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

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 $\mathit{Man} \to \mathit{Mortal}$; the minor premise asserts that being Greek *implies* being a man, denoted $\mathit{Greek} \to \mathit{Man}$. To prove the conclusion that all Greeks are mortal – $\mathit{Greek} \to \mathit{Mortal}$ – we construct a proof through logical *inferences*. For example, to prove $\mathit{Greek} \to \mathit{Mortal}$, it is sufficient to prove *Mortal*, provided an *abstract* inhabitant x of the type *Greek*, or formally:

$$\frac{x : \mathit{Greek} \vdash \mathit{Mortal}}{\vdash \mathit{Greek} \to \mathit{Mortal}}$$

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:

$$\frac{\overline{x : \mathit{Greek} \vdash \mathit{Man} \to \mathit{Mortal}} \qquad x : \mathit{Greek} \vdash \mathit{Man}}{x : \mathit{Greek} \vdash \mathit{Mortal}}$$

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:

$$\frac{\overline{x : \mathit{Greek} \vdash \mathit{Greek} \to \mathit{Man}} \qquad \overline{x : \mathit{Greek} \vdash \mathit{Greek}}}{x : \mathit{Greek} \vdash \mathit{Man}}$$

We have thus *formally* derived that $\mathit{Greek} \to \mathit{Mortal}$ from the assumptions that $\mathit{Man} \to \mathit{Mortal}$ and $\mathit{Greek} \to \mathit{Man}$.

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: Π-types and Σ-types. Read as a proposition, a Π-type $(x : A) \to B$ 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 zip function[1] has to cope with two logically meaningless cases:

$$
\begin{array}{llllll}
\mathsf{zip} & (xs : \mathsf{List}\ A) & (ys : \mathsf{List}\ B) & : & \mathsf{List}\ (A \times B) \\
\mathsf{zip} & \mathsf{nil} & \mathsf{nil} & \mapsto & \mathsf{nil} \\
\mathsf{zip} & (\mathsf{cons}\, a\ as) & (\mathsf{cons}\, b\ bs) & \mapsto & \mathsf{cons}\ (a, b)\ (\mathsf{zip}\, as\ bs) \\
\mathsf{zip} & \mathsf{nil} & (\mathsf{cons}\, b\ bs) & \{?\} & \\
\mathsf{zip} & (\mathsf{cons}\, a\ as) & \mathsf{nil} & \{?\} &
\end{array}
$$

Simple types do not let us express the invariant that the two input lists have the same

[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.

length. Using dependent types, we can express this invariant at the type level and write the function we intended to:

$$\begin{array}{lllll}
\mathsf{zip} & (n : \mathsf{Nat}) & (xs : \mathsf{Vec}\, A\, n) & (ys : \mathsf{Vec}\, B\, n) & : \ \mathsf{Vec}\, (A \times B)\, n \\
\mathsf{zip} & \quad 0 & \quad \mathsf{nil} & \quad \mathsf{nil} & \mapsto \mathsf{nil} \\
\mathsf{zip} & (\mathsf{suc}\, n) & (\mathsf{cons}\, a\, as) & (\mathsf{cons}\, b\, bs) & \mapsto \mathsf{cons}\, (a, b)\, (\mathsf{zip}\, as\, bs)
\end{array}$$

Intuitively, a vector $\mathsf{Vec}\, A\, n$ is a list of length n. We shall encounter several, equivalent definitions of vectors below.

(1.2) Read as a proposition, a Σ-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 *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 isEven predicate:

$$\begin{array}{l}
\mathsf{EvenNat} \triangleq (n : \mathsf{Nat}) \times \mathsf{isEven}\, n \quad \textbf{where} \\
\quad \begin{array}{lll}
\mathsf{isEven} & (n : \mathsf{Nat}) & : \ \textsc{Set} \\
\mathsf{isEven} & \quad 0 & \mapsto \mathbb{1} \\
\mathsf{isEven} & (\mathsf{suc}\, 0) & \mapsto \mathbb{0} \\
\mathsf{isEven} & (\mathsf{suc}\, (\mathsf{suc}\, n)) & \mapsto \mathsf{isEven}\, n
\end{array}
\end{array}$$

Vectors can also be defined through a Σ-type. They correspond to the lists whose length matches the index:

$$\mathsf{Vec}\, A\, n \triangleq (as : \mathsf{List}\, A) \times \mathsf{length}\, as = n$$

Operationally, Σ-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 Σ-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 $m + n$:

$$\begin{array}{llll}
(xs : \mathsf{Vec}\, A\, m) & +\!\!+ & (ys : \mathsf{Vec}\, A\, n) & : \ \mathsf{Vec}\, A\, (m + n) \\
\mathsf{nil} & +\!\!+ & \quad ys & \mapsto ys \\
(\mathsf{cons}\, a\, xs) & +\!\!+ & \quad ys & \mapsto \mathsf{cons}\, a\, (xs +\!\!+ ys)
\end{array}$$

(1.3) 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 mathematical 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 [Milner et al., 1997, Peyton Jones, 2003].

(1.4) Datatypes are a key convenience of modern functional programming languages. Together with polymorphism, datatypes are the key contribution of ML over LISP. Intuitively, 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

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Nat}\!:\!\textrm{S}\scriptstyle{\textrm{ET}}\ \textbf{where} \\
\quad \mathsf{Nat}\ \ni\ \mathsf{0} \\
\qquad\quad\ \mid\ \mathsf{suc}\,(n\!:\!\mathsf{Nat})
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{List}\,[A\!:\!\textrm{S}\scriptstyle{\textrm{ET}}]\!:\!\textrm{S}\scriptstyle{\textrm{ET}}\ \textbf{where} \\
\quad \mathsf{List}\,A\ \ni\ \mathsf{nil} \\
\qquad\qquad\ \mid\ \mathsf{cons}\,(a\!:\!A)(as\!:\!\mathsf{List}\,A)
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Tree}\,[A\!:\!\textrm{S}\scriptstyle{\textrm{ET}}]\!:\!\textrm{S}\scriptstyle{\textrm{ET}}\ \textbf{where} \\
\quad \mathsf{Tree}\,A\ \ni\ \mathsf{leaf} \\
\qquad\qquad\ \mid\ \mathsf{node}\,(lb\!:\!\mathsf{Tree}\,A)(a\!:\!A)(rb\!:\!\mathsf{Tree}\,A)
\end{array}
$$

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:

$$\begin{array}{l}
\textbf{data}\ \mathsf{Vec}\,[A\!:\!\textsc{Set}](n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathsf{Vec}\,A\ \ (n = 0) & \ni & \mathsf{nil} \\
\mathsf{Vec}\,A\ (n = \mathsf{suc}\,n') & \ni & \mathsf{cons}\,(n'\!:\!\mathsf{Nat})(a\!:\!A)(vs\!:\!\mathsf{Vec}\,A\,n')
\end{array}
\end{array}$$

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

```
data Vec (A : Set) : Nat → Set where
  nil  : Vec A zero
  cons : ∀ {n'} (x : A)(xs : Vec A n') → Vec A (suc n')
```

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

$$\begin{array}{l}
\textbf{data}\ \mathsf{AVL}\,(n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathsf{AVL}\ \ (n = 0) & \ni & \mathsf{leaf} \\
\mathsf{AVL}\ (n = \mathsf{suc}\,n') & \ni & \mathsf{nodeE}\,(n'\!:\!\mathsf{Nat})(lb\!:\!\mathsf{AVL}\,n')(rb\!:\!\mathsf{AVL}\,n') \\
 & \mid & \mathsf{nodeL}\,(n'\!:\!\mathsf{Nat})(lb\!:\!\mathsf{AVL}\,n')(rb\!:\!\mathsf{AVL}\,(\mathsf{suc}\,n')) \\
 & \mid & \mathsf{nodeR}\,(n'\!:\!\mathsf{Nat})(lb\!:\!\mathsf{AVL}\,(\mathsf{suc}\,n'))(rb\!:\!\mathsf{AVL}\,n')
\end{array}
\end{array}$$

An inhabitant of 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.

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:

$$\begin{array}{l}
\textbf{data}\ \mathsf{Fin}\,(n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathsf{Fin}\ (n = \mathsf{suc}\,n') & \ni & \mathsf{f0}\,(n'\!:\!\mathsf{Nat}) \\
 & \mid & \mathsf{fsuc}\,(n'\!:\!\mathsf{Nat})(k\!:\!\mathsf{Fin}\,n')
\end{array}
\end{array}$$

(1.6) 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 *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.

(1.7) 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 *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 *derivative* of a datatype [Abbott et al., 2005], which is a key component in the definition of the Zipper data-structure [Huet, 1997].

(1.8) Interestingly, this grammar of inductive definitions is itself nothing but an inductive type, which we can reify internally, *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 *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 *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.

(1.9) 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:

$$\begin{array}{lcl}
\textbf{data}\ \mathsf{Nat}\!:\!\textsc{Set}\ \textbf{where} & & \textbf{data}\ \mathsf{List}\ [A\!:\!\textsc{Set}]\!:\!\textsc{Set}\ \textbf{where} \\
\quad \mathsf{Nat}\ \ni \mathsf{0} & \Rightarrow & \quad \mathsf{List}\,A\ \ni \mathsf{nil} \\
\qquad\quad \mid\ \mathsf{suc}\,(n\!:\!\mathsf{Nat}) & & \qquad\qquad \mid\ \mathsf{cons}\,(a\!:\!A)(as\!:\!\mathsf{List}\,A)
\end{array}$$

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.

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

$$\begin{array}{lcl}
\textbf{data}\ \mathsf{Nat}\!:\!\textsc{Set}\ \textbf{where} & & \textbf{data}\ \mathsf{Fin}\ (n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where} \\
\quad \mathsf{Nat}\ \ni \mathsf{0} & \Rightarrow & \quad \mathsf{Fin}\ (n = \mathsf{suc}\,n')\ \ni \mathsf{f0}\ (n'\!:\!\mathsf{Nat}) \\
\qquad\quad \mid\ \mathsf{suc}\,(n\!:\!\mathsf{Nat}) & & \qquad\qquad\qquad\qquad \mid\ \mathsf{fsuc}\,(n'\!:\!\mathsf{Nat})(k\!:\!\mathsf{Fin}\,n')
\end{array}$$

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:

$$\begin{array}{lcl}
\textbf{data}\ \mathsf{Nat}\!:\!\textsc{Set}\ \textbf{where} & & \textbf{data}\ \mathsf{Vec}\ [A\!:\!\textsc{Set}](n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where} \\
\quad \mathsf{Nat}\ \ni \mathsf{0} & \Rightarrow & \quad \mathsf{Vec}\,A\quad (n = 0)\quad\ \ni \mathsf{nil} \\
\qquad\quad \mid\ \mathsf{suc}\,(n\!:\!\mathsf{Nat}) & & \quad \mathsf{Vec}\,A\ (n = \mathsf{suc}\,n')\ \ni \mathsf{cons}\,(n'\!:\!\mathsf{Nat})(a\!:\!A)(vs\!:\!\mathsf{Vec}\,A\,n')
\end{array}$$

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

(1.12) 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:

$$\begin{array}{lcll}
(m : \mathsf{Nat}) & < & (n : \mathsf{Nat}) & : \ \mathsf{Bool} \\
m & < & \mathsf{0} & \mapsto \mathsf{false} \\
\mathsf{0} & < & \mathsf{suc}\, n & \mapsto \mathsf{true} \\
\mathsf{suc}\, m & < & \mathsf{suc}\, n & \mapsto m < n
\end{array}$$

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

$$\begin{array}{llll}
\mathsf{lookup}\ (m : \mathsf{Nat}) & (xs : \mathsf{List}\ A) & : & \mathsf{Maybe}\ A \\
\mathsf{lookup}\quad m & \mathsf{nil} & \mapsto & \mathsf{nothing} \\
\mathsf{lookup}\quad \mathsf{0} & (\mathsf{cons}\, a\ xs) & \mapsto & \mathsf{just}\, a \\
\mathsf{lookup}\ (\mathsf{suc}\, n) & (\mathsf{cons}\, a\ xs) & \mapsto & \mathsf{lookup}\, n\ xs
\end{array}$$

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:

$$\begin{array}{lclcl c lcccl}
(m\!:\!\mathsf{Nat}) & < & (n\!:\!\mathsf{Nat}) & : & \mathsf{Bool} & & \mathsf{lookup} & (m\!:\!\mathsf{Nat}) & (xs\!:\!\mathsf{List}\,A) & : & \mathsf{Maybe}\,A \\
m & < & 0 & \{?\} & & & \mathsf{lookup} & m & \mathsf{nil} & \{?\} & \\
0 & < & \mathsf{suc}\,n & \{?\} & & \Rightarrow & \mathsf{lookup} & 0 & (\mathsf{cons}\,a\,xs) & \{?\} & \\
\mathsf{suc}\,m & < & \mathsf{suc}\,n & \{?\} & & & \mathsf{lookup} & (\mathsf{suc}\,n) & (\mathsf{cons}\,a\,xs) & \{?\} &
\end{array}$$

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

$$\begin{array}{lclcl c lcccl}
(m\!:\!\mathsf{Nat}) & < & (n\!:\!\mathsf{Nat}) & : & \mathsf{Bool} & & \mathsf{lookup} & (m\!:\!\mathsf{Nat}) & (xs\!:\!\mathsf{List}\,A) & : & \mathsf{Maybe}\,A \\
m & < & 0 & \mapsto & \mathsf{false} & & \mathsf{lookup} & m & \mathsf{nil} & \mapsto & \mathsf{nothing} \\
0 & < & \mathsf{suc}\,n & \mapsto & \mathsf{true} & \Rightarrow & \mathsf{lookup} & 0 & (\mathsf{cons}\,a\,xs) & \mapsto & \mathsf{just}\,a \\
\mathsf{suc}\,m & < & \mathsf{suc}\,n & \{?\} & & & \mathsf{lookup} & (\mathsf{suc}\,n) & (\mathsf{cons}\,a\,xs) & \{?\} &
\end{array}$$

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.

(1.13) 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

(1.14) 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 Π and Σ quantifiers. Compared to a strictly stratified logic such as System F^{ω}, 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 $\textsc{Set}_0$. $\textsc{Set}_0$ is itself a type former, of type $\textsc{Set}_1$. Again, in $\textsc{Set}_1$, we can quantify over objects in $\textsc{Set}_0$, 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.

(1.15) 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 Op – the set of operations – and an indexed set Ar – the set of arities of each constructor. The W-type is the least set closed under Op-nodes

$$\frac{\mathsf{Op} : \textsc{Set} \qquad \mathsf{Ar} : \mathsf{Op} \to \textsc{Set}}{\mathcal{W}\,\mathsf{Op}\,\mathsf{Ar} : \textsc{Set}}$$

whose inhabitants are introduced by choosing a kind of node *op* and providing its arguments with *xs*:

$$\frac{op : \mathsf{Op} \qquad xs : \mathsf{Ar}\,op \to \mathcal{W}\,\mathsf{Op}\,\mathsf{Ar}}{\mathsf{sup}\,op\,xs : \mathcal{W}\,\mathsf{Op}\,\mathsf{Ar}}$$

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

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

$$\mathsf{Nat} \triangleq \mathcal{W}\,\mathsf{Bool}\,\lambda \begin{cases} \mathsf{true} \mapsto \mathbb{0} \\ \mathsf{false} \mapsto \mathbb{1} \end{cases}$$

Consequently, the constructor of Nat relies on functions, respectively from the empty type and from the unit type:

$$\begin{aligned} \mathsf{0} &\triangleq \mathsf{sup}\,\mathsf{true}\,\mathbb{0}\text{-}\mathsf{elim} \\ \mathsf{suc}\,(n : \mathsf{Nat}) &\triangleq \mathsf{sup}\,\mathsf{false}\,\lambda *.\, n \end{aligned}$$

Extensionally, this is not an issue: we can collapse the trivial function spaces $X^{\mathbb{0}} \cong \mathbb{1}$ and $X^{\mathbb{1}} \cong X$. 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.

(1.17) 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 SET. 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.

(1.18) 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.

(1.19) 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 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, 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 syntax, 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:

KINDS OF FRAMES

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:

$$\begin{array}{lcl} \langle n \rangle & ::= & \mathbf{0} \\ & | & \mathbf{suc}\ \langle n \rangle \end{array}$$

where $\langle n \rangle$ denotes a non-terminal symbol "n", **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: $\mathbb{C}, \mathbb{D}, \ldots$
- Objects of a category $\mathbb{C}$: $|\mathbb{C}|$
- Hom-set of a category $\mathbb{C}$: $\mathbb{C}(A, B)$
- Category of morphisms: $\mathbb{C}^{\to}$
- Slice category: $\mathbb{C}/I$
- 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 $X, Y : I \to \textsc{Set}$ – or equivalently, morphisms of predicates – are defined pointwise by

$$X \dot{\to} Y \triangleq \forall i \!:\! I.\, X\, i \to Y\, i$$

where the index i is kept implicit.

1.5. 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] ;
4. Chapter 9 gives a slower paced presentation of the techniques presented in a second ICFP paper [Dagand and McBride, 2012].

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:

Syntax

$$
\begin{array}{lcllllllll}
\langle t\rangle, \langle T\rangle & ::= & x & & & & & & \\
 & \mid & \mathrm{SET}_{\ell} & & & & & & \\
 & \mid & \mathbb{0} & & & \mid & \mathbb{0}\text{-elim}\ \langle t\rangle & & \\
 & \mid & \mathbb{1} & \mid & * & & & & \\
 & \mid & (x\!:\!\langle T_1\rangle) \to \langle T_2\rangle & \mid & \lambda x\!:\!\langle T\rangle.\ \langle t\rangle & \mid & \langle t_1\rangle\ \langle t_2\rangle & & \\
 & \mid & (x\!:\!\langle T_1\rangle) \times \langle T_2\rangle & \mid & (x = \langle t_1\rangle, \langle t_2\rangle : \langle T_2\rangle) & \mid & \pi_0\ \langle t\rangle & \mid & \pi_1\ \langle t\rangle \\
\langle \Gamma\rangle & ::= & \epsilon & & & & & & \\
 & \mid & \langle \Gamma\rangle ; x\!:\!\langle T\rangle & & & & & &
\end{array}
$$

By convention, the non-terminal $\langle T \rangle$ ranges over types – *i.e.* $\langle T \rangle$ has type SET_ℓ – while $\langle t \rangle$ ranges over terms – *i.e.* $\langle t \rangle$ has type $\langle T \rangle$, a type. Because our language is dependently-typed, terms and types belong to the same syntactic category: the non-terminals $\langle t \rangle$ and $\langle T \rangle$ 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 $(x : A) \to B$ and Σ-types are denoted $(x : A) \times B$, 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 $\mathbb{1}$.

