. Every draft paper I have written during my PhD has received his scrutiny. His insightful comments have greatly improved the final products. Without his encouragements, these papers would have remained in the state of drafts. Finally, I cannot thank him enough for the time and energy he has spent reading the first draft of this document. It has been a pleasure and, I daresay, an honour to receive the attention of such a selfless, humanistic scholar.

I will conclude with a few words for my family, in French. Je remercie mes parents pour leur soutien indéfectible durant toutes ces années. Je remercie ma mère pour m'avoir donné “Pierre” – la passion de bâtir – et mon père pour m'avoir donné “Évariste” – la passion des mathématiques. Je remercie aussi mes frères pour leur inspiration: merci à Pascal pour l'air frais des Calanques, merci à Yannick pour m'avoir ouvert la voie, et merci à Giovanni pour m'avoir constamment rappelé que “Qui ose gagne”. Mes derniers mots vont à Samantha dont les encouragements et la confiance m'ont portés tout au long de cette aventure. Merci.