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.* SET_ℓ ;
- Canonical objects, *i.e.* inhabitants of SET, are written in blue as well, *e.g.* ⊤ ;
- Value constructors are written in red and sans serif, *e.g.* cons ;
- Computations, and therefore eliminators, are written in green and sans serif, *e.g.* 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 Π-types and Σ-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 Π-types and Σ-types, we expect the arrow to bind

META-THEORY

$$\boxed{\langle\Gamma\rangle \vdash \text{VALID}}$$

$$\frac{}{\vdash \text{VALID}}\ \text{CTXT-EMPTY} \qquad \frac{\begin{array}{ll}\Gamma \vdash \text{VALID} & \\ x \notin \Gamma & \Gamma \vdash S : \text{SET}_\ell\end{array}}{\Gamma; x : S \vdash \text{VALID}}\ \text{CTXT-VAR}$$

$$\frac{\begin{array}{ll}\Gamma \vdash \text{VALID} & \Gamma \vdash S : \text{SET}_\ell \\ x \notin \Gamma & \Gamma \vdash s : S\end{array}}{\Gamma; x \mapsto s : S \vdash \text{VALID}}\ \text{CTXT-DEF}$$

Figure 2.1.: Context validity

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 α-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:

$\langle\Gamma\rangle \vdash \text{VALID}$	Γ is a valid context, giving types to variables and definitions
$\langle\Gamma\rangle \vdash \langle t\rangle : \langle T\rangle$	the term t has type T in context Γ
$\langle\Gamma\rangle \vdash \langle s\rangle \equiv \langle t\rangle : \langle T\rangle$	the terms s and t are equal at type T in context Γ

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 Γ with a variable x of type S ; or
- the extension of a context Γ with a definition x that stands for a term s of type S

META-THEORY

$$\boxed{\langle\Gamma\rangle \vdash \langle t\rangle : \langle T\rangle}$$

$$\frac{\Gamma; x : S; \Delta \vdash \text{VALID}}{\Gamma; x : S; \Delta \vdash x : S}\,\textsc{Var} \qquad \frac{\Gamma; x \mapsto t : S; \Delta \vdash \text{VALID}}{\Gamma; x \mapsto t : S; \Delta \vdash x : S}\,\textsc{Def}$$

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \textsc{Set}_\ell : \textsc{Set}_{\ell+1}}\,\textsc{Sort} \qquad \frac{\Gamma \vdash S : \textsc{Set}_\ell}{\Gamma \vdash S : \textsc{Set}_{\ell+1}}\,\textsc{Cuml}$$

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \mathbb{0} : \textsc{Set}_\ell}\,\textsc{Bottom} \qquad \frac{\Gamma \vdash b : \mathbb{0} \qquad \Gamma \vdash A : \textsc{Set}_\ell}{\Gamma \vdash \mathbb{0}\text{-elim}\, b : A}\,\textsc{Absurd}$$

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash \mathbb{1} : \textsc{Set}_\ell}\,\textsc{Unit} \qquad \frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash * : \mathbb{1}}\,\textsc{Void}$$

$$\frac{\Gamma \vdash S : \textsc{Set}_\ell \quad \Gamma; x : S \vdash T : \textsc{Set}_\ell}{\Gamma \vdash (x : S) \to T : \textsc{Set}_\ell}\,\textsc{Pi}$$

$$\frac{\begin{array}{c}\Gamma \vdash S : \textsc{Set}_\ell \\ \Gamma; x : S \vdash t : T\end{array}}{\Gamma \vdash \lambda x : S.\, t : (x : S) \to T}\,\textsc{Lam} \qquad \frac{\begin{array}{l}\Gamma \vdash f : (x : S) \to T \\ \Gamma \vdash s : S\end{array}}{\Gamma \vdash f\, s : T[s/x]}\,\textsc{App}$$

$$\frac{\Gamma \vdash S : \textsc{Set}_\ell \quad \Gamma; x : S \vdash T : \textsc{Set}_\ell}{\Gamma \vdash (x : S) \times T : \textsc{Set}_\ell}\,\textsc{Sigma}$$

$$\frac{\Gamma \vdash s : S \quad \Gamma; x : S \vdash T : \textsc{Set}_\ell \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash (x = s, t : T) : (x : S) \times T}\,\textsc{Pair}$$

$$\frac{\Gamma \vdash p : (x : S) \times T}{\Gamma \vdash \pi_0\, p : S}\,\textsc{Fst} \qquad \frac{\Gamma \vdash p : (x : S) \times T}{\Gamma \vdash \pi_1\, p : T[\pi_0\, p/x]}\,\textsc{Snd}$$

$$\frac{\Gamma \vdash s : S \quad \Gamma \vdash S \equiv T : \textsc{Set}_\ell}{\Gamma \vdash s : T}\,\textsc{Conv}$$

Figure 2.2.: Typing judgments

Meta-theory

$$\boxed{\langle\Gamma\rangle \vdash \langle s\rangle \equiv \langle t\rangle : \langle T\rangle}$$

$$\frac{\begin{array}{l}\Gamma \vdash S : \textsc{Set}_\ell \\ \Gamma \vdash s : S \qquad\qquad \Gamma; x : S \vdash t : T\end{array}}{\Gamma \vdash (\lambda x : S.\, t)\, s \equiv t[s/x] : T[s/x]}\ \textsc{Beta}$$

$$\frac{\begin{array}{c}\Gamma; x : S \vdash T : \textsc{Set}_\ell \\ \Gamma \vdash s : S \qquad \Gamma \vdash t : T[s/x]\end{array}}{\Gamma \vdash \pi_0\, (x = s, t : T) \equiv s : S}\ \textsc{Beta-Fst}$$

$$\frac{\begin{array}{c}\Gamma; x : S \vdash T : \textsc{Set}_\ell \\ \Gamma \vdash s : S \qquad \Gamma \vdash t : T[s/x]\end{array}}{\Gamma \vdash \pi_1\, (x = s, t : T) \equiv t : T[s/x]}\ \textsc{Beta-Snd}$$

$$\frac{\Gamma; x \mapsto t : S; \Delta \vdash \textsc{valid}}{\Gamma \vdash x \equiv t : S}\ \textsc{Delta}$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash x \equiv x : T}\ \textsc{Refl} \qquad \frac{\Gamma \vdash x \equiv y : T}{\Gamma \vdash y \equiv x : T}\ \textsc{Sym} \qquad \frac{\begin{array}{l}\Gamma \vdash x \equiv y : T \\ \Gamma \vdash y \equiv z : T\end{array}}{\Gamma \vdash x \equiv z : T}\ \textsc{Trans}$$

Figure 2.3.: Judgmental equality

2.9 Remark (Use of definitions)**.** From a proof-theoretic standpoint, the rule CTXT-DEF, 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:

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

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 $\textsc{Set}_\ell$, each of which is closed under the empty type, the unit type, Σ-types, and Π-types.

2.11 Remark (Cumulativity)**.** Cumulativity of universes (rule CUML) lets us shift types to higher universes [Luo, 1994]. This rule captures the inclusion of universes $\textsc{Set}_0 \subset \textsc{Set}_1 \subset \ldots$ For example, it enables us to write the type-polymorphic identity function:

$$\begin{array}{llll} \mathsf{id}\,(X : \textsc{Set}) & (x : X) & : & X \\ \mathsf{id}\quad X & \quad x & \mapsto & x \end{array}$$

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 λ-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

(2.13) Unlike traditional presentations [Luo, 1994] based on Pure Type Systems (PTS), we do not define our notion of *definitional* 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:

$$\frac{\Gamma \vdash s_1 \equiv s_2 : S \qquad \Gamma \vdash t_1 \equiv t_2 : T[x/s_1]}{\Gamma \vdash (x = s_1, t_1 : S) \equiv (x = s_2, t_2 : S) : (x : S) \times T} \qquad \frac{\Gamma \vdash f_1 \equiv f_2 : (x : S) \to T \qquad \Gamma \vdash s_1 \equiv s_2 : S}{\Gamma \vdash f_1\, s_1 \equiv f_2\, s_2 : T[x/s_1]}$$

(2.15) By specifying equality as a judgment, we leave open the problem of implementing equality, requiring only a congruence including ordinary computation (β-rules). Such an equality can be decided, for example, by testing β-normal forms up to α-equivalence [Adams, 2006]. Richer equalities, involving various degrees of η-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]:

$$\begin{array}{c} \Gamma \vdash u : A \to C \quad \Gamma \vdash v : B \to C \\ \hline\hline \Gamma \vdash \langle u, v \rangle : A + B \to C \end{array}$$

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

$$\begin{array}{l} \Gamma \vdash p : (s : S) \times T \\ \Gamma \vdash f : (s : S)(t : T) \to P\ (s, t) \\ \hline\hline \Gamma \vdash \textsf{sigma-split}\ f\ p : P\ p \end{array}$$

This corresponds to a positive presentation of Σ-types. Similarly, one could treat Π-types as positive types and eliminate them through the 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, *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:

$$\begin{array}{c} \Gamma ; x : A \vdash b : B \\ \hline\hline \Gamma \vdash \lambda x .\, b : A \to B \end{array}$$

In our type theory, we adopt this traditional presentation of functions for Π-types, eliminating a function by *observing* its behavior on an argument. Similarly, we treat Σ-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: with relative ease, the type theory can be extended to support the so-called η-laws for

[1] The use of a double line, reminiscent of the categorical notation for adjunctions, to signify the invertibility of a connective is not a coincidence.

our Π-type, Σ-type, and unit [Abel et al., 2008, 2009]:

$$\frac{\Gamma \vdash S : \textsc{Set}_\ell \qquad \Gamma \vdash f : (x : S) \to T}{\Gamma \vdash f \equiv \lambda x : S.\, f\, x : (x : S) \to T}\ \textsc{Eta-Lam}$$

$$\frac{\Gamma \vdash S : \textsc{Set}_\ell \qquad \Gamma \vdash p : (x : S) \times T}{\Gamma \vdash p \equiv (x = \pi_0\, p, \pi_1\, p : S) : (x : S) \times T}\ \textsc{Eta-Pair}$$

$$\frac{\Gamma \vdash t : \mathbb{1}}{\Gamma \vdash t \equiv * : \mathbb{1}}\ \textsc{Eta-Void}$$

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 η-laws for Σ-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

$$\begin{array}{lll} \mathsf{swap}\ (p : A \times B) & : & B \times A \\ \mathsf{swap}\quad p & \mapsto & (\pi_1\, p, \pi_0\, p) \end{array}$$

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

$$\Gamma \vdash \mathsf{swap} \equiv \mathsf{id} : \mathbb{1} \times \mathbb{1} \to \mathbb{1} \times \mathbb{1}$$

2.1.4. Propositional equality

(2.18) Definitional equality captures the equalities that hold *by definition*, *i.e.* by unfolding of definitions. Its concrete nature leaves no place for reasoning: it cannot be manipulated within the logic. This design choice has the benefit of decidability: we can always decide whether two objects are definitionally equal or not. However, more objects are equal than a computer can decide. Whilst we cannot decide equality, we can define a language of equality *certificates* that the computer can *check*: we work around the undecidability of equality by defining a checkable language for human-generated evidence. 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

META-THEORY

$$\frac{\begin{array}{c}\Gamma \vdash A : \textsc{Set}_\ell \\ \Gamma \vdash x : A \qquad \Gamma \vdash y : A\end{array}}{\Gamma \vdash x = y : \textsc{Set}_\ell}\ \textsc{Prop-Equal}$$

$$\frac{\Gamma \vdash A : \textsc{Set}_\ell \qquad \Gamma \vdash x : A}{\Gamma \vdash \mathsf{refl} : x = x}\ \textsc{Refl}$$

$$\frac{\begin{array}{c}\Gamma \vdash P : A \to \textsc{Set}_\ell \\ \Gamma \vdash x : A \qquad \Gamma \vdash y : A\end{array}}{\Gamma \vdash \mathsf{subst} : x = y \to P\,x \to P\,y}\ \textsc{Subst}$$

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \mathsf{absorb} : \mathsf{subst}\ \mathsf{refl}\ x = x}\ \textsc{Abs}$$

Figure 2.4.: Axiomatic propositional equality

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 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 $A : \textsc{Set}_\ell$ 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 $\textsc{Set}_\ell$. 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 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:

$$\frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash \mathsf{refl} : a = b}$$

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

$$b : \mathbb{0} \vdash \mathbb{1} = \mathbb{0} : \textsc{Set}$$

We might be reluctant to have the corresponding equation to hold definitionally. Extensional 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:

$$\mathsf{UIP} : (\alpha : a = b)(\beta : a = b) \to \alpha = \beta$$

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 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 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 rewritings 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 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*

```
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:

```
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 $\{\mathsf{'a}\,\mathsf{'b}\,\mathsf{'c}\}$ of tags $\mathsf{'a}$, $\mathsf{'b}$, and $\mathsf{'c}$. Then, we want to be able to index into such collection by name, for example pointing to the element $\mathsf{'b}$ – 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.*

$$\{\mathsf{'a} \mapsto e_a \,;\, \mathsf{'b} \mapsto e_b \,;\, \mathsf{'c} \mapsto e_c\}$$

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 *presentational* 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 *presentational* information in their types and therefore support readability through elaboration.

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

$$\mathsf{EnumU} : \textsc{Set}_\ell \qquad \mathsf{EnumT}\ (E : \mathsf{EnumU}) : \textsc{Set}_\ell$$

Inhabitants of EnumU are finite collections of tags, the so-called enumerations. For an enumeration $E : \mathsf{EnumU}$, $\mathsf{EnumT}\ E$ 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.

Syntax

$$\langle t \rangle, \langle T \rangle ::= \ \ldots \quad | \quad \mathsf{UId} \quad | \quad \mathsf{'ident}$$

Meta-theory

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathsf{UId} : \textsc{Set}_\ell} \qquad \frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathsf{'}s : \mathsf{UId}}\ s \text{ a valid identifier}$$

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.

(2.33) Tag equality is nominal: two tags are equal if and only if their identifiers are the same. We extend judgmental equality accordingly:

Meta-theory

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathsf{'}s \equiv \mathsf{'}s : \mathsf{UId}}$$

2.34 Example. Hence, for instance, we have that $\nvdash \mathsf{'true} \equiv \mathsf{'false} : \mathsf{UId}$.

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:

Syntax

$$\langle t \rangle, \langle T \rangle ::= \ \ldots \quad \mid \quad \mathsf{EnumU} \quad \mid \quad \mathsf{nilE} \quad \mid \quad \mathsf{consE}\ \langle t_1 \rangle\ \langle t_2 \rangle$$

Specification

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathsf{EnumU} : \textsc{Set}_\ell}$$

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathsf{nilE} : \mathsf{EnumU}} \qquad \frac{\Gamma \vdash t : \mathsf{UId} \quad \Gamma \vdash E : \mathsf{EnumU}}{\Gamma \vdash \mathsf{consE}\, t\, E : \mathsf{EnumU}}$$

Intuitively, an 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 EnumU and the introduction rule nilE and 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, EnumU can be introduced as an inductive type.

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

$$\{\mathsf{'true}\ \mathsf{'false}\} \triangleq \mathsf{consE}\ \mathsf{'true}\ (\mathsf{consE}\ \mathsf{'false}\ \mathsf{nilE})$$

(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 names appear in the same order:

SPECIFICATION

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathsf{nilE} \equiv \mathsf{nilE} : \mathsf{EnumU}} \qquad \frac{\Gamma \vdash t_1 \equiv t_1 : \mathsf{UId} \quad \Gamma \vdash E_1 \equiv E_2 : \mathsf{EnumU}}{\Gamma \vdash \mathsf{consE}\, t_1\, E_1 \equiv \mathsf{consE}\, t_2\, E_2 : \mathsf{EnumU}}$$

2.39 Example. The following collections are therefore *not* equal:

$$\begin{aligned} &\nvdash \{\text{'a 'b}\} \equiv \{\text{'a 'b 'c}\} : \mathsf{EnumU} \\ &\nvdash \{\text{'a 'b}\} \equiv \{\text{'c 'd}\} : \mathsf{EnumU} \\ &\nvdash \{\text{'a 'b}\} \equiv \{\text{'b 'a}\} : \mathsf{EnumU} \end{aligned}$$

2.2.3. Indexing into enumerations

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

SYNTAX

$$\langle t \rangle, \langle T \rangle ::= \ \ldots \quad | \quad \mathsf{EnumT}\ \langle t \rangle \quad | \quad 0 \quad | \quad 1\!+\ \langle t \rangle$$

META-THEORY

$$\frac{\Gamma \vdash E : \mathsf{EnumU}}{\Gamma \vdash \mathsf{EnumT}\ E : \textsc{Set}_\ell}$$

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash 0 : \mathsf{EnumT}\ (\mathsf{consE}\ t\ E)} \qquad \frac{\Gamma \vdash n : \mathsf{EnumT}\ E}{\Gamma \vdash 1\!+\ n : \mathsf{EnumT}\ (\mathsf{consE}\ t\ E)}$$

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

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

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

We check that for any $k : \mathsf{EnumT}\ E$, $\nvdash 1\!+ (1\!+ k) : \mathsf{Bool}$. That is, Bool has indeed only two inhabitants.

(2.43) Equality of indices is the expected congruence:

META-THEORY

$$\frac{\Gamma \vdash \text{VALID}}{\Gamma \vdash 0 \equiv 0 : \mathsf{consE}\ t\ E} \qquad \frac{\Gamma \vdash n_1 \equiv n_2 : E}{\Gamma \vdash 1\!+ n_1 \equiv 1\!+ n_2 : \mathsf{consE}\ t\ E}$$

2.44 Example. We thus (reassuringly) cannot identify true and false:

$$\nvdash \mathsf{true} \triangleq 0 \equiv 1\!+ 0 \triangleq \mathsf{false} : \mathsf{EnumT}\ \{\text{'}\mathsf{true}\ \text{'}\mathsf{false}\}$$

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, false when referring to an inhabitant of the type 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:

$$\{\text{'}\mathsf{this}\ \text{'}\mathsf{this}\} \triangleq \mathsf{consE}\ \text{'}\mathsf{this}\ (\mathsf{consE}\ \text{'}\mathsf{this}\ \mathsf{nilE})$$

It is in fact isomorphic (but not equal, equality being nominal) to the enumeration defining Bool. Indeed, it is inhabited by two terms: 0 that indexes the first '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 $(e : \mathsf{EnumT}\ \{'a_1 \ldots 'a_n\}) \to T$ is an interesting object: it is finite, isomorphic to the (right-nested) tuple $T\ a_1 \times \ldots \times T\ a_n \times \mathbb{1}$. Being first-order, we shall 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 $(e : \mathsf{EnumT}\ E) \to P\ e$:

META-THEORY

$$
\begin{array}{llll}
\pi\ (E : \mathsf{EnumU}) & (P : \mathsf{EnumT}\ E \to \textsc{Set}_\ell) & : & \textsc{Set}_\ell \\
\pi\quad \mathsf{nilE} & P & \mapsto & \mathbb{1} \\
\pi\ \ (\mathsf{consE}\ t\ E) & P & \mapsto & P\ 0 \times \pi\ E\ \lambda e.\ P\ (1{+}\ e)
\end{array}
$$

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 $P : \mathsf{Bool} \to \textsc{Set}$. The finite function space from $b : \mathsf{Bool}$ to $P\ b$ unfolds to:

$$\pi\ \{'\mathsf{true}\ '\mathsf{false}\}\ P \leadsto P\ \mathsf{true} \times P\ \mathsf{false} \times \mathbb{1}$$

That is, it is a pair of an inhabitant of $P\ \mathsf{true}$ *and* an inhabitant of $P\ \mathsf{false}$.

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 $(e : \mathsf{EnumT}\ E) \to P\ e$. In effect, switch simply peels off the lookup table based on the index e:

META-THEORY

$$\begin{array}{l}
\mathsf{switch}\ (ps\!:\!\pi\ E\ P)\ (e\!:\!\mathsf{EnumT}\ E)\ :\ P\ e \\
\mathsf{switch}\ b \quad 0 \quad\ \mapsto \pi_0\ b \\
\mathsf{switch}\ b\ (1{+}\ e)\ \mapsto \mathsf{switch}\ (\pi_1\ b)\ e
\end{array}$$

2.2.5. Sum type

(2.53) Using a finite set of cardinality n in the domain of a Σ-type, we obtain an encoding of n-ary sums. In the binary case, using Bool in the domain, we obtain an encoding of the usual intuitionistic disjunction:

$$A + B \triangleq (b\!:\!\mathsf{Bool}) \times \left\{\begin{array}{ll} \text{'true} & \mapsto A \\ \text{'false} & \mapsto B \end{array}\right\} b$$

We can then derive the usual introduction and elimination forms of disjunction by falling back to the introduction and elimination forms of Σ-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:

SPECIFICATION

$$\frac{\Gamma \vdash A\!:\!\mathrm{SET}_\ell \quad \Gamma \vdash B\!:\!\mathrm{SET}_\ell}{\Gamma \vdash A + B\!:\!\mathrm{SET}_\ell}\ \textsc{Sum}$$

$$\begin{array}{l}
\langle (u\!:\!A \to C), (v\!:\!B \to C)\rangle\ (s\!:\!A + B)\ :\ C \\
\langle u, v\rangle\ (\mathsf{inj}_l\ a)\ \mapsto u\ a \\
\langle u, v\rangle\ (\mathsf{inj}_r\ b)\ \mapsto v\ b
\end{array}$$

$$\frac{\Gamma \vdash a\!:\!A}{\Gamma \vdash \mathsf{inj}_l\ a\!:\!A + B}\ \textsc{Inj-Left}$$

$$\frac{\Gamma \vdash b\!:\!B}{\Gamma \vdash \mathsf{inj}_r\ b\!:\!A + B}\ \textsc{Inj-Right}$$

2.3. Meta-Theoretical Properties

(2.54) 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 η-laws, or Σ-types. Nonetheless, by making a normalisation assumption, Werner [1994] has shown how to regain these η-laws with an untyped conversion. Conversely, Goguen [1994] has shown that a typed conversion can be faithfully modelled by an underlying, untyped conversion.

(2.55) 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 hypothesis in the context, making terms available to larger contexts:

1. If $\begin{cases} \Gamma; \Delta \vdash \textsc{valid} \\ \Gamma \vdash A : \textsc{Set}_\ell \\ x \notin \Gamma; \Delta \end{cases}$, then $\Gamma; x : A; \Delta \vdash \textsc{valid}$

2. If $\begin{cases} \Gamma; \Delta \vdash t : T \\ \Gamma \vdash A : \textsc{Set}_\ell \\ x \notin \Gamma; \Delta \end{cases}$, then $\Gamma; x : A; \Delta \vdash t : T$

3. If $\begin{cases} \Gamma;\Delta \vdash s \equiv t : T \\ \Gamma \vdash A : \mathrm{SET}_\ell \\ x \notin \Gamma;\Delta \end{cases}$, then $\Gamma; x : A; \Delta \vdash s \equiv t : T$

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:

1. If $\begin{cases} \Gamma; x : A; \Delta \vdash \text{VALID} \\ \Gamma \vdash a : A \end{cases}$, then $\Gamma; \Delta[a/x] \vdash \text{VALID}$
2. If $\begin{cases} \Gamma; x : A; \Delta \vdash t : T \\ \Gamma \vdash a : A \end{cases}$, then $\Gamma; \Delta[a/x] \vdash t[a/x] : T[a/x]$
3. If $\begin{cases} \Gamma; x : A; \Delta \vdash s \equiv t : T \\ \Gamma \vdash a : A \end{cases}$, then $\Gamma; \Delta[a/x] \vdash s[a/x] \equiv t[a/x] : T[a/x]$

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 contexts are convertible, and, inductively, if two types are convertible in convertible contexts, they form a convertible context:

$$\frac{}{\epsilon \equiv \epsilon \vdash \text{VALID}} \qquad \frac{\Gamma \equiv \Delta \vdash \text{VALID} \qquad \Gamma \vdash S \equiv T : \mathrm{SET}_\ell}{\Gamma; x : S \equiv \Delta; x : T \vdash \text{VALID}}$$

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

1. If $\begin{cases} \Gamma \vdash t : T \\ \Gamma \equiv \Delta \vdash \text{VALID} \\ \Delta \vdash \text{VALID} \end{cases}$, then $\Delta \vdash t : T$
2. If $\begin{cases} \Gamma \vdash s \equiv t : T \\ \Gamma \equiv \Delta \vdash \text{VALID} \\ \Delta \vdash \text{VALID} \end{cases}$, then $\Delta \vdash s \equiv t : T$

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 Γ and Δ – to match up the type in the new context Δ.

□

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 Γ, we can deduce that its context itself is well-formed: if $\Gamma \vdash t : T$, then $\Gamma \vdash \textsc{valid}$.

Proof. By induction of the typing judgment.

□

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 $\textsc{Set}_\ell$, or $\Gamma \vdash T : \textsc{Set}_\ell$
2. If $\Gamma \vdash s \equiv t : T$, then either T is a sort $\textsc{Set}_\ell$, or $\Gamma \vdash T : \textsc{Set}_\ell$
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.

□

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 Γ, 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 Γ, 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.

□

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 Σ-types, and functions using Π-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):

$$\frac{\Gamma; x\!:\!S; \Delta \vdash \text{VALID}}{\Gamma; x\!:\!S; \Delta \vdash x\!:\!S} \quad \Rightarrow \quad \frac{\Gamma; x\!:\!S; \Delta \vdash \text{VALID}}{\Gamma; x\!:\!S; \Delta \vdash x \overset{Syn}{\leadsto} x \in S}$$

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):

$$\frac{\Gamma \vdash S\!:\!\mathrm{SET} \quad \Gamma; x\!:\!S \vdash t\!:\!T}{\Gamma \vdash \lambda x\!:\!S.\, t\!:\!(x\!:\!S) \to T} \quad \Rightarrow \quad \frac{\Gamma; x\!:\!S \vdash T \ni t \overset{Chk}{\leadsto} t'}{\Gamma \vdash (x\!:\!S) \to T \ni \lambda x.\, t \overset{Chk}{\leadsto} \lambda x\!:\!S.\, t'}$$

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 $\langle\Gamma\rangle \vdash \langle T\rangle \ni \langle\mathsf{exprCheck}\rangle \overset{Chk}{\leadsto} \langle t\rangle$ specifies *type checking* ;
- The judgment $\langle\Gamma\rangle \vdash \langle\mathsf{exprSynth}\rangle \overset{Syn}{\leadsto} \langle t\rangle \in \langle T\rangle$ specifies *type synthesis*.

During *type synthesis*, we actually deduce a type T from an expression 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

ELABORATION

$$
\begin{array}{rcll}
\langle \mathsf{exprSynth} \rangle & ::= & x & \textit{(variable)} \\
& \mid & (\langle \mathsf{exprCheck}_1 \rangle : \langle \mathsf{exprCheck}_2 \rangle) & \textit{(type annotation)} \\
& \mid & \langle \mathsf{exprSynth} \rangle \; \langle \mathsf{exprCheck} \rangle & \textit{(application)} \\
& \mid & \pi_0 \, \langle \mathsf{exprCheck} \rangle \quad \mid \quad \pi_1 \, \langle \mathsf{exprCheck} \rangle & \textit{(projections)} \\
\\
\langle \mathsf{exprCheck} \rangle & ::= & \langle \mathsf{exprSynth} \rangle & \textit{(synthesis mode)} \\
& \mid & \lambda x.\, \langle \mathsf{exprCheck} \rangle & \textit{(abstraction)} \\
& \mid & (\langle \mathsf{exprCheck}_1 \rangle, \langle \mathsf{exprCheck}_2 \rangle) & \textit{(pair)} \\
& \mid & \langle t \rangle & \textit{(term)} \\
& \mid & [\langle \mathsf{exprCheck}_0 \rangle \ldots \langle \mathsf{exprCheck}_n \rangle] & \textit{(tuple, ¶ 3.12)} \\
& \mid & \{\langle \mathsf{exprCheck}_0 \rangle \ldots \langle \mathsf{exprCheck}_n \rangle\} & \textit{(enumeration, ¶ 3.16)} \\
& \mid & \left\{ \begin{array}{c} \langle \mathsf{exprCheck}^I_0 \rangle \mapsto \langle \mathsf{exprCheck}^O_0 \rangle \\ \ldots \\ \langle \mathsf{exprCheck}^I_n \rangle \mapsto \langle \mathsf{exprCheck}^O_n \rangle \end{array} \right\} & \textit{(enum. elimination, ¶ 3.18)}
\end{array}
$$

Figure 3.1.: Expression language

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 (π and switch). We expose these operators as (defined) variables. Doing so, we avoid cluttering our definition of the 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

Elaboration

$$\boxed{\langle\Gamma\rangle \vdash \langle\mathsf{exprSynth}\rangle \overset{\mathit{Syn}}{\rightsquigarrow} \langle t\rangle \in \langle T\rangle}$$

$$\frac{\Gamma; x\!:\!S; \Delta \vdash \textsc{valid}}{\Gamma; x\!:\!S; \Delta \vdash x \overset{\mathit{Syn}}{\rightsquigarrow} x \in S}$$

$$\frac{\Gamma \vdash \mathrm{S{\scriptsize ET}}_\ell \ni T \overset{\mathit{Chk}}{\rightsquigarrow} T' \quad \Gamma \vdash T' \ni t \overset{\mathit{Chk}}{\rightsquigarrow} t'}{\Gamma \vdash (t\!:\!T) \overset{\mathit{Syn}}{\rightsquigarrow} t' \in T'}$$

$$\frac{\begin{array}{c}\Gamma \vdash f \overset{\mathit{Syn}}{\rightsquigarrow} f' \in (x\!:\!S) \to T \\ \Gamma \vdash S \ni s \overset{\mathit{Chk}}{\rightsquigarrow} s'\end{array}}{\Gamma \vdash f\, s \overset{\mathit{Syn}}{\rightsquigarrow} f'\, s' \in T\{s'/x\}}$$

$$\frac{\Gamma \vdash p \overset{\mathit{Syn}}{\rightsquigarrow} p' \in (x\!:\!S) \times T}{\Gamma \vdash \pi_0 p \overset{\mathit{Syn}}{\rightsquigarrow} \pi_0 p' \in S}$$

$$\frac{\Gamma \vdash p \overset{\mathit{Syn}}{\rightsquigarrow} p' \in (x\!:\!S) \times T}{\Gamma \vdash \pi_1 p \overset{\mathit{Syn}}{\rightsquigarrow} \pi_1 p' \in T\{\pi_0 p'/x\}}$$

(a) Type synthesis

$$\boxed{\langle\Gamma\rangle \vdash \langle T\rangle \ni \langle\mathsf{exprCheck}\rangle \overset{\mathit{Chk}}{\rightsquigarrow} \langle t\rangle}$$

$$\frac{\Gamma \vdash s \overset{\mathit{Syn}}{\rightsquigarrow} s' \in S \quad \Gamma \vdash S \equiv T\!:\!\mathrm{S{\scriptsize ET}}_\ell}{\Gamma \vdash T \ni s \overset{\mathit{Chk}}{\rightsquigarrow} s'}$$

$$\frac{\Gamma; x\!:\!S \vdash T \ni t \overset{\mathit{Chk}}{\rightsquigarrow} t'}{\Gamma \vdash (x\!:\!S) \to T \ni \lambda x.\, t \overset{\mathit{Chk}}{\rightsquigarrow} \lambda x\!:\!S.\, t'}$$

$$\frac{\Gamma \vdash S \ni s \overset{\mathit{Chk}}{\rightsquigarrow} s' \quad \Gamma \vdash T\{s'/x\} \ni t \overset{\mathit{Chk}}{\rightsquigarrow} t'}{\Gamma \vdash (x\!:\!S) \times T \ni (s, t) \overset{\mathit{Chk}}{\rightsquigarrow} (x = s', t'\!:\!T)}$$

(b) Type checking

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathrm{S{\scriptsize ET}}_{\ell+1} \ni \mathrm{S{\scriptsize ET}}_\ell \overset{\mathit{Chk}}{\rightsquigarrow} \mathrm{S{\scriptsize ET}}_\ell}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{S{\scriptsize ET}}_\ell \ni S \overset{\mathit{Chk}}{\rightsquigarrow} S' \\ \Gamma; x\!:\!S' \vdash \mathrm{S{\scriptsize ET}}_\ell \ni T \overset{\mathit{Chk}}{\rightsquigarrow} T'\end{array}}{\Gamma \vdash \mathrm{S{\scriptsize ET}}_\ell \ni (x\!:\!S) \to T \overset{\mathit{Chk}}{\rightsquigarrow} (x\!:\!S') \to T'}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathrm{S{\scriptsize ET}}_\ell \ni S \overset{\mathit{Chk}}{\rightsquigarrow} S' \\ \Gamma; x\!:\!S' \vdash \mathrm{S{\scriptsize ET}}_\ell \ni T \overset{\mathit{Chk}}{\rightsquigarrow} T'\end{array}}{\Gamma \vdash \mathrm{S{\scriptsize ET}}_\ell \ni (x\!:\!S) \times T \overset{\mathit{Chk}}{\rightsquigarrow} (x\!:\!S') \times T'}$$

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathrm{S{\scriptsize ET}}_\ell \ni \mathbb{1} \overset{\mathit{Chk}}{\rightsquigarrow} \mathbb{1}} \qquad \frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathbb{1} \ni * \overset{\mathit{Chk}}{\rightsquigarrow} *}$$

$$\vdots$$

(c) Type checking (structural rules)

Figure 3.2.: Bidirectional type checker

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 normalising substitution, denoted $\{-\}$. 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) **Elaboration of tuples.** By design of our universe of enumerations, we are often going to build inhabitants of Σ-telescope of the form $(a : A) \times (b : B) \times \ldots \times (z : Z) \times \mathbb{1}$. To reduce the syntactic burden of these nested pairs, we elaborate a tuple notation inspired by LISP:

ELABORATION

$$\frac{}{\Gamma \vdash \mathbb{1} \ni [\,] \overset{Chk}{\leadsto} *} \qquad \frac{\Gamma \vdash A \ni x \overset{Chk}{\leadsto} x' \quad \Gamma \vdash B\{x'/a\} \ni [xs] \overset{Chk}{\leadsto} xs'}{\Gamma \vdash (a\!:\!A) \times B \ni [x\; xs] \overset{Chk}{\leadsto} (a = x', xs'\!:\!B)}$$

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 $*$:

$$\dfrac{}{\Gamma \vdash \mathbb{1} \ni * \overset{\mathit{Chk}}{\rightsquigarrow} *} \qquad\qquad \dfrac{}{\Gamma \vdash \mathbb{1} \ni [] \overset{\mathit{Chk}}{\rightsquigarrow} *}$$

3.14 Example. Let $A\!:\!\textsc{Set}$, $B\!:\!A \to \textsc{Set}$, and $C\!:\!(x\!:\!A) \to B\,x \to \textsc{Set}$.

The tuple $[a\,b\,c]$ checked against the Σ-telescope $(x\!:\!A)(y\!:\!B\,x)(z\!:\!C\,x\,y) \times \mathbb{1}$ elaborates to the expected right-nested tuple:

$$\dfrac{\dfrac{\vdots}{\vdash A \ni a \overset{\mathit{Chk}}{\rightsquigarrow} t_a} \quad \dfrac{\dfrac{\vdots}{\vdash B\,t_a \ni b \overset{\mathit{Chk}}{\rightsquigarrow} t_b} \quad \dfrac{\dfrac{\vdots}{\vdash C\,t_a\,t_b \ni c \overset{\mathit{Chk}}{\rightsquigarrow} t_c} \quad \dfrac{}{\vdash \mathbb{1} \ni [] \overset{\mathit{Chk}}{\rightsquigarrow} *}}{\vdash (z\!:\!C\,t_a\,t_b) \times \mathbb{1} \ni [c] \overset{\mathit{Chk}}{\rightsquigarrow} (z = t_c, *\!:\!\mathbb{1})}}{\vdash (y\!:\!B\,t_a)(z\!:\!C\,t_a\,y) \times \mathbb{1} \ni [b\,c] \overset{\mathit{Chk}}{\rightsquigarrow} (y = t_b, (z = t_c, *\!:\!\mathbb{1})\!:\!C\,t_a\,y)}}{\vdash (x\!:\!A)(y\!:\!B\,x)(z\!:\!C\,x\,y) \times \mathbb{1} \ni [a\,b\,c] \overset{\mathit{Chk}}{\rightsquigarrow} (x = t_a, (y = t_b, (z = t_c, *\!:\!\mathbb{1})\!:\!C\,t_a\,y)\!:\!B\,x)}$$

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 $l\!:\!\mathsf{Tree}\,A$, $a\!:\!A$, and $r\!:\!\mathsf{Tree}\,A$, the node constructor of the binary tree $\mathsf{Tree}\,A$ is essentially encoded by the tuple $['\mathsf{node}\,l\,a\,r]$.

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

ELABORATION

$$\dfrac{}{\Gamma \vdash \mathsf{EnumU} \ni \{\} \overset{\mathit{Chk}}{\rightsquigarrow} \mathsf{nilE}} \qquad \dfrac{\Gamma \vdash \mathsf{UId} \ni t \overset{\mathit{Chk}}{\rightsquigarrow} t' \quad \Gamma \vdash \mathsf{EnumU} \ni \{ts\} \overset{\mathit{Chk}}{\rightsquigarrow} E}{\Gamma \vdash \mathsf{EnumU} \ni \{t\;ts\} \overset{\mathit{Chk}}{\rightsquigarrow} \mathsf{consE}\,t'\,E}$$

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

$$\dfrac{\dfrac{}{\vdash \mathsf{UId} \ni \mathsf{'a} \overset{Chk}{\leadsto} \mathsf{'a}} \qquad \dfrac{\dfrac{}{\vdash \mathsf{UId} \ni \mathsf{'b} \overset{Chk}{\leadsto} \mathsf{'b}} \qquad \dfrac{\dfrac{}{\vdash \mathsf{UId} \ni \mathsf{'c} \overset{Chk}{\leadsto} \mathsf{'c}} \quad \dfrac{}{\vdash \mathsf{EnumU} \ni \{\} \overset{Chk}{\leadsto} \mathsf{nilE}}}{\vdash \mathsf{EnumU} \ni \{\mathsf{'c}\} \overset{Chk}{\leadsto} \mathsf{consE\,'c\,nilE}}}{\vdash \mathsf{EnumU} \ni \{\mathsf{'b\,'c}\} \overset{Chk}{\leadsto} \mathsf{consE\,'b}\,(\mathsf{consE\,'c\,nilE})}}{\vdash \mathsf{EnumU} \ni \{\mathsf{'a\,'b\,'c}\} \overset{Chk}{\leadsto} \mathsf{consE\,'a}\,(\mathsf{consE\,'b}\,(\mathsf{consE\,'c\,nilE}))}$$

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

$$\{\mathsf{'a} \mapsto e_a \,;\, \mathsf{'b} \mapsto e_b \,;\, \mathsf{'c} \mapsto e_c\} : (e : \mathsf{EnumT}\,\{\mathsf{'a\,'b\,'c}\}) \to P\,e$$

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:

ELABORATION

$$\frac{\Gamma \vdash \mathsf{EnumU} \ni \{'l_0 \ldots 'l_k\} \overset{Chk}{\leadsto} E \qquad \Gamma \vdash \pi\,E\,P \ni [e_0 \ldots e_k] \overset{Chk}{\leadsto} ts}{\Gamma \vdash (e : \mathsf{EnumT}\,E) \to P\,e \ni \{'l_0 \mapsto e_0 \,;\, \ldots \,;\, 'l_k \mapsto e_k\} \overset{Chk}{\leadsto} \mathsf{switch}\,ts}$$

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 $E \triangleq \mathsf{consE\,'a}\,(\mathsf{consE\,'b}\,(\mathsf{consE\,'c\,nilE}))$, this definition unfolds to:

$$\frac{\vdash \mathsf{EnumU} \ni \{\mathsf{'a\,'b\,'c}\} \overset{Chk}{\leadsto} E \qquad \vdash \pi\,E\,P \ni [e_a\,e_b\,e_c] \overset{Chk}{\leadsto} (t_a, (t_b, (t_c, *)))}{\vdash (e : \mathsf{EnumT}\,E) \to P\,e \ni \{\mathsf{'a} \mapsto e_a \,;\, \mathsf{'b} \mapsto e_b \,;\, \mathsf{'c} \mapsto e_c\} \overset{Chk}{\leadsto} \mathsf{switch}\,(t_a, (t_b, (t_c, *)))}$$

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:

$$\vdash (e : \mathsf{EnumT}\ E) \to P\ e \ni \left\{ \begin{array}{l} \mathsf{'a} \mapsto e_a \\ \mathsf{'b} \mapsto e_b \\ \mathsf{'c} \mapsto e_c \end{array} \right\} \overset{Chk}{\leadsto} \mathsf{switch}\ (t_a, (t_b, (t_c, *)))$$

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

Elaboration

$$\frac{}{\Gamma \vdash \mathsf{consE}\ \mathsf{'t}\ E \ni \mathsf{'t} \overset{Chk}{\leadsto} 0} \qquad \frac{\mathsf{u} \neq \mathsf{t} \qquad \Gamma \vdash E \ni \mathsf{'t} \overset{Chk}{\leadsto} n}{\Gamma \vdash \mathsf{consE}\ \mathsf{'u}\ E \ni \mathsf{'t} \overset{Chk}{\leadsto} 1{+}\ n}$$

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 $\mathsf{'a}$ – the 0^{th} element – in the enumeration $\{\mathsf{'a}\ \mathsf{'b}\ \mathsf{'c}\}$:

$$\frac{}{\vdash \mathsf{consE}\ \mathsf{'a}\ (\mathsf{consE}\ \mathsf{'b}\ (\mathsf{consE}\ \mathsf{'c}\ \mathsf{nilE})) \ni \mathsf{'a} \overset{Chk}{\leadsto} 0}$$

3.23 Example. Similarly, we can index the tag $\mathsf{'b}$ – the 1^{st} element – in the enumeration $\{\mathsf{'a}\ \mathsf{'b}\ \mathsf{'c}\}$:

$$\frac{\dfrac{}{\vdash \mathsf{consE}\ \mathsf{'b}\ (\mathsf{consE}\ \mathsf{'c}\ \mathsf{nilE}) \ni \mathsf{'b} \overset{Chk}{\leadsto} 0}}{\vdash \mathsf{consE}\ \mathsf{'a}\ (\mathsf{consE}\ \mathsf{'b}\ (\mathsf{consE}\ \mathsf{'c}\ \mathsf{nilE})) \ni \mathsf{'b} \overset{Chk}{\leadsto} 1{+}\ 0}$$

3.1.3. Soundness

(3.24) Our elaboration procedure is *sound* if:

- The term elaborated by checking an expression 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 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)**.**

- If $\Gamma \vdash \mathsf{exprSynth} \overset{Syn}{\leadsto} t' \in T$, then
  there exists a level ℓ such that $\Gamma \vdash T : \textsc{Set}_\ell$ and $\Gamma \vdash t' : T$.
- If $\Gamma \vdash T \ni \mathsf{exprCheck} \overset{Chk}{\leadsto} t'$, 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

$$\cancel{\dfrac{}{\Gamma \vdash \mathsf{consE}\,\mathsf{'t}\,E \ni \mathsf{'t} \overset{Chk}{\leadsto} 0} \qquad \dfrac{}{\Gamma \vdash \mathsf{consE}\,\mathsf{'u}\,E \ni \mathsf{'t} \overset{Chk}{\leadsto} 0}}$$

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 $\mathcal{R} \subseteq X \times Y$ is *functional* if for all $x \in X$, and $y, z \in Y$, if $x \mathrel{\mathcal{R}} y$ and $x \mathrel{\mathcal{R}} z$ 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 Nat. Recall that the induction principle over natural numbers has type:

$$\mathsf{Nat\text{-}elim} : (P : \mathsf{Nat} \to \mathrm{SET})(n : \mathsf{Nat}) \to P\,0 \to ((m : \mathsf{Nat}) \to P\,m \to P\,(\mathsf{suc}\,m)) \to P\,n$$

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

$$+ : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \triangleq \lambda m\,n.\,\mathsf{Nat\text{-}elim}\;\; \begin{array}[t]{l} (\lambda m'.\,\mathsf{Nat})\,m \\ \{? : \mathsf{Nat}\} \\ \{? : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\} \end{array}$$

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

$$\begin{array}{lcl} \mathsf{f}\,(n : \mathsf{Nat}) & : & \mathsf{Nat} \\ \mathsf{f}\quad n & \mapsto & \{?\} \end{array}$$

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:

$$\begin{array}{lcl} \mathsf{vs} & : & \mathsf{Vec}\,\mathsf{Nat}\,(\mathsf{fib}\,14) \\ \mathsf{vs} & \mapsto & \{? : \mathsf{Vec}\,\mathsf{Nat}\,377\} \end{array}$$

(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 $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$. 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 $\wr \mathbf{f}\ t_0\ \ldots\ t_k : T \wr$ 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].

META-THEORY

$$\frac{\Gamma \vdash T : \textsc{Set}_\ell}{\Gamma \vdash \lbag l : T \rbag : \textsc{Set}_\ell}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \mathsf{return}\, t : \lbag l : T \rbag} \qquad \begin{array}{lcl} \mathsf{call}\lbag l : T \rbag\,(c : \lbag l : T \rbag) & : & T \\ \mathsf{call}\lbag l : T \rbag\,\,(\mathsf{return}\, t) & \mapsto & t \end{array}$$

Figure 3.3.: Programming labels

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

META-THEORY

$$\langle t \rangle ::= \ldots \quad | \quad \mathbf{f}\, \overrightarrow{\langle t \rangle}$$

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 $\lbag l : T \rbag$ is just a phantom type around T: we pack a term $t : T$ with return, and unpack it with $\mathsf{call}\lbag l : T \rbag$.

3.33 Example. Pursuing Example 3.29, we can express our intention of defining addition using the label $\lbag +\ m\, n : \mathsf{Nat} \rbag$:

$$\begin{array}{l} + : (m : \mathsf{Nat})(n : \mathsf{Nat}) \rightarrow \lbag +\ m\, n : \mathsf{Nat} \rbag \\ \quad \triangleq \lambda m\, n.\, \mathsf{Nat\text{-}elim}\ \ (\lambda m'.\, \lbag +\ m'\, n : \mathsf{Nat} \rbag)\, m \\ \qquad\qquad\qquad\qquad\ \ \{? : \lbag +\ 0\, n : \mathsf{Nat} \rbag\} \\ \qquad\qquad\qquad\qquad\ \ \{? : (m : \mathsf{Nat}) \rightarrow \lbag +\ m\, n : \mathsf{Nat} \rbag \rightarrow \lbag +\ (\mathsf{suc}\, m)\, n : \mathsf{Nat} \rbag\} \end{array}$$

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 $\mathsf{suc}\, m + n$ provided the result of $m + n$ (induction hypothesis).

In an implementation, the label $\lbag +\ m\, n : \mathsf{Nat} \rbag$ 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:

$$\begin{array}{llll} \mathsf{f}\,(x\!:\!X) & (y\!:\!Y) & \cdots & :\; Z \\ \mathsf{f}\quad x & \quad y & \cdots & \end{array}$$

That is, we specify the type of the function f on the first line, through a telescope of arguments x, y, *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:

$$\begin{array}{lclcl} (m\!:\!\mathsf{Nat}) & + & (n\!:\!\mathsf{Nat}) & : & \mathsf{Nat} \\ m & + & n & & \cdots \end{array}$$

(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:

- *Return* a value, using the "return" ($\mapsto$) gadget ; or
- *Eliminate* an argument, using the "by" ($\Leftarrow$) gadget ; or
- *Introduce* an argument under the function's scrutiny, using the "with" (|) gadget.

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 $m = 0$:

$$\begin{array}{lclcl} (m\!:\!\mathsf{Nat}) & + & (n\!:\!\mathsf{Nat}) & : & \mathsf{Nat} \\ m & + & n & \Leftarrow & \mathsf{Nat\text{-}elim}\, m \\ \quad 0 & + & n & \mapsto & n \\ \quad \mathsf{suc}\, m & + & n & \{?\} & \end{array}$$

(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:

$$
\begin{array}{llll}
(m : \mathsf{Nat}) + (n : \mathsf{Nat}) & : & \mathsf{Nat} \\
m \qquad\quad + \quad n & \Leftarrow & \mathsf{Nat\text{-}elim}\, m \\
\quad 0 \qquad\; + \quad n & \mapsto & n \\
\quad \mathsf{suc}\, m \;\; + \quad n & \mapsto & \mathsf{suc}\, (m + n)
\end{array}
$$

Indeed, the inductive step of the induction principle introduces a hypothesis of type $\wr + \; m \, n : \mathsf{Nat} \int$ 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:

- T-elim the elimination principle of T ;
- T-case the case analysis over T ; and
- T-rec the strong induction principle over T.

3.43 Example. For Peano numbers (*i.e.* the natural numbers Nat), these induction principles correspond to the propositions:

$$
\begin{aligned}
\mathsf{Nat\text{-}elim} &: (P : \mathsf{Nat} \to \mathrm{SET})(n : \mathsf{Nat}) \to P\, 0 \to ((m : \mathsf{Nat}) \to P\, m \to P\, (\mathsf{suc}\, m)) \to P\, n \\
\mathsf{Nat\text{-}case} &: (P : \mathsf{Nat} \to \mathrm{SET})(n : \mathsf{Nat}) \to P\, 0 \to ((m : \mathsf{Nat}) \to P\, (\mathsf{suc}\, m)) \to P\, n \\
\mathsf{Nat\text{-}rec} &: (P : \mathsf{Nat} \to \mathrm{SET})(n : \mathsf{Nat}) \to \\
&\qquad ((m : \mathsf{Nat}) \to ((k : \mathsf{Nat}) \to k \leq m \to P\, k) \to P\, (\mathsf{suc}\, m)) \to P\, n
\end{aligned}
$$

(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 Σ-telescope of arguments. For example, List is defined by:

$$\begin{array}{l}\textbf{data}\ \mathsf{List}\,[A\!:\!\textsc{Set}]\!:\!\textsc{Set}\ \textbf{where}\\ \quad \begin{array}{lll}\mathsf{List}\,A & \ni & \mathsf{nil}\\ & \mid & \mathsf{cons}\,(a\!:\!A)(as\!:\!\mathsf{List}\,A)\end{array}\end{array}$$

Ordinals also follow this pattern:

$$\begin{array}{l}\textbf{data}\ \mathsf{Ord}\!:\!\textsc{Set}\ \textbf{where}\\ \quad \begin{array}{lll}\mathsf{Ord} & \ni & \mathsf{0}\\ & \mid & \mathsf{suc}\,(o\!:\!\mathsf{Ord})\\ & \mid & \mathsf{lim}\,(l\!:\!\mathsf{Nat} \to \mathsf{Ord})\end{array}\end{array}$$

3.46 Remark. We declare datatype parameters in brackets – *e.g.* $[A : \textsc{Set}]$ in the definition of List above (¶ 3.45) – and datatype indices in parentheses – *e.g.* $(n : \mathsf{Nat})$ in the definition of Fin below (¶ 3.47). We make equational constraints on the latter only when needed, and explicitly – *e.g.* $(n = \mathsf{suc}\, n')$. The use of explicit equality constraints is reminiscent of the original syntax for GADTs[1] [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 *constrained* to some particular value. For example, vectors can be defined by constraining the index to be 0 in the nil case and $\mathsf{suc}\, n'$ for some $n' : \mathsf{Nat}$ in the cons case:

$$\begin{array}{l}\textbf{data}\ \mathsf{Vec}\,[A\!:\!\textsc{Set}](n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where}\\ \quad \begin{array}{llll}\mathsf{Vec}\,A & (n = \mathsf{0}) & \ni & \mathsf{nil}\\ \mathsf{Vec}\,A & (n = \mathsf{suc}\, n') & \ni & \mathsf{cons}\,(n'\!:\!\mathsf{Nat})(a\!:\!A)(vs\!:\!\mathsf{Vec}\,A\,n')\end{array}\end{array}$$

[1] 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.

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 fsuc by the predecessor:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathsf{Fin}\,(n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where}} \\
\quad \mathsf{Fin}\,(n = \mathsf{suc}\,n') & \ni & \mathsf{f0}\,(n'\!:\!\mathsf{Nat}) \\
 & | & \mathsf{fsuc}\,(n'\!:\!\mathsf{Nat})(k\!:\!\mathsf{Fin}\,n')
\end{array}$$

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

$$\cancel{\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathsf{Id}\,[a_1\!:\!A](a_2\!:\!A)\!:\!\textsc{Set}\ \textbf{where}} \\
\quad \mathsf{Id}\,a_1\,(a_2 = a_1) & \ni & \mathsf{refl}
\end{array}}$$

whose elimination principle gives the J-rule, sometimes strengthened to K [McBride, 1999, Hofmann and Streicher, 1994].

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:

$$\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ \mathsf{Vec}\,[A\!:\!\textsc{Set}](n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where}} \\
\quad \mathsf{Vec}\,A & n & \Leftarrow & \mathsf{Nat\text{-}case}\,n \\
\qquad \mathsf{Vec}\,A & 0 & \ni & \mathsf{nil} \\
\qquad \mathsf{Vec}\,A & (\mathsf{suc}\,m) & \ni & \mathsf{cons}\,(a\!:\!A)(vs\!:\!\mathsf{Vec}\,A\,m)
\end{array}$$

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 constructor in the 0 case, and the two expected constructors in the $\mathsf{suc}\,n$ case:

$$\begin{array}{l} \textbf{data}\ \mathsf{Fin}\,(n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where} \\ \quad \begin{array}{lcl} \mathsf{Fin}\quad 0 & \ni & \\ \mathsf{Fin}\,(\mathsf{suc}\,n) & \ni & \mathsf{f0} \\ & | & \mathsf{fsuc}\,(k\!:\!\mathsf{Fin}\,n) \end{array} \end{array}$$

(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 Γ of typed variables and a type T:

$$\begin{array}{l} \textbf{data}\ (\Gamma\!:\!\mathsf{Context}) \vdash\ (T\!:\!\mathsf{Type})\!:\!\textsc{Set}\ \textbf{where} \\ \quad \begin{array}{rcl} \Gamma \vdash \quad T & \ni & \mathsf{var}\,(v\!:\!T \in \Gamma) \\ & | & \mathsf{app}\,(S\!:\!\mathsf{Type})(f\!:\!\Gamma \vdash S \Rightarrow T)(s\!:\!\Gamma \vdash S) \\ \Gamma \vdash \quad T & \Leftarrow & \mathsf{Type\text{-}case}\ T \\ \Gamma \vdash\ \mathsf{unit} & \ni & * \\ \Gamma \vdash A \Rightarrow B & \ni & \mathsf{lam}\,(b\!:\!\Gamma\,;A \vdash B) \end{array} \end{array}$$

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

$$\begin{array}{l} \textbf{data}\ \mathsf{Type}\!:\!\textsc{Set}\ \textbf{where} \\ \quad \begin{array}{lcl} \mathsf{Type} & \ni & \mathsf{unit} \\ & | & (A\!:\!\mathsf{Type}) \Rightarrow (B\!:\!\mathsf{Type}) \end{array} \end{array} \qquad \begin{array}{l} \textbf{data}\ \mathsf{Context}\!:\!\textsc{Set}\ \textbf{where} \\ \quad \begin{array}{lcl} \mathsf{Context} & \ni & \epsilon \\ & | & (\Gamma\!:\!\mathsf{Context})\,;(T\!:\!\mathsf{Type}) \end{array} \end{array}$$

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

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

(3.53) 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.

(3.54) As in ML, unbound variables in type definitions are implicitly universally quantified. For example, we do not explicitly quantify over $n : \mathsf{Nat}$ or $A : \mathrm{SET}$ in the definition of vector lookup:

$$\begin{array}{l}\mathsf{vlookup}\;(m : \mathsf{Fin}\,n)\;(vs : \mathsf{Vec}\,A\,n)\;:\;A\\ \ldots\end{array}$$

(3.55) 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 $\forall\,x.\,(\ldots)$ – or its annotated variant $\forall\,x : T.\,(\ldots)$ – as follows:

$$\begin{array}{l}\mathsf{example}\;(op : \forall\,n.\,\mathsf{Vec}\,A\,n \to \mathbb{1})\;(xs : \mathsf{Vec}\,A\,k)\;(ys : \mathsf{Vec}\,A\,l)\;:\;op\,xs = op\,ys\\ \ldots\end{array}$$

(3.56) 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:

$$
\begin{array}{l}
(m{:}\mathsf{Nat}) + (n{:}\mathsf{Nat}) \ : \ \mathsf{Nat} \\
\ldots
\end{array}
$$

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

(3.57) 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 therefore define addition (Example 3.39) more concisely by:

$$
\begin{array}{lcll}
(m{:}\mathsf{Nat}) & + & (n{:}\mathsf{Nat}) & : \ \mathsf{Nat} \\
0 & + & n & \mapsto n \\
\mathsf{suc}\, m & + & n & \mapsto \mathsf{suc}\,(m+n)
\end{array}
$$

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

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ (\Gamma{:}\mathsf{Context}) \vdash \ (T{:}\mathsf{Type}){:}\textsc{Set}\ \textbf{where}} \\
\quad \Gamma \vdash \quad T & \ni & \mathsf{var}\,(v{:}T \in \Gamma) \\
 & \mid & \mathsf{app}\,(S{:}\mathsf{Type})(f{:}\Gamma \vdash S \Rightarrow T)(s{:}\Gamma \vdash S) \\
\quad \Gamma \vdash \ \ \mathsf{unit} & \ni & * \\
\quad \Gamma \vdash A \Rightarrow B & \ni & \mathsf{lam}\,(b{:}\Gamma\,;A \vdash B)
\end{array}
$$

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

$$
\lambda \begin{cases} \mathsf{true} \mapsto \mathsf{false} \\ \mathsf{false} \mapsto \mathsf{true} \end{cases} : \mathsf{Bool} \to \mathsf{Bool}
$$

(3.58) 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 nil case when taking the tail of a non-empty vector:

$$
\begin{array}{lll}
\mathsf{tl}\,(vs{:}\mathsf{Vec}\,A\,(\mathsf{suc}\,n)) & : & \mathsf{Vec}\,A\,n \\
\mathsf{tl}\quad (\mathsf{cons}\,a\,vs) & \mapsto & vs
\end{array}
$$

(3.59) 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 lengthVec:

$$\begin{array}{lcl} \mathsf{lengthVec}\,(vs\!:\!\mathsf{Vec}\,A\,n) & : & \mathsf{Nat} \\ \mathsf{lengthVec}\quad vs & \mapsto & n \end{array}$$

(3.60) By design, pattern variable must be *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 *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:

$$\begin{array}{llllcl} \mathsf{patt} & (m\!:\!\mathsf{Nat}) & (n\!:\!\mathsf{Nat}) & (q\!:\!m = n) & : & \mathbb{1} \\ \mathsf{patt} & m & m & \mathsf{refl} & \mapsto & * \end{array}$$

(3.61) 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

$$\begin{array}{llcl} \mathsf{Pow} & (X\!:\!\textsc{Set}) & : & \textsc{Set}_1 \\ \mathsf{Pow} & X & \mapsto & X \to \textsc{Set} \end{array}$$

and we would then be able to apply the powerset construction to an hypothetical T : $\textsc{Set}_2$, which shifts the resulting type uniformly:

$$\mathsf{Pow}\,T\!:\!\textsc{Set}_3$$

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.

Conclusion

(3.62) 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.

(3.63) 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.

(3.64) 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

(3.65) 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 subsumes 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

$$\begin{aligned} \mathit{Nat}\ X &= 1 + X \\ \mathit{List}_A\ X &= 1 + A \times X \\ \mathit{Tree}_A\ X &= 1 + A \times X \times X \\ \mathit{Ordinal}\ X &= 1 + X + X^{\mathbb{N}} \end{aligned}$$

where the sums encode the choice of constructors and the products encode the constructors' arguments. We have used an exponential notation $X^{\mathbb{N}}$ 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 Set. For instance, the earlier examples are modelled by the following functions on Set

$$
\begin{aligned}
\mathit{Nat}\ X &\triangleq \sum_{\mathsf{Bool}} \left\{ \begin{array}{ll} \mathsf{'true} & \mapsto \mathbb{1} \\ \mathsf{'false} & \mapsto X \end{array} \right\} \\
\mathit{List}_A\ X &\triangleq \sum_{\mathsf{Bool}} \left\{ \begin{array}{ll} \mathsf{'true} & \mapsto \mathbb{1} \\ \mathsf{'false} & \mapsto (a\!:\!A) \times X \end{array} \right\} \\
\mathit{Tree}_A\ X &\triangleq \sum_{\mathsf{Bool}} \left\{ \begin{array}{ll} \mathsf{'true} & \mapsto \mathbb{1} \\ \mathsf{'false} & \mapsto (a\!:\!A) \times (x\!:\!X) \times X \end{array} \right\} \\
\mathit{Ordinal}\ X &\triangleq \sum_{\mathsf{EnumT}\ \{\mathsf{'0}\ \mathsf{'suc}\ \mathsf{'lim}\}} \left\{ \begin{array}{ll} \mathsf{'0} & \mapsto \mathbb{1} \\ \mathsf{'cons} & \mapsto X \\ \mathsf{'lim} & \mapsto \mathbb{N} \to X \end{array} \right\}
\end{aligned}
$$

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

4.1. The Universe of Descriptions

Model: `Chapter4.Desc`[1]

(4.5) 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 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

[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.

SPECIFICATION

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathsf{Desc} : \textsc{Set}_1}$$

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash 1 : \mathsf{Desc}} \qquad \frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathsf{var} : \mathsf{Desc}}$$

$$\frac{\Gamma \vdash A : \mathsf{Desc} \quad \Gamma \vdash B : \mathsf{Desc}}{\Gamma \vdash A \times B : \mathsf{Desc}}$$

$$\frac{\Gamma \vdash S : \textsc{Set} \quad \Gamma \vdash D : S \to \mathsf{Desc}}{\Gamma \vdash \Sigma\, S\, D : \mathsf{Desc}}$$

$$\frac{\Gamma \vdash S : \textsc{Set} \quad \Gamma \vdash D : S \to \mathsf{Desc}}{\Gamma \vdash \Pi\, S\, D : \mathsf{Desc}}$$

(a) Code

META-THEORY

$$\begin{array}{lll}
\multicolumn{3}{l}{[\![(D : \mathsf{Desc})]\!]\ (X : \textsc{Set}) \ :\ \textsc{Set}} \\
[\![1]\!] & X & \mapsto \mathbb{1} \\
[\![\mathsf{var}]\!] & X & \mapsto X \\
[\![A \times B]\!] & X & \mapsto [\![A]\!]\, X \times [\![B]\!]\, X \\
[\![\Sigma\, S\, T]\!] & X & \mapsto (s : S) \times [\![T\, s]\!]\, X \\
[\![\Pi\, S\, T]\!] & X & \mapsto (s : S) \to [\![T\, s]\!]\, X
\end{array}$$

(b) Interpretation

Figure 4.1.: Universe of Descriptions

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 SET:

- Σ codes Σ-types – to build sigmas-of-sigmas;
- $\times$ codes products – the first-order, finitary counterpart of Π;
- Π codes Π-types – to capture higher-order arguments ;
- 1 codes the unit type – to terminate codes ;
- var codes the identity functor – to introduce the recursive argument (X in our examples).

We call an object $D : \mathsf{Desc}$ a *description*.

4.7 Remark (First-order representations)**.** Up to isomorphism, $\times$ is subsumed by Π, *i.e.*

we extensionally have that:

$$\begin{aligned} [\![A \times B]\!]\, X &\equiv [\![A]\!]\, X \times [\![B]\!]\, X \\ &\cong (b : \mathsf{Bool}) \to \begin{Bmatrix} \text{'true} \mapsto [\![A]\!]\, X \\ \text{'false} \mapsto [\![B]\!]\, X \end{Bmatrix} b \\ &\cong [\![\Pi\, \mathsf{Bool} \begin{Bmatrix} \text{'true} \mapsto A \\ \text{'false} \mapsto B \end{Bmatrix}]\!]\, X \end{aligned}$$

However, this apparent duplication has some value. Unlike its counterpart, × 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 $\mathbb{0} \to \mathsf{Nat}$. 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 (*i.e.* surjective pairing) than we can hope for functions (Remark 2.16).

4.8 Remark. Our original presentation of descriptions [Chapman et al., 2010] was based on a stripped-down version of induction-recursion [Dybjer and Setzer, 1999], where the code δ was restricted to a constant set-domain H. The universe was the following:

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \iota : \mathsf{Desc}}$$

$$\frac{\Gamma \vdash S : \textsc{Set} \qquad \Gamma \vdash T : S \to \mathsf{Desc}}{\Gamma \vdash \sigma\, S\, T : \mathsf{Desc}}$$

$$\frac{\Gamma \vdash H : \textsc{Set} \qquad \Gamma \vdash D : \mathsf{Desc}}{\Gamma \vdash \delta\, H\, D : \mathsf{Desc}}$$

$$\begin{array}{lll} \multicolumn{3}{l}{[\![(D : \mathsf{Desc})]\!]\ (X : \textsc{Set})\ :\ \textsc{Set}} \\ [\![\iota]\!] & X & \mapsto\ X \\ [\![\sigma\, S\, T]\!] & X & \mapsto\ (s : S) \times [\![T\, s]\!]\, X \\ [\![\delta\, H\, D]\!] & X & \mapsto\ (H \to X) \times [\![D]\!]\, X \end{array}$$

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 Π – which is the only code interpreting to an arrow – takes a constant 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 $S : \mathrm{Desc}$ in the domain of the Π 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 Σ code to be a $S : \mathrm{Desc}$: 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]

(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 $[\![-]\!]^{\rightarrow}$:

META-THEORY

$$\begin{array}{llll}
[\![(D\!:\!\mathsf{Desc})]\!]^{\rightarrow} & (f\!:\!X \rightarrow Y) & (xs\!:\![\![D]\!]\,X) & :\ [\![D]\!]\,Y \\
[\![1]\!]^{\rightarrow} & f & * & \mapsto * \\
[\![\mathsf{var}]\!]^{\rightarrow} & f & x & \mapsto f\,x \\
[\![A \times B]\!]^{\rightarrow} & f & (a,b) & \mapsto ([\![A]\!]^{\rightarrow}\,f\,a, [\![B]\!]^{\rightarrow}\,f\,b) \\
[\![\Sigma\,S\,T]\!] & f & (s,t) & \mapsto (s, [\![T\,s]\!]^{\rightarrow}\,f\,t) \\
[\![\Pi\,S\,T]\!]^{\rightarrow} & f & t & \mapsto \lambda s.\,[\![T\,s]\!]^{\rightarrow}\,f\,(t\,s)
\end{array}$$

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.

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

$$\begin{array}{l}
\mathsf{NatD}\ :\ \mathsf{Desc} \\
\mathsf{NatD} \mapsto \Sigma\,\mathsf{EnumT}\left\{\begin{array}{c}\text{'0} \\ \text{'suc}\end{array}\right\}\left\{\begin{array}{l}\text{'0}\ \ \mapsto 1 \\ \text{'suc} \mapsto \mathsf{var} \times 1\end{array}\right\}
\end{array}$$

This construction reads as follow. First, we use Σ 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 Σ-type to a binary sum, we have effectively described the desired functor:

$$[\![\mathsf{NatD}]\!]\ X \cong \mathbb{1} + X$$

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:

$$\begin{array}{lll} \mathsf{ListD}\,(A\,{:}\,\mathrm{SET}) & : & \mathsf{Desc} \\ \mathsf{ListD}\quad A & \mapsto & \Sigma\,\mathsf{EnumT}\left\{\begin{array}{c}\mathsf{'nil}\\ \mathsf{'cons}\end{array}\right\}\left\{\begin{array}{ll}\mathsf{'nil} & \mapsto 1 \\ \mathsf{'cons} & \mapsto \Sigma\,A\,\lambda -.\,\mathsf{var} \times 1\end{array}\right\} \end{array}$$

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

$$[\![\mathsf{ListD}\ A]\!]\ X \cong \mathbb{1} + A \times X$$

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:

$$\begin{array}{lll} \mathsf{TreeD}\,(A\,{:}\,\mathrm{SET}) & : & \mathsf{Desc} \\ \mathsf{TreeD}\quad A & \mapsto & \Sigma\,\mathsf{EnumT}\left\{\begin{array}{c}\mathsf{'leaf}\\ \mathsf{'node}\end{array}\right\}\left\{\begin{array}{ll}\mathsf{'leaf} & \mapsto 1 \\ \mathsf{'node} & \mapsto \mathsf{var} \times \Sigma\,A\,\lambda -.\,\mathsf{var} \times 1\end{array}\right\} \end{array}$$

Again, we are only one evolutionary step away from ListD. However, instead of a single call to the induction code, we add another one. The interpretation of this code is isomorphic to the expected functor:

$$[\![\mathsf{TreeD}\ A]\!]\ X \cong \mathbb{1} + X \times A \times X$$

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 $X \mapsto A + \mathbb{1}$. In our system, this translates to:

$$\begin{array}{lllll} \mathsf{MaybeD}\,(A\!:\!\mathrm{S}\small{\mathrm{ET}}) & : & \mathsf{Desc} \\ \mathsf{MaybeD} \quad A & \mapsto & \Sigma\,\mathsf{EnumT}\left\{\begin{array}{c}\mathsf{'just}\\ \mathsf{'nothing}\end{array}\right\}\left\{\begin{array}{ll}\mathsf{'just} & \mapsto \Sigma\,A\,\lambda - .\,1\\ \mathsf{'nothing} & \mapsto 1\end{array}\right\} \end{array}$$

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

$$[\![\mathsf{MaybeD}\ A]\!]\ X \cong A + \mathbb{1}$$

4.23 Definition (Tagged description)**.** From the examples above, we observe that a typical datatype definition consists of a Σ 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:

$$\begin{array}{lll} \mathsf{tagDesc} & : & \mathrm{SET}_1 \\ \mathsf{tagDesc} & \mapsto & (E\!:\!\mathsf{EnumU}) \times \pi\,E\,\lambda - .\,\mathsf{Desc} \end{array}$$

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

$$\begin{array}{lll} \mathsf{toDesc}\,(D\!:\!\mathsf{tagDesc}) & : & \mathsf{Desc} \\ \mathsf{toDesc} \quad (E, cs) & \mapsto & \Sigma\,(\mathsf{EnumT}\ E)\ (\mathsf{switch}\ cs) \end{array}$$

(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:

$$\begin{array}{lll} \mathsf{toTagDesc}\,(D\!:\!\mathsf{Desc}) & : & \mathsf{tagDesc} \\ \mathsf{toTagDesc} \quad D & \mapsto & (\mathsf{EnumT}\ \{\mathsf{'con}\}, (D, *)) \end{array}$$

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

$$\begin{array}{l} \mathsf{NatD} \ : \ \mathsf{tagDesc} \\ \mathsf{NatD} \mapsto \begin{cases} \mathsf{'0} & : \quad 1 \\ \mathsf{'suc} : & \mathsf{var} \times 1 \end{cases} \end{array}$$

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 toDesc and 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 μF. In this framework, the unique algebra homomorphism from the initial algebra to a given algebra $\alpha : F\ X \to X$ corresponds to a function from μF to X, which recursively applies α bottom-up on the inhabitant of μF.

However, not every signature functor admits an initial algebra. Typical examples include the powerset functor $X \mapsto X \to \mathsf{Bool}$ [Awodey, 2006], or, in a predicative setting, the functor $X \mapsto (X \to \mathsf{Bool}) \to \mathsf{Bool}$ [Reynolds, 1984, Pitts, 1987]. This is why we restrict ourselves to descriptions, which define only strictly-positive functors (¶ 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 $\mathbb{C}$.

An F-algebra is the pair of a *carrier* $C \in |\mathbb{C}|$ and a morphism $\alpha : F\, C \to C$ in $\mathbb{C}$.

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 $List_A\, X \mapsto 1 + A \times X$ be the signature functor of lists.

An example of a $List_A$-algebra on Nat is:

$$\begin{array}{lcl} \alpha\ (xs : List_A\ \mathsf{Nat}) & : & \mathsf{Nat} \\ \alpha \quad (\mathsf{inj}_l\ *) & \mapsto & 0 \\ \alpha\ \ (\mathsf{inj}_r\ (a, n)) & \mapsto & \mathsf{suc}\, n \end{array}$$

4.30 Definition (Category of F-algebras)**.** We organise the F-algebras in a category, denoted $F{-}alg$, whose objects are F-algebras (C, α), and for which a morphism from (C, α) to (D, β) is a morphism $f : C \to D$ in $\mathbb{C}$ such that the following diagram commutes:

$$\begin{tikzcd} F\,C \arrow[r, "F\,f"] \arrow[d, "\alpha"'] & F\,D \arrow[d, "\beta"] \\ C \arrow[r, "f"'] & D \end{tikzcd}$$

4.31 Definition (Initial F-algebra)**.** The initial F-algebra is the initial object in the category $F{-}alg$. 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 $(\!|\alpha|\!)$ in $\mathbb{C}$ making the following diagram commute:

$$\begin{tikzcd}
F\ (\mu F) \arrow[r, dashed, "F\ (\!|\alpha|\!)"] \arrow[d, "in"'] & F\ X \arrow[d, "\alpha"] \\
\mu F \arrow[r, dashed, "(\!|\alpha|\!)"'] & X
\end{tikzcd}$$

4.32 Remark (Terminology)**.** In the functional programming community, the morphism $(\!|\alpha|\!)$ is called the *catamorphism*, or *fold*.

4.33 Example. Applied to Example 4.29, initiality gives us a (unique) morphism $(\!|\alpha|\!)$: $\mu List_A \to \mathsf{Nat}$ that satisfies:

$$\begin{tikzcd}
List_A\ \mu List_A \arrow[r, dashed, "List_A\ (\!|\alpha|\!)"] \arrow[d, "in"'] & List_A\ \mathsf{Nat} \arrow[d, "\alpha"] \\
\mu List_A \arrow[r, dashed, "(\!|\alpha|\!)"'] & \mathsf{Nat}
\end{tikzcd}$$

That is, we have the equation:

$$\forall xs : List_A\ \mu List_A.\ (\!|\alpha|\!)(in\ xs) = \alpha(List_A\ (\!|\alpha|\!)\ xs) \tag{4.1}$$

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

$$\begin{aligned}
\mathsf{nil} &\triangleq in\ (\mathsf{inj}_l\ *) \\
\mathsf{cons}\, a\ ys &\triangleq in\ (\mathsf{inj}_r\ (a, ys)) && \text{for } a : A \text{ and } ys : \mu List_A
\end{aligned}$$

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

$$\begin{aligned} (\!|\alpha|\!)\,\mathsf{nil} &= (\!|\alpha|\!)\,(in\,(\mathsf{inj}_l\,*)) & (\!|\alpha|\!)\,(\mathsf{cons}\,a\,ys) &= (\!|\alpha|\!)(in\,(\mathsf{inj}_r\,(a,ys))) \\ &= \alpha(\mathsf{inj}_l\,*) & &= \alpha(\mathsf{inj}_r\,(a,(\!|\alpha|\!)\,ys)) \\ &= 0 & &= \mathsf{suc}\,((\!|\alpha|\!)\,ys) \end{aligned}$$

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

$$\begin{array}{lll} (\!|\alpha|\!) & \mathsf{List}\,A & : \mathsf{Nat} \\ (\!|\alpha|\!) & \mathsf{nil} & \mapsto 0 \\ (\!|\alpha|\!) & (\mathsf{cons}\,a\,xs) & \mapsto \mathsf{suc}\,((\!|\alpha|\!)\,xs) \end{array}$$

(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 μF datatype: they are the inhabitants of the carrier algebra given by in – such as nil and cons in Example 4.33.

Besides, the initiality of $(\mu F, in)$ gives a catamorphism, of type

$$(\!|(\alpha : F\,X \to X)|\!) : \mu F \to X$$

satisfying the recursive equation:

$$\forall xs : F\,\mu F.\ (\!|\alpha|\!)(in\,xs) = \alpha(F\,(\!|\alpha|\!)\,xs)$$

This equation captures the computational behavior of the catamorphism: it applies α recursively, *i.e.* from the datatypes leaves – such as nil in Example 4.33 – bubbling up along the recursively-defined constructors – such as 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 constructor:

4.35 Lemma (Lambek [1968])**.** Let $F : \mathbb{C} \to \mathbb{C}$ be a functor that admits an initial algebra $(\mu F, in)$.

The initial algebra $in : F\,\mu F \to \mu F$ is an isomorphism.

(4.36) **Computational interpretation.** In particular, Lambek's lemma tells us that the *only* inhabitants of the carrier μF *are* the constructors obtained from $F\,\mu F$. The initial algebra in is *the* generic datatype constructor. Applied to our Example 4.33, this means that the *only* constructors of List are nil and 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 considering any functor F, we now focus on the functor $[\![D]\!]$, for D 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.

4.38 Definition (Least fixpoint)**.** We extend our type theory with a new type former μ 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 in that plays the rôle of a generic datatype constructor:

META-THEORY

$$\frac{\Gamma \vdash D : \mathsf{Desc}}{\Gamma \vdash \mu\, D : \textsc{Set}} \qquad\qquad \frac{\Gamma \vdash xs : [\![D]\!]\, (\mu\, D)}{\Gamma \vdash \mathsf{in}\, xs : \mu\, D}$$

(4.39) **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 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.

4.2.2. Examples

(4.40) We can now define some actual datatypes and their constructors. We take this opportunity to recast our earlier examples as tagged descriptions. The resulting description code is exactly the same, but the tagged syntax is more palatable.

4.41 Example (Natural number, Example 4.18)**.**

$$\begin{array}{l}
\mathsf{Nat}\ :\ \textsc{Set} \\
\mathsf{Nat} \mapsto \mu \begin{cases} \mathsf{'0} & : \ 1 \\ \mathsf{'suc} : & \mathsf{var} \times 1 \end{cases}
\end{array}$$

$$\begin{array}{l}
\mathsf{0}\ :\ \mathsf{Nat} \\
\mathsf{0} \mapsto \mathsf{in}\, [\mathsf{'0}]
\end{array}$$

$$\begin{array}{lll}
\mathsf{suc}\, (n : \mathsf{Nat}) & : & \mathsf{Nat} \\
\mathsf{suc} \quad n & \mapsto & \mathsf{in}\, [\mathsf{'suc}\, n]
\end{array}$$

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:

$$
\begin{aligned}
[\textsf{'0}] &\triangleq (\textsf{'0}, *) \\
[\textsf{'suc}\, n] &\triangleq (\textsf{'suc}, (n, *))
\end{aligned}
$$

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

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

$$
\begin{array}{lcl}
\textsf{List}(A : \textsc{Set}) & : & \textsc{Set} \\
\textsf{List}\, A & \mapsto & \mu \begin{cases} \textsf{'nil} & : \quad 1 \\ \textsf{'cons} : & \Sigma\, A\, \lambda - .\, \textsf{var} \times 1 \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\textsf{nil} \ : \ \textsf{List}\, A \\
\textsf{nil} \mapsto \textsf{in}\, [\textsf{'nil}]
\end{array}
$$

$$
\begin{array}{llll}
\textsf{cons}\, (a : A) & (xs : \textsf{List}\, A) & : & \textsf{List}\, A \\
\textsf{cons} \quad a & \quad xs & \mapsto & \textsf{in}\, [\textsf{'cons}\, a\, xs]
\end{array}
$$

4.44 Example (Tree, Example 4.21)**.** Binary trees, on the other hand, take an extra recursive argument in the 'node case:

$$
\begin{array}{lcl}
\textsf{Tree}\, (A : \textsc{Set}) & : & \textsc{Set} \\
\textsf{Tree} \quad A & \mapsto & \mu \begin{cases} \textsf{'leaf} & : \quad 1 \\ \textsf{'node} : & \textsf{var} \times \Sigma\, A\, \lambda - .\, \textsf{var} \times 1 \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\textsf{leaf} \ : \ \textsf{Tree}\, A \\
\textsf{leaf} \mapsto \textsf{in}\, [\textsf{'leaf}]
\end{array}
$$

$$
\begin{array}{l}
\textsf{node}\, (lt : \textsf{Tree}\, A)\, (a : A)\, (rt : \textsf{Tree}\, A) \ : \ \textsf{Tree}\, A \\
\textsf{node}\, lt\, a\, rt \mapsto \textsf{in}\, [\textsf{'node}\, lt\, a\, rt]
\end{array}
$$

4.2.3. Induction

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

(4.45) Categorically, for an algebra (X, α), 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:

$$(\!|(\alpha : [\![D]\!]\, X \to X)|\!) : \mu\, D \to X$$

(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 $\square_D$ (Definition 4.52) and its associated lifting map $\square_D^{\rightarrow}$ (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 $P : \mu\, D \to \mathrm{SET}$ and an $x : \mu\, D$, it establishes $P\, x$. To do so, the user must provide the *inductive step* α:

META-THEORY

$$\begin{array}{l}\mathsf{induction}\ (\alpha : \square_D\, P \dot{\to} P \circ \mathsf{in})\ (x : \mu\, D)\ :\ P\, x \\ \mathsf{induction}\ \alpha\ (\mathsf{in}\ xs)\ \mapsto\ \alpha\ (\square_D^{\rightarrow}\ (\mathsf{induction}\ \alpha)\ xs)\end{array}$$

Intuitively, $\square_D$ – pronounced "everywhere in D" – represents the induction hypothesis (Definition 4.52): $\Box_D\, P\, xs$ states that P holds in all subnodes of xs, *i.e.* P holds everywhere in D. Following this reading, the function α of type $\Box_D\, P \dot{\to} P \circ \mathsf{in}$ 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 $\Box_D^{\to}$ (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 α is (actually) a $\Box_D$-algebra, we have used a compact notation for morphisms between predicates (¶ 1.32) and wrote:

$$\alpha : \Box_D\, P \dot{\to} P \circ \mathsf{in}$$

Put explicitly, this type signature unfolds to:

$$\alpha : (xs : [\![D]\!]\ (\mu\, D)) \to \Box_D\, P\, xs \to P(\mathsf{in}\, xs)$$

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

(4.50) **Computational interpretation.** The induction principle applies the inductive step α once, and then calls itself recursively on all subelements of xs using the canonical lifting map $\Box_D^{\to}$. 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 $xs : [\![D]\!]\ X$ and a predicate $P : X \to \textsc{Set}$, 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 $[\![-]\!]$ closely: indeed, it merely *lifts* the interpretation of descriptions – defining endofunctors on Set – over to functors on predicates. On all codes but var, it is identical and thus builds a D-structure. The key difference is in the treatment of the var code, where $\Box_{\mathsf{var}}\, P$ 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 the corresponding subnodes of xs.

META-THEORY

$$\begin{array}{llcll}
\Box_{(D\mathsf{Desc})} & (P : X \to \textsc{Set}) & (xs : [\![D]\!]\, X) & : & \textsc{Set} \\
\Box_{1} & P & * & \mapsto & \mathbb{1} \\
\Box_{\mathsf{var}} & P & x & \mapsto & P\, x \\
\Box_{A\,\times\,B} & P & (a, b) & \mapsto & \Box_{A}\, P\, a \times \Box_{B}\, P\, b \\
\Box_{\Sigma\, S\, D} & P & (s, d) & \mapsto & \Box_{D\, s}\, P\, d \\
\Box_{\Pi\, S\, D} & P & f & \mapsto & (s : S) \to \Box_{D\, s}\, P\, (f\, s)
\end{array}$$

Figure 4.2.: Canonical lifting

(4.54) The canonical lifting $\Box_D$ can be seen as a *monotonic predicate transformer*, taking a predicate $P : X \to \textsc{Set}$ on X – asserting a certain property on X – to the predicate $\Box_D : [\![D]\!]\, X \to \textsc{Set}$ on $[\![D]\!]\, X$. That latter predicate asserts that P holds on all subelements of $[\![D]\!]\, X$. Interestingly, $\Box_D$ 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:

$$\Box_{\mathsf{NatD}}\, P\, ['0] \leadsto \mathbb{1}$$

That is, there is no induction hypothesis. On the inductive step $'\mathsf{suc}$ from n, we have:

$$\Box_{\mathsf{NatD}}\, P\, ['\mathsf{suc}\, n] \leadsto P\, n \times \mathbb{1}$$

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

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 $'\mathsf{nil}$, we have, as expected, no induction hypothesis:

$$\Box_{\mathsf{ListD}\, A}\, P\, ['\mathsf{nil}] \leadsto \mathbb{1}$$

In the inductive step $'\mathsf{cons}$ from xs, we have:

$$\Box_{\mathsf{ListD}\, A}\, P\, ['\mathsf{cons}\, a\, xs] \leadsto P\, xs \times \mathbb{1}$$

That is, the induction hypothesis gives us $P\, xs$. During induction, from this $P\, xs$, we

META-THEORY

$$\begin{array}{llcll}
\Box^{\rightarrow}_{(D:\mathsf{Desc})} & (p:(x:X) \rightarrow P\,x) & (xs:[\![D]\!]\,X) & : & \Box_D\,P\,xs \\
\Box^{\rightarrow}_{1} & p & * & \mapsto & * \\
\Box^{\rightarrow}_{\mathsf{var}} & p & x & \mapsto & p\,x \\
\Box^{\rightarrow}_{A \times B} & p & (a,b) & \mapsto & (\Box^{\rightarrow}_{A}\,p\,a, \Box^{\rightarrow}_{B}\,p\,b) \\
\Box^{\rightarrow}_{\Sigma\,S\,D} & p & (s,d) & \mapsto & \Box^{\rightarrow}_{D\,s}\,p\,d \\
\Box^{\rightarrow}_{\Pi\,S\,D} & p & f & \mapsto & \lambda s.\,\Box^{\rightarrow}_{D\,s}\,p\,(f\,s)
\end{array}$$

Figure 4.3.: Lifting map

have to prove $P\,(\mathsf{in}\,[\text{'}\mathsf{cons}\,a\,xs])$, *i.e.* $P\,(\mathsf{cons}\,a\,n)$.

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:

$$\Box_{\mathsf{TreeD}\,A}\,P\,[\text{'}\mathsf{leaf}] \rightsquigarrow \mathbb{1}$$

In the inductive step 'node from xs, we have:

$$\Box_{\mathsf{TreeD}\,A}\,P\,[\text{'}\mathsf{node}\,lt\,a\,rt] \rightsquigarrow P\,lt \times P\,rt \times \mathbb{1}$$

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

(4.58) To describe the operational behavior of induction, we use the $\Box^{\rightarrow}_{D}$ 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 $\Box^{\rightarrow}_{D}\,p:(xs:[\![D]\!]\,X) \rightarrow \Box_D\,P\,xs$.

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

(4.60) **Computational interpretation.** Applying $\Box^{\rightarrow}_{D}\,p$ corresponds to taking a recursive step over a D-structure, applying p to all subnodes.

4.61 Remark (Categorical origin of $\Box^{\rightarrow}_{D}$)**.** We have presented $\Box^{\rightarrow}_{D}$ as the functorial action of $\Box_D$. 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 $P : X \to \textsc{Set}$ on X to predicates $\square_D\ P : [\![D]\!]\ X \to \textsc{Set}$ 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

$$Q \dot{\to} P \triangleq \forall\, x : X.\ Q\ x \to P\ x$$

to a morphism of type

$$\square_D\ Q \dot{\to} \square_D\ P \triangleq \forall\, xs : [\![D]\!]\ X.\ \square_D\ Q\ xs \to \square_D\ P\ xs$$

Looking closely at the type of $\square_D^{\to}$, we observe a similar pattern:

$$\square_D^{\to}\ (f : (x : X) \to P\ x) : (xs : [\![D]\!]\ X) \to \square_D\ P\ xs$$

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

$$\square_D^{\to}\ (f : \mathbf{1}\, X \dot{\to} P) : \mathbf{1}\, [\![D]\!]\ X \dot{\to} \square_D\ P$$

where $\mathbf{1}\, X \triangleq \lambda x : X.\ \mathbb{1}$ 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 $\square_D$ is a canonical lifting, we have that $\square_D\ (\mathbf{1}\, X) \cong \mathbf{1}\, ([\![D]\!]\ X)$ [Fumex, 2012, Definition 3.1.8]. We thus obtain the following equivalent type for $\square_D^{\to}$

$$\square_D^{\to}\ (f : \mathbf{1}\, X \dot{\to} P) : \square_D\ (\mathbf{1}\, X) \dot{\to} \square_D\ P$$

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

Note that we could provide the morphism part of $\square_D$ 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 $\square_D$ (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 $\square^{\rightarrow}_{D}$ (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 $\square_D$-algebra in disguise (Remark 4.49). This is justified categorically by the fact that induction is derivable from the initial algebra semantics of $\square_D$-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, 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 in, from which all other constructors are coded. For instance, in Example 4.41, we had to manually define the constructors of the 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:

ELABORATION

$$\frac{\Gamma \vdash \mu\,(\Sigma\,(\mathsf{EnumT}\,E)\;T) \ni \mathsf{in}\,[\text{'c}\,a_0 \ldots a_k] \overset{Chk}{\rightsquigarrow} t}{\Gamma \vdash \mu\,(\Sigma\,(\mathsf{EnumT}\,E)\;T) \ni c\,a_0 \ldots a_k \overset{Chk}{\rightsquigarrow} t}$$

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

$$\frac{\vdash \mathsf{Nat} \ni \mathsf{in}\,[\text{'0}] \overset{Chk}{\rightsquigarrow} \mathsf{in}\,(\text{'0}, *)}{\vdash \mathsf{Nat} \ni 0 \overset{Chk}{\rightsquigarrow} \mathsf{in}\,(\text{'0}, *)} \qquad \frac{\vdash \mathsf{Nat} \ni \mathsf{in}\,[\text{'suc}\,n] \overset{Chk}{\rightsquigarrow} \mathsf{in}\,(\text{'suc}, (n, *))}{\vdash \mathsf{Nat} \ni \mathsf{suc}\,n \overset{Chk}{\rightsquigarrow} \mathsf{in}\,(\text{'suc}, (n, *))}$$

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 Σ-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 in constructor. Inductive definitions ought to automatically and transparently translate to descriptions. And finally, using the *by* ($\Leftarrow$) gadget, we can readily appeal to induction on inductive types. In other words, we can already write programs manipulating datatypes.

Conclusion

(4.72) In this chapter, we have extended our type theory with a universe of descriptions, 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 *dependent* inductive types, its simplicity makes it an excellent platform for experimentation.

Related work

(4.73) 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 Π and Σ 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 *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 suc n) and offer stronger guarantees (pointwise addition of n-vectors yields an n-vector). However, these datatypes are not *individually* inductive. For instance, we have to define the whole *family* of vectors mutually, in one go:

$$\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ \mathsf{Vec}\,[A\!:\!\textsc{Set}](n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where}}\\
\quad \mathsf{Vec}\,A & (n=0) & \ni & \mathsf{nil}\\
\quad \mathsf{Vec}\,A & (n=\mathsf{suc}\,n') & \ni & \mathsf{cons}\,(n'\!:\!\mathsf{Nat})(a\!:\!A)(vs\!:\!\mathsf{Vec}\,A\,n')
\end{array}$$

The definition of vectors *indexed* by suc n depends on the definition of vectors indexed by n, and so on until reaching the empty vector indexed by 0. On the other hand, we could instantiate the *parameter* 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

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 I : SET, we play a similar game with endofunctors on the category SET^I, whose objects are families of sets $X, Y : I \to \textsc{Set}$, and whose morphisms are families of functions in $X \dot{\to} Y \triangleq \forall i : I.\, X\, i \to Y\, i$ (where the index i is kept implicit, as per ¶ 3.55).

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

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

Objects: the functions P, Q of codomain I

Morphisms: A morphism from P to Q is a function $f : \mathrm{dom}(P) \to \mathrm{dom}(Q)$ such that $P = Q \circ f$

The category $\textsc{Set}^I$ – *i.e.* the functor category from the discrete category I to $\textsc{Set}$ – and the slice category $\textsc{Set}/I$ 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 $\textsc{Set}$ let us describe mono-sorted signatures: there is only one sort of operation. When defining a signature functor from $\textsc{Set}/I$ to $\textsc{Set}/J$, 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 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 $\mathsf{suc}\, n$ 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 $\textsc{Set}^I$ to $\textsc{Set}^J$. However, we have that

$$[\textsc{Set}^I, \textsc{Set}^J] \cong [\textsc{Set}^I, \textsc{Set}]^J$$

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

$$(I \to \textsc{Set}) \to (J \to \textsc{Set}) \cong J \to (I \to \textsc{Set}) \to \textsc{Set}$$

This isomorphism lets us focus on describing functors from $\textsc{Set}^I \to \textsc{Set}$, 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 $[\textsc{Set}^I, \textsc{Set}]$. In Definition 5.12, we shall lift this universe to endofunctors on slices of $\textsc{Set}$. Our universe is thus defined by a set of codes IDesc and its extension $[\![-]\!]$, whose types are:

$$\mathsf{IDesc}\,(I : \textsc{Set}) : \textsc{Set} \qquad [\![-]\!] : \mathsf{IDesc}\, I \to (I \to \textsc{Set}) \to \textsc{Set}$$

An IDesc code is a syntactic object *desc*ribing a functor from $\textsc{Set}^I$ to $\textsc{Set}$. To compute its extension, we interpret the code with $[\![-]\!]$.

5.8 Definition (Indexed descriptions 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 ;
- Σ and Π respectively code Σ-types and Π-types of constant domain ;
- σ and $\times$ respectively code n-ary sum and binary product, using a first-order representation ;
- var codes a recursive argument, taken at a given sort $i : I$.

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 Π and Σ, we offer their first-order counterparts $\times$ and σ. Extensionally, we have:

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

However, intensionally, σ 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, σ, $\times$, and var admits a decidable equality (Example 7.96).

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

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

Specification

$$\frac{\Gamma \vdash I : \textsc{Set}}{\Gamma \vdash \mathsf{IDesc}\, I : \textsc{Set}_1}$$

$$\frac{\Gamma \vdash i : I}{\Gamma \vdash \mathsf{var}\, i : \mathsf{IDesc}\, I} \qquad \frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash 1 : \mathsf{IDesc}\, I}$$

$$\frac{\Gamma \vdash A : \mathsf{IDesc}\, I \quad \Gamma \vdash B : \mathsf{IDesc}\, I}{\Gamma \vdash A \times B : \mathsf{IDesc}\, I}$$

$$\frac{\Gamma \vdash E : \mathsf{EnumU} \quad \Gamma \vdash T : \pi\, E\, (\lambda \_ .\, \mathsf{IDesc}\, I)}{\Gamma \vdash \sigma\, E\, T : \mathsf{IDesc}\, I}$$

$$\frac{\Gamma \vdash S : \textsc{Set} \quad \Gamma \vdash T : S \to \mathsf{IDesc}\, I}{\Gamma \vdash \Pi\, S\, T : \mathsf{IDesc}\, I}$$

$$\frac{\Gamma \vdash S : \textsc{Set} \quad \Gamma \vdash T : S \to \mathsf{IDesc}\, I}{\Gamma \vdash \Sigma\, S\, T : \mathsf{IDesc}\, I}$$

(a) Code

Meta-theory

$$\begin{array}{lll}
\multicolumn{3}{l}{[\![(D : \mathsf{IDesc}\, I)]\!]\ (X : I \to \textsc{Set})\ :\ \textsc{Set}} \\
[\![\mathsf{var}\, i]\!] & X & \mapsto\ X\, i \\
[\![1]\!] & X & \mapsto\ \mathbb{1} \\
[\![A \times B]\!] & X & \mapsto\ [\![A]\!]\, X \times [\![B]\!]\, X \\
[\![\sigma\, E\, T]\!] & X & \mapsto\ (e : \mathsf{EnumT}\, E) \times [\![\mathsf{switch}\, T\, e]\!]\, X \\
[\![\Pi\, S\, T]\!] & X & \mapsto\ (s : S) \to [\![T\, s]\!]\, X \\
[\![\Sigma\, S\, T]\!] & X & \mapsto\ (s : S) \times [\![T\, s]\!]\, X
\end{array}$$

(b) Interpretation

Figure 5.1.: Universe of indexed descriptions

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

$$\begin{array}{ll} \mathsf{func}\,(I\!:\!\textsc{Set})\,(J\!:\!\textsc{Set})\ :\ \textsc{Set}_1 & [\![(D\!:\!\mathsf{func}\,I\,J)]\!]\ (X\!:\!I \rightarrow \textsc{Set})\ :\ J \rightarrow \textsc{Set} \\ \mathsf{func}\,I\,J\ \mapsto\ J \rightarrow \mathsf{IDesc}\,I & [\![D]\!]\ X\ \mapsto\ \lambda j.\,[\![D\,j]\!]\ X \end{array}$$

Inhabitants of the 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 nil is only available at index 0, while cons is only available at index 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 nil constructor ; if it is strictly positive, we describe the cons constructor:

$$\begin{array}{l} \mathsf{VecD}(A\!:\!\textsc{Set})\ :\ \mathsf{func}\,\mathsf{Nat}\,\mathsf{Nat} \\ \mathsf{VecD}\,A\quad 0\qquad \mapsto\ 1 \\ \mathsf{VecD}\,A\ (\mathsf{suc}\,n)\ \mapsto\ \Sigma\,A\,\lambda -.\,\mathsf{var}\,n \times 1 \end{array}$$

5.14 Remark (Strictly-positive family)**.** By induction on the interpretation function, we observe that the functors captured by the 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 containers [Abbott, 2003, Morris and Altenkirch, 2009].

5.2. Initial Algebra Semantics

(5.15) 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 $\textsc{Set}^I$, we compute the fixpoint at some index i and tie the knot with the fixpoint $\mu\, D$:

META-THEORY

$$\frac{\Gamma \vdash I : \textsc{Set} \qquad \Gamma \vdash D : \mathsf{func}\, I\, I}{\Gamma \vdash \mu\, D : I \to \textsc{Set}} \qquad \frac{\Gamma \vdash i : I \qquad \Gamma \vdash x : [\![D]\!]\ (\mu D)\ i}{\Gamma \vdash \mathsf{in}\, x : \mu\, D\, i}$$

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 $[\![D]\!]\ X \dot{\to} X$, there exists a (unique) function from $\mu\, D\, i$ to $X\, i$. In type theory, this corresponds to the catamorphism:

$$(\!|(\alpha : [\![D]\!]\ X \dot{\to} X)|\!) : \mu\, D \dot{\to} X$$

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`

(5.18) 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) 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 $\Box_D$ for the canonical lifting of an indexed description D (Definition 5.23), and $\Box_D^{\rightarrow}$ 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 $P : \mu\, D\, i \to \textsc{Set}$ and a term $x : \mu\, D\, i$, the induction principle lets us prove $P\, x$. To do so, we must prove the inductive step α:

META-THEORY

$$\mathsf{iinduction}\ (\alpha : \square_D\, P \dot{\to} P \circ \mathsf{in})\ (x : \mu\, D\, i)\ :\ P\, x$$
$$\mathsf{iinduction}\ \alpha\ (\mathsf{in}\ xs)\ \mapsto\ \alpha\ (\square^{\rightarrow}_D\ (\mathsf{iinduction}\ \alpha)\ xs)$$

Computationally, we take an inductive step α and, then, recursively apply induction on the substructures through the lifting map $\square^{\rightarrow}_D$.

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

$$\alpha : \square_D\, P \dot{\to} P \circ \mathsf{in}$$

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

$$\alpha : (i : I)(xs : [\![D]\!]\ (\mu\, D)\ i) \to \square_D\, P\, xs \to P\ (\mathsf{in}\ xs)$$

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 $\mathsf{in}\ xs$.

(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 $P : (i : I) \times X\, i \to \textsc{Set}$ on $(i : I) \times X\, i$ to a predicate $\square_D\, P : [\![D]\!]\ X \to \textsc{Set}$ on $[\![D]\!]\ X$. Interestingly, we can represent this functor with an indexed description. We thus define $\mathsf{D}\square_D$, a description indexed by $(i : I) \times X\, i$ and $[\![D]\!]\ X$. We obtain the canonical lifting $\square_D$ by interpreting $\mathsf{D}\square_D$ 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 func. Overloading notation, we obtain a description indexed

META-THEORY

$$\begin{array}{llcl}
\mathsf{D}\Box_{(D:\mathsf{IDesc}\,I)} & (xs : [\![D]\!]\,X) & : & \mathsf{IDesc}\,((i:I)\times X\,i) \\
\mathsf{D}\Box_{1} & * & \mapsto & 1 \\
\mathsf{D}\Box_{\mathsf{var}\,i} & x & \mapsto & \mathsf{var}(i, x) \\
\mathsf{D}\Box_{A\,\times\,B} & (a, b) & \mapsto & \mathsf{D}\Box_{A}\,a \times \mathsf{D}\Box_{B}\,b \\
\mathsf{D}\Box_{\sigma\,E\,D} & (e, d) & \mapsto & \mathsf{D}\Box_{\mathsf{switch}\,D\,e}\,d \\
\mathsf{D}\Box_{\Pi\,S\,D} & f & \mapsto & \Pi\,S\,\lambda s.\,\mathsf{D}\Box_{D\,s}\,(f\,s) \\
\mathsf{D}\Box_{\Sigma\,S\,D} & (s, d) & \mapsto & \mathsf{D}\Box_{D\,s}\,d
\end{array}$$

Figure 5.2.: Canonical lifting (internalised)

from predicates on $(i:I) \times X\,i$ to predicates on $(j:J) \times [\![D]\!]\,X\,j$:

$$\mathsf{D}\Box_{(D:\mathsf{func}\,I\,J)} : \mathsf{func}\,((i:I)\times X\,i)\,((j:J)\times [\![D]\!]\,X\,j)$$

from which we derive the predicate transformer $\Box_D$, by interpreting the description:

$$\begin{array}{l}
\Box_{(D:\mathsf{func}\,I\,I)}\,(P : (i:I)\times \mu\,D\,i \to \textsc{Set})\,(xs : [\![D]\!]\,(\mu\,D)\,i) \;:\; \textsc{Set} \\
\Box_D\,P\,xs \;\mapsto\; [\![\mathsf{D}\Box_D]\!]\,P\,xs
\end{array}$$

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 $n : \mathsf{Nat}$. First, for a vector of length $\mathsf{0}$, *i.e.* the empty vector, the lifting provides no induction hypothesis:

$$\Box_{\mathsf{VecD}\,A}\,P\,[\mathsf{'nil}] \leadsto \mathbb{1}$$

For a vector of length $\mathsf{suc}\,n$, *i.e.* a constructor node, the lifting unfolds to:

$$\Box_{\mathsf{VecD}\,A}\,P\,[\mathsf{'cons}\,a\,xs] \leadsto P\,(n, xs) \times \mathbb{1}$$

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

5.25 Remark. This definition of the canonical lifting is an example of *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 reifed 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

META-THEORY

$$\begin{array}{llcl}
\Box^{\rightarrow}_{(D:\mathsf{IDesc}\,I)} & (p:(x:X\,i)\rightarrow P\,x) & (xs:[\![D]\!]\,X) & :\ \Box_D\,P\,xs \\
\Box^{\rightarrow}_{1} & p & * & \mapsto *\\
\Box^{\rightarrow}_{\mathsf{var}\,i} & p & x & \mapsto p\,x\\
\Box^{\rightarrow}_{A\times B} & p & (a,b) & \mapsto (\Box^{\rightarrow}_{A}\,p\,a, \Box^{\rightarrow}_{B}\,p\,b)\\
\Box^{\rightarrow}_{\sigma\,E\,D} & p & (e,d) & \mapsto \Box^{\rightarrow}_{\mathsf{switch}\,D\,e}\,p\,d\\
\Box^{\rightarrow}_{\Pi\,S\,D} & p & f & \mapsto \lambda a.\,\Box^{\rightarrow}_{D\,a}\,p\,(f\,a)\\
\Box^{\rightarrow}_{\Sigma\,S\,D} & p & (s,d) & \mapsto \Box^{\rightarrow}_{D\,s}\,p\,d
\end{array}$$

Figure 5.3.: Lifting map

canonical lifting is useful on its own. It is indeed a generic program that asserts that a property P holds *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 $\Box^{\rightarrow}_{D}$)**.** 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 description. 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 $\Box^{\rightarrow}_{D}$)**.** In the non-indexed setting, we explained how the type of $\Box^{\rightarrow}_{D}$ can be massaged to reveal its functorial origin (Remark 4.61). The same reasoning can be carried out on $\Box^{\rightarrow}_{D}$.

(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 $\mathbb{1}$)**.** 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:

$$\begin{array}{lcl}
\mathsf{toIDesc}\,(D\!:\!\mathsf{Desc}) & : & \mathsf{IDesc}\,\mathbb{1} \\
\mathsf{toIDesc}\quad 1 & \mapsto & 1 \\
\mathsf{toIDesc}\quad \mathsf{var} & \mapsto & \mathsf{var}\,* \\
\mathsf{toIDesc}\quad (A \times B) & \mapsto & (\mathsf{toIDesc}\,A) \times (\mathsf{toIDesc}\,B) \\
\mathsf{toIDesc}\quad (\Sigma\, S\, D) & \mapsto & \Sigma\, S\, \lambda s.\, \mathsf{toIDesc}\,(D\, s) \\
\mathsf{toIDesc}\quad (\Pi\, S\, D) & \mapsto & \Pi\, S\, \lambda s.\, \mathsf{toIDesc}\,(D\, s)
\end{array}$$

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:

$$\mathsf{toIDesc}\,\mathsf{NatD} \leadsto \Sigma\,\mathsf{EnumT}\begin{Bmatrix}\text{'0}\\ \text{'suc}\end{Bmatrix}\begin{Bmatrix}\text{'0} & \mapsto 1\\ \text{'suc} & \mapsto \mathsf{var}\,* \times 1\end{Bmatrix}$$

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

$$\begin{array}{l}
\textbf{data}\ \mathsf{NTree}\,[A\!:\!\mathrm{SET}]\!:\!\mathrm{SET}\ \textbf{where} \\
\quad \mathsf{NTree}\,A \ni \mathsf{node}\,(a\!:\!A)(ts\!:\!\mathsf{NForest}\,A) \\[1ex]
\textbf{data}\ \mathsf{NForest}\,[A\!:\!\mathrm{SET}]\!:\!\mathrm{SET}\ \textbf{where} \\
\quad \mathsf{NForest}\,A \ni \mathsf{nil} \\
\qquad\qquad\qquad \mid\ \mathsf{cons}\,(t\!:\!\mathsf{NTree}\,A)(ts\!:\!\mathsf{NForest}\,A)
\end{array}$$

is described by the following code:

$$
\begin{array}{l}
\mathsf{TreeKind} \;:\; \textsc{Set} \\
\mathsf{TreeKind} \;\mapsto\; \mathsf{EnumT}\,\{\mathsf{'NTree}\;\mathsf{'NForest}\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{NTreesD}\,(A\!:\!\textsc{Set}) \;:\; \mathsf{func}\,\mathsf{TreeKind}\,\mathsf{TreeKind} \\
\mathsf{NTreesD}\,A\;\;\mathsf{'NTree} \;\mapsto\; \left\{ \mathsf{'node}:\;\; \Sigma\, A\,\lambda - .\,\mathsf{var}\,\mathsf{'NForest} \times 1 \right. \\
\mathsf{NTreesD}\,A\,\mathsf{'NForest} \;\mapsto\; \left\{ \begin{array}{ll} \mathsf{'nil} \;\;: & 1 \\ \mathsf{'cons}: & \mathsf{var}\,\mathsf{'NTree} \times \mathsf{var}\,\mathsf{'NForest} \times 1 \end{array} \right.
\end{array}
$$

$$
\begin{array}{l}
\mathsf{NTree}\,(A\!:\!\textsc{Set}) \;:\; \textsc{Set} \\
\mathsf{NTree}\quad A \quad\;\mapsto\; \mu\,(\mathsf{NTreesD}\,A)\,\mathsf{'NTree}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{NForest}\,(A\!:\!\textsc{Set}) \;:\; \textsc{Set} \\
\mathsf{NForest}\quad A \quad\;\mapsto\; \mu\,(\mathsf{NTreesD}\,A)\,\mathsf{'NForest}
\end{array}
$$

(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:

$$
\begin{array}{l}
\textbf{data}\;\mathsf{Vec}\,[A\!:\!\textsc{Set}](n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\;\textbf{where} \\
\quad \mathsf{Vec}\,A\quad (n\!=\!0) \quad\;\; \ni\; \mathsf{nil} \\
\quad \mathsf{Vec}\,A\,(n\!=\!\mathsf{suc}\,n') \;\ni\; \mathsf{cons}\,(n'\!:\!\mathsf{Nat})(a\!:\!A)(vs\!:\!\mathsf{Vec}\,A\,n')
\end{array}
$$

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:

$$
\begin{array}{l}
\mathsf{VecD}(A\!:\!\textsc{Set}) \;:\; \mathsf{func}\,\mathsf{Nat}\,\mathsf{Nat} \\
\mathsf{VecD}\,A \;\mapsto\; \lambda n.\,\Sigma\,\mathsf{EnumT} \left\{ \begin{array}{l} \mathsf{'nil} \\ \mathsf{'cons} \end{array} \right\} \left\{ \begin{array}{l} \mathsf{'nil} \quad\; \mapsto \Sigma\,(n = 0)\,\lambda - .\,1 \\ \mathsf{'cons} \mapsto \Sigma\,\mathsf{Nat}\,\lambda m.\,\Sigma\,(n = \mathsf{suc}\,m)\,\lambda - . \\ \qquad\qquad \Sigma\,A\,\lambda - .\,\mathsf{var}\,m \times 1 \end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Vec}\,(A\!:\!\textsc{Set})\,(n\!:\!\mathsf{Nat}) \;:\; \textsc{Set} \\
\mathsf{Vec}\,A\,n \;\mapsto\; \mu\,(\mathsf{VecD}\,A)\,n
\end{array}
$$

That is, we may choose nil for any index we like as long as that index is $\mathsf{0}$; 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 $\mathsf{Vec}\ A$, 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 $[\textsc{Set}/I, \textsc{Set}]^J$: 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:

$$
\begin{array}{l}
\mathsf{VecD}(A\!:\!\textsc{Set}) \;:\; \mathsf{func}\,\mathsf{Nat}\,\mathsf{Nat} \\
\mathsf{VecD}\ A \quad\ \mathsf{0} \quad\ \mapsto \left\{ \mathsf{nil}: \quad 1 \right. \\
\mathsf{VecD}\ A\ (\mathsf{suc}\,n) \mapsto \left\{ \mathsf{cons}: \quad \Sigma\, A\, \lambda -.\, \mathsf{var}\, n \times 1 \right.
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Vec}\ (A\!:\!\textsc{Set})\ (n\!:\!\mathsf{Nat}) \;:\; \textsc{Set} \\
\mathsf{Vec}\ A\ n \mapsto \mu\,(\mathsf{VecD}\ A)\ n
\end{array}
$$

The choice of constructors and equations have simply disappeared.

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

$$\begin{array}{l}\textbf{data}\ \mathsf{Fin}\ (n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where}\\ \quad \mathsf{Fin}\ (n = \mathsf{suc}\, n') \ni \mathsf{f0}\ (n'\!:\!\mathsf{Nat})\\ \qquad\qquad\qquad\quad \mid\ \mathsf{fsuc}\ (n'\!:\!\mathsf{Nat})(k\!:\!\mathsf{Fin}\, n')\end{array}$$

This definition is described by the following description:

$$\begin{array}{l}\mathsf{FinD}\ :\ \mathsf{func}\,\mathsf{Nat}\,\mathsf{Nat}\\ \mathsf{FinD}\, n\ \mapsto \begin{cases}\text{'}\mathsf{f0} & :\ \Sigma\,\mathsf{Nat}\,\lambda m.\,\Sigma\,(n = \mathsf{suc}\, m)\ \lambda - .\, 1\\ \text{'}\mathsf{fsuc} & :\ \Sigma\,\mathsf{Nat}\,\lambda m.\,\Sigma\,(n = \mathsf{suc}\, m)\ \lambda - .\,\mathsf{var}\, m \times 1\end{cases}\end{array}$$

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

$$\begin{array}{l}\mathsf{FinD}\ :\ \mathsf{func}\,\mathsf{Nat}\,\mathsf{Nat}\\ \mathsf{FinD}\quad 0\quad\ \ \mapsto \Sigma\,\mathbb{0}\ \mathbb{0}\text{-}\mathsf{elim}\qquad \textit{– No constructor}\\ \mathsf{FinD}\ (\mathsf{suc}\, n)\ \mapsto \begin{cases}\text{'}\mathsf{f0} & :\ 1\\ \text{'}\mathsf{fsuc} & :\ \mathsf{var}\, n \times 1\end{cases}\end{array}$$

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 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 Vec (Example 5.38) is obtained by detagging. By noticing whether the index is 0 or suc, we deduce the vector constructor.

5.44 Definition (Tagged indexed descriptions)**.** Let us reflect this index analysis technique in the structure of descriptions, thus extending tagged descriptions to an indexed setting (Definition 4.23). We divide tagged indexed descriptions into two categories:

first, the constructors that do not depend on the index, inhabiting $\mathsf{Tags}\, I$; then, the constructors that do, inhabiting $\mathsf{iTags}\, I$:

$$\begin{array}{lll} \mathsf{tagIDesc}\,(I\!:\!\textsc{Set}) & : & \textsc{Set}_1 \\ \mathsf{tagIDesc}\quad I & \mapsto & \mathsf{Tags}\, I \times \mathsf{iTags}\, I \end{array}$$

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

$$\begin{array}{lll} \mathsf{Tags}\,(I\!:\!\textsc{Set}) & : & \textsc{Set}_1 \\ \mathsf{Tags}\quad I & \mapsto & (E\!:\!\mathsf{EnumU}) \times (i\!:\!I) \to \pi\, E\, \lambda - .\, \mathsf{IDesc}\, I \end{array}$$

The index-dependent part indexes the choice of constructors. Thus, by inspecting the index, it is possible to vary the choice of constructors:

$$\begin{array}{lll} \mathsf{iTags}\,(I\!:\!\textsc{Set}) & : & \textsc{Set}_1 \\ \mathsf{iTags}\quad I & \mapsto & (F\!:\!I \to \mathsf{EnumU}) \times (i\!:\!I) \to \pi\, (F\, i)\, \lambda - .\, \mathsf{IDesc}\, I \end{array}$$

(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 Vec (Example 5.38) is more concisely defined as:

$$\begin{array}{lll} \mathsf{VecD}(A\!:\!\textsc{Set}) & : & \mathsf{func}\,\mathsf{Nat}\,\mathsf{Nat} \\ \mathsf{VecD}\; A & \mapsto & \left\{ \begin{array}{rlcl} 0 & \leftarrow \mathsf{'nil} & : & 1 \\ \mathsf{suc}\, n & \leftarrow \mathsf{'cons} & : & \Sigma\, A\, \lambda - .\, \mathsf{var}\, n \times 1 \end{array} \right. \end{array}$$

Similarly, the signature of the judgments of minimal logic (Example 3.51) is given by:

$$\begin{array}{lll} \vdash^D & : & \mathsf{func}\,(\mathsf{Context} \times \mathsf{Type})\,(\mathsf{Context} \times \mathsf{Type}) \\ \vdash^D & \mapsto & \left\{ \begin{array}{rlcl} (\Gamma, T) & \leftarrow \mathsf{'var} & : & \Sigma\, (T \in \Gamma)\, \lambda - .\, 1 \\ (\Gamma, T) & \leftarrow \mathsf{'app} & : & \Sigma\, \mathsf{Type}\, \lambda S.\, \mathsf{var}\,(\Gamma, S \Rightarrow T) \times \mathsf{var}\,(\Gamma, S) \times 1 \\ (\Gamma, \mathsf{unit}) & \leftarrow \mathsf{'}{*} & : & 1 \\ (\Gamma, A \Rightarrow B) & \leftarrow \mathsf{'lam} & : & \mathsf{var}\,(\Gamma\, ; A, B) \end{array} \right. \end{array}$$

5.47 Remark (Notation)**.** We write $\mathsf{toIDesc}\, D$ to denote the description computed from the tagged indexed description D. Its expansion is similar to the definition of 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 descriptions, and back. In particular, we write $\mu\, D$ instead of $\mu\, (\mathsf{toIDesc}\, D)$.

5.48 Example (Typed expressions)**.** As a larger example, we define a syntax for a small language with two types, natural numbers and Booleans:

$$\begin{array}{lcl} \mathsf{Ty} & : & \textsc{Set} \\ \mathsf{Ty} & \mapsto & \mathsf{EnumT}\, \{\mathsf{'nat}\, \mathsf{'bool}\} \end{array}$$

This language has values, conditional expression, addition and comparison. Informally, their signatures are:

$$\begin{array}{lll} \mathsf{val} & : & \mathsf{Val}\, ty \to ty \\ \mathsf{cond} & : & \mathsf{'bool} \to ty \to ty \to ty \end{array} \qquad \begin{array}{lll} \mathsf{plus} & : & \mathsf{'nat} \to \mathsf{'nat} \to \mathsf{'nat} \\ \mathsf{le} & : & \mathsf{'nat} \to \mathsf{'nat} \to \mathsf{'bool} \end{array}$$

The function Val interprets the object language types in the host language, so that arguments to val fit their expected type.

$$\begin{array}{llcl} \mathsf{Val}\, (ty : \mathsf{Ty}) & & : & \textsc{Set} \\ \mathsf{Val} & \mathsf{'nat} & \mapsto & \mathsf{Nat} \\ \mathsf{Val} & \mathsf{'bool} & \mapsto & \mathsf{Bool} \end{array}$$

We take Nat and Bool to represent natural numbers and Booleans in the host language, equipped with addition and comparison:

$$- + - : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$
$$- \leq - : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool}$$

We express our syntax as a tagged description, indexing over object types Ty:

$$\begin{array}{lcl} \mathsf{ExprD} & : & \mathsf{tagIDescTy} \\ \mathsf{ExprD} & \mapsto & \left\{ \begin{array}{lcl} ty & \leftarrow \mathsf{'val} & : \Sigma\, (\mathsf{Val}\, ty)\, \lambda - .\, 1 \\ ty & \leftarrow \mathsf{'cond} & : \mathsf{var}\, \mathsf{'bool} \times \mathsf{var}\, ty \times \mathsf{var}\, ty \times 1 \\ \mathsf{'nat} & \leftarrow \mathsf{'plus} & : \mathsf{var}\, \mathsf{'nat} \times \mathsf{var}\, \mathsf{'nat} \times 1 \\ \mathsf{'bool} & \leftarrow \mathsf{'le} & : \mathsf{var}\, \mathsf{'nat} \times \mathsf{var}\, \mathsf{'nat} \times 1 \end{array} \right. \end{array}$$

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) containers [Abbott, 2003, Morris and Altenkirch, 2009]. 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].

(5.50) 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

$$\cancel{\frac{\Gamma \vdash p : x = y}{\Gamma \vdash x \equiv y : A}}\ \textsc{Ext}$$

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`

(5.52) 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 $I, J : \mathrm{SET}$.

A *container* indexed by I and J is a triple

$$\begin{cases} \mathsf{Op} : & J \to \mathrm{SET} \\ \mathsf{Ar} : & \mathsf{Op}\, j \to \mathrm{SET} \\ \mathsf{Sort} : & \mathsf{Ar}\, op \to I \end{cases}$$

where Op stands for "operations", and Ar stands for "arities". Such a triple is denoted $\mathsf{Op} \lhd^{\mathsf{Sort}} \mathsf{Ar}$. The class of containers indexed by I and J is denoted $\mathsf{ICont}\, I\, J$.

(5.54) **Intuition.** Containers can be understood as multi-sorted signatures. The indices specify the sorts. The set Op for a given index j specifies the operations available at that sort. The set Ar for a given operation op specifies the arity of that operation. The function 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:

$$\mathsf{NatCont} \triangleq \begin{cases} \mathsf{Op}_{\mathsf{Nat}}\ (*:\mathbb{1}) \ : \ \textsc{Set} \\ \mathsf{Op}_{\mathsf{Nat}}\quad *\quad \mapsto \mathbb{1}+\mathbb{1} \\[1ex] \mathsf{Ar}_{\mathsf{Nat}}\ (op:\mathsf{Op}_{\mathsf{Nat}}\ *) \ : \ \textsc{Set} \\ \mathsf{Ar}_{\mathsf{Nat}}\quad (\mathsf{inj}_l\ *)\quad \mapsto \mathbb{0} \\ \mathsf{Ar}_{\mathsf{Nat}}\quad (\mathsf{inj}_r\ *)\quad \mapsto \mathbb{1} \\[1ex] \mathsf{Sort}_{\mathsf{Nat}}\ (ar:\mathsf{Ar}_{\mathsf{Nat}}\ op) \ : \ \mathbb{1} \\ \mathsf{Sort}_{\mathsf{Nat}}\quad ar\quad \mapsto * \end{cases}$$

There are two operations, one to represent the 0 case, the other to represent the successor case, suc. For the arities, none is offered by the operation 0, while the operation 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:

$$\mathsf{ListCont}_A \triangleq \begin{cases} \mathsf{Op}_{\mathsf{List}}\ (*:\mathbb{1}) \ : \ \textsc{Set} \\ \mathsf{Op}_{\mathsf{List}}\quad *\quad \mapsto \mathbb{1}+A \\[1ex] \mathsf{Ar}_{\mathsf{List}}\ (op:\mathsf{Op}_{\mathsf{List}}\ *) \ : \ \textsc{Set} \\ \mathsf{Ar}_{\mathsf{List}}\quad (\mathsf{inj}_l\ *)\quad \mapsto \mathbb{0} \\ \mathsf{Ar}_{\mathsf{List}}\quad (\mathsf{inj}_r\ a)\quad \mapsto \mathbb{1} \\[1ex] \mathsf{Sort}_{\mathsf{List}}\ (ar:\mathsf{Ar}_{\mathsf{List}}\ op) \ : \ \mathbb{1} \\ \mathsf{Sort}_{\mathsf{List}}\quad ar\quad \mapsto * \end{cases}$$

Note the similarity with natural numbers. There are $1 + A$ operations, *i.e.* either the empty list nil or the list constructor cons of some $a : A$. The arity of the operation nil is nil, while the operation cons has arity one. Again, indices are trivial, for this datatype 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 $\{X_n \mid n \in \mathsf{Nat}\} \mapsto \{n = 0 \mid n \in \mathsf{Nat}\} + \{A \times X_{n-1} \mid n \in \mathsf{Nat}, n \neq 0\}$ where the empty vector nil requires the length n to be 0, while the vector constructor 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:

$$\mathsf{VecCont}_A \triangleq \begin{cases} \mathsf{Op}_{\mathsf{Vec}}\,(n\!:\!\mathsf{Nat}) \;:\; \mathrm{SET} \\ \mathsf{Op}_{\mathsf{Vec}}\quad 0 \quad\mapsto \mathbb{1} \\ \mathsf{Op}_{\mathsf{Vec}}\;(\mathsf{suc}\,n) \;\mapsto A \\ \\ \mathsf{Ar}_{\mathsf{Vec}}\,(n\!:\!\mathsf{Nat})\,(op\!:\!\mathsf{Op}_{\mathsf{Vec}}\,n) \;:\; \mathrm{SET} \\ \mathsf{Ar}_{\mathsf{Vec}}\quad 0 \quad * \quad\mapsto \mathbb{0} \\ \mathsf{Ar}_{\mathsf{Vec}}\;(\mathsf{suc}\,n) \quad a \quad\mapsto \mathbb{1} \\ \\ \mathsf{Sort}_{\mathsf{Vec}}\,(n\!:\!\mathsf{Nat})\,(op\!:\!\mathsf{Op}_{\mathsf{Vec}}\,n)\,(ar\!:\!\mathsf{Ar}_{\mathsf{Vec}}\,ar) \;:\; \mathsf{Nat} \\ \mathsf{Sort}_{\mathsf{Vec}}\;(\mathsf{suc}\,n) \quad a \quad * \quad\mapsto n \end{cases}$$

At index 0, only the operation nil is available while the index suc n offers a choice of A operations. As for lists, the arity of the operation nil is nil while the operation cons has arity one. It is necessary to compute the sort (*i.e.* the length of the tail) only when the input index is 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 (Σ-type) of operation ; then, for each (Π-type) arity, a variable X taken at the sort for that arity:

$$\begin{array}{l} [\![(\sigma\!:\!\mathsf{ICont}\,I\,J)]\!]_{\mathsf{Cont}}\,(X\!:\!I \to \mathrm{SET}) \;:\; J \to \mathrm{SET} \\ [\![\mathsf{Op}\lhd^{\mathsf{Sort}}\mathsf{Ar}]\!]_{\mathsf{Cont}}\,X \;\mapsto\; \lambda j.\,(op\!:\!\mathsf{Op}\,j) \times ((ar\!:\!\mathsf{Ar}\,op) \to X\,(\mathsf{Sort}\,ar)) \end{array}$$

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 Op, Ar, and Sort such that

$$F \cong \lambda j.\,\lambda X.\,(op\!:\!\mathsf{Op}\,j) \times ((ar\!:\!\mathsf{Ar}\,op) \to X\,(\mathsf{Sort}\,ar))$$

This terminology is justified by fact that the interpretation computes a multivariate polynomial (in J variables) defined as an Op-indexed sum of monomials X taken at some exponent $ar\!:\!\mathsf{Ar}\,op$, or put informally:

$$[\![\mathsf{Op}\lhd^{\mathsf{Sort}}\mathsf{Ar}]\!]_{\mathsf{Cont}}\;\{X_i \mid i \in I\} \mapsto \left\{ \sum_{op \in \mathsf{Op}_j} \prod_{ar \in \mathsf{Ar}_{op}} X_{\mathsf{Sort}\,ar} \mid j \in J \right\}$$

(5.60) We leave it to the reader to verify that the interpretation of NatCont (Example 5.55), ListCont (Example 5.56), and 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].

5.3.2. 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 $D : \mathsf{func}\; I\; J$ 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 $F : \mathrm{SET}^I \to \mathrm{SET}^J$ 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 $j : J$, there is a $D : \mathsf{IDesc}\; I$ 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 $\mathsf{Op}_D : \mathrm{SET}$, $\mathsf{Ar}_D : \mathsf{Op}_D \to \mathrm{SET}$, and $\mathsf{Sort}_D : \mathsf{Ar}_D\; op \to I$ such that $[\![D]\!]\; X \cong (op : \mathsf{Op}_D) \times (ar : \mathsf{Ar}_D\; op) \to X\; (\mathsf{Sort}_D\; ar)$. We proceed by induction over D:

Case $D = 1$**:** We have $[\![1]\!]\; X \cong \mathbb{1} \times X^{\mathbb{0}}$, which is clearly polynomial.

Case $D = \mathbf{var}\; i$**:** We have $[\![\mathsf{var}\; i]\!]\; X \cong \mathbb{1} \times (X\; i)^{\mathbb{1}}$, which is clearly polynomial.

Case $D = \Sigma\; S\; T$**:** We have $[\![\Sigma\; S\; T]\!]\; X = (s : S) \times [\![T\; s]\!]\; X$. By induction hypothesis, $[\![T\; s]\!]\; X \cong (x : S_{T\, s}) \times (p : P_{T\, s}\; x) \to X\; (n_{T\, s}\; p)$. Therefore, we have that:

$$\begin{aligned} [\![\Sigma\; S\; T]\!]\; X &\cong (s : S) \times (x : S_{T\, s}) \times (p : P_{T\, s}) \to X\; (n_{T\, s}\; p) \\ &\cong (sx : (s : S) \times S_{T\, s}) \times (p : P_{T\, (\pi_0\, sx)} (\pi_1\; sx)) \to X\; (n_{T\, (\pi_0\, sx)}\; p) \end{aligned}$$

This last functor is clearly polynomial.

Case $D = \Pi\, S\, T$**:** We have $[\![\Pi\, S\, T]\!]\, X \;=\; (s : S) \to [\![T\, s]\!]\, X$. By induction hypothesis, $[\![T\, s]\!]\, X \cong (x : S_{T\,s}) \times (p : P_{T\,s}\, x) \to X\, (n_{T\,s}\, p)$. Therefore, we have that:

$$\begin{aligned}
[\![\Pi\, S\, T]\!]\, X &\cong (s : S) \to (x : S_{T\,s}) \times (p : P_{T\,s}\, x) \to X\, (n_{T\,s}\, p) \\
&\cong (f : (s : S) \to S_{T\,s}) \times (s : S)(p : P_{T\,s}\, (f\, s)) \to X\, (n_{T\,s}\, p) \\
&\cong (f : (s : S) \to S_{T\,s}) \times (sp : (s : S) \times P_{T\,s}\, (f\, s)) \to X\, (n_{T\,(\pi_0\, sp)}\, (\pi_1\, sp))
\end{aligned}$$

This last functor is clearly polynomial.

□

(5.65) 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 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 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 *reindexes* a predicate $P : J \to \textsc{Set}$ into a predicate over I along a renaming function $\sigma : I \to J$:

$$\begin{array}{llll}
\Delta_{(\sigma{:}I \to J)}\, (P : J \to \textsc{Set})\, (I : \textsc{Set}) & : & \textsc{Set} \\
\Delta_{\sigma} \quad\quad\quad\quad P \quad\quad\quad\quad\; i & \mapsto & P(\sigma\, i)
\end{array}$$

Seen as a functor from $\textsc{Set} / J$ to $\textsc{Set} / I$, the reindexing functor Δ_{σ} has a left adjoint Σ_{σ}, which generalises Σ-types:

$$\begin{array}{lll}
\Sigma_{(\sigma{:}I \to J)}\, (P : I \to \textsc{Set})\, (j : J) & : & \textsc{Set} \\
\Sigma_{\sigma} \quad\quad\quad\quad P \quad\quad\quad\quad j & \mapsto & (i : I) \times \sigma\, i = j \times P\, i
\end{array}$$

It also has a right adjoint Π_{σ}, which generalises Π-types:

$$\begin{array}{lll}
\Pi_{(\sigma{:}I \to J)}\, (P : I \to \textsc{Set})\, (j : J) & : & \textsc{Set} \\
\Pi_{\sigma} \quad\quad\quad\quad P \quad\quad\quad\quad j & \mapsto & (i : I) \times \sigma\, i = j \to P\, i
\end{array}$$

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:

$$\begin{array}{l} \textbf{data}\ [f : A \to B]^{-1}\ (b : B) : \textsc{Set}\ \textbf{where} \\ \quad f^{-1}\ (b = f\ a)\ \ni \mathsf{inv}\ (a : A) \end{array}$$

Equivalently, it can be defined with a Σ-type:

$$\begin{array}{lcl} (f : A \to B)^{-1}\ (b : B) & : & \textsc{Set} \\ f^{-1}\ b & \mapsto & (a : A) \times f\ a = b \end{array}$$

5.68 Lemma. Described functors are closed under reindexing and its adjoints.

Proof. We describe the pullback functors and their adjoints by:

$$\begin{array}{lcl} D\Delta_{(\sigma A \to B)} & : & \mathsf{func}\ B\ A \\ D\Delta_\sigma & \mapsto & \lambda a.\ \mathsf{var}\ (\sigma\ a) \end{array}$$

$$\begin{array}{lcl} D\Sigma_{(\sigma A \to B)} & : & \mathsf{func}\ A\ B \\ D\Sigma_\sigma & \mapsto & \lambda b.\ \Sigma\ \sigma^{-1}\ b\ \lambda a.\ \mathsf{var}\ a \end{array} \qquad \begin{array}{lcl} D\Pi_{(\sigma A \to B)} & : & \mathsf{func}\ A\ B \\ D\Pi_\sigma & \mapsto & \lambda b.\ \Pi\ \sigma^{-1}\ b\ \lambda a.\ \mathsf{var}\ a \end{array}$$

It is straightforward to check that these descriptions are interpreted to the expected operation on slices of $\textsc{Set}$, *i.e.* that we have:

$$[\![D\Delta_\sigma]\!] \cong \Delta_\sigma \qquad [\![D\Sigma_\sigma]\!] \cong \Sigma_\sigma \qquad [\![D\Pi_\sigma]\!] \cong \Pi_\sigma$$

□

5.69 Lemma. Described functors are closed under composition.

Proof. We define composition of descriptions by:

$$\begin{array}{lclclcl} (D : \mathsf{func}\ B\ C) & \circ_D & (E : \mathsf{func}\ A\ B) & : & \mathsf{func}\ A\ C \\ D & \circ_D & E & \mapsto & \lambda c.\ \mathsf{compose}\ (D\ c)\ E \end{array}$$

Which relies on the Kleisli extension:

$$
\begin{array}{llll}
\mathsf{compose}\,(D\!:\!\mathsf{IDesc}\,B)\,(E\!:\!\mathsf{func}\,B\,A) & & : & \mathsf{IDesc}\,A \\
\mathsf{compose}\quad (\mathsf{var}\,b) & E & \mapsto & E\,b \\
\mathsf{compose}\quad 1 & E & \mapsto & 1 \\
\mathsf{compose}\quad (A \times B) & E & \mapsto & \mathsf{compose}\,A\,E \times \mathsf{compose}\,B\,E \\
\mathsf{compose}\quad (\sigma\,En\,T) & E & \mapsto & \sigma\,En\,\lambda e.\,\mathsf{compose}\,(\mathsf{switch}\,T\,e)\,E \\
\mathsf{compose}\quad (\Pi\,S\,T) & E & \mapsto & \Pi\,S\,\lambda s.\,\mathsf{compose}\,(T\,s)\,E \\
\mathsf{compose}\quad (\Sigma\,S\,T) & E & \mapsto & \Sigma\,S\,\lambda s.\,\mathsf{compose}\,(T\,s)\,E
\end{array}
$$

It is then straightforward to check that this is indeed computing the composition of the functors, *i.e.* that we have:

$$[\![D \circ_D E]\!] \cong [\![D]\!] \circ [\![E]\!]$$

□

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 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.

□

(5.72) The benefit of this algebraic approach is its flexibility with respect to the universe definition: for practical purposes, we are likely to introduce new 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: `Chapter5.Equivalence.ToContainer`
`Chapter5.Equivalence.ToDesc`

(5.73) 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, *i.e.* operation, arity, and/or sorts. Crucially, the 1 and Σ codes contribute only to the operation. The var and Π codes, on the other hand, contribute to the arity. Finally, the 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:

$$\begin{array}{lcl} \langle(\sigma : \mathsf{ICont}\, I\, J)\rangle^{-1} & : & \mathsf{func}\, I\, J \\ \langle \mathsf{Op} \lhd^{\mathsf{Sort}} \mathsf{Ar}\rangle^{-1} & \mapsto & \Sigma\, \mathsf{Op}\, \lambda op.\, \Pi\, (\mathsf{Ar}\, op)\, \lambda ar.\, \mathsf{var}\, (\mathsf{Sort}\, ar) \end{array}$$

(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).

$$
\begin{array}{l}
\langle (D : \mathsf{func}\, I\, J) \rangle \;:\; \mathsf{ICont}\, I\, J \\
\langle D \rangle \;\mapsto\; \lambda j.\, \mathsf{Op}_{\mathsf{func}}\, (D\, j) \lhd^{\lambda j.\, \mathsf{Sort}_{\mathsf{func}}\, (D\, j)} \lambda j.\, \mathsf{Ar}_{\mathsf{func}}\, (D\, j) \qquad \textbf{where}
\end{array}
$$

$$
\begin{array}{llll}
\mathsf{Op}_{\mathsf{func}}\, (D : \mathsf{IDesc}\, I) & & : & \textsc{Set} \\
\mathsf{Op}_{\mathsf{func}} & \mathsf{var}\, i & \mapsto & \mathbb{1} \\
\mathsf{Op}_{\mathsf{func}} & 1 & \mapsto & \mathbb{1} \\
\mathsf{Op}_{\mathsf{func}} & A \times B & \mapsto & \mathsf{Op}_{\mathsf{func}}\, A \times \mathsf{Op}_{\mathsf{func}}\, B \\
\mathsf{Op}_{\mathsf{func}} & \sigma\, E\, T & \mapsto & (e : \mathsf{EnumT}\, E) \times \mathsf{Op}_{\mathsf{func}}\, (\mathsf{switch}\, T\, e) \\
\mathsf{Op}_{\mathsf{func}} & \Pi\, S\, T & \mapsto & (s : S) \to \mathsf{Op}_{\mathsf{func}}\, (T\, s) \\
\mathsf{Op}_{\mathsf{func}} & \Sigma\, S\, T & \mapsto & (s : S) \times \mathsf{Op}_{\mathsf{func}}\, (T\, s)
\end{array}
$$

$$
\begin{array}{lllll}
\mathsf{Ar}_{\mathsf{func}}\, (D : \mathsf{IDesc}\, I)\, (op : \mathsf{Op}_{\mathsf{func}}\, D) & & & : & \textsc{Set} \\
\mathsf{Ar}_{\mathsf{func}} & \mathsf{var}\, i & * & \mapsto & \mathbb{1} \\
\mathsf{Ar}_{\mathsf{func}} & 1 & * & \mapsto & \mathbb{0} \\
\mathsf{Ar}_{\mathsf{func}} & A \times B & (a, b) & \mapsto & \mathsf{Ar}_{\mathsf{func}}\, A\, a + \mathsf{Ar}_{\mathsf{func}}\, B\, b \\
\mathsf{Ar}_{\mathsf{func}} & \sigma\, E\, T & (e, t) & \mapsto & \mathsf{Ar}_{\mathsf{func}}\, (\mathsf{switch}\, T\, e)\, t \\
\mathsf{Ar}_{\mathsf{func}} & \Pi\, S\, T & f & \mapsto & (s : S) \times \mathsf{Ar}_{\mathsf{func}}\, (T\, s)\, (f\, s) \\
\mathsf{Ar}_{\mathsf{func}} & \Sigma\, S\, T & (s, t) & \mapsto & \mathsf{Ar}_{\mathsf{func}}\, (T\, s)\, t
\end{array}
$$

$$
\begin{array}{lllll}
\mathsf{Sort}_{\mathsf{func}}\, (D : \mathsf{IDesc}\, I)\, (ar : \mathsf{Ar}_{\mathsf{func}}\, D\, op) & & & : & I \\
\mathsf{Sort}_{\mathsf{func}} & \mathsf{var}\, i & * & \mapsto & i \\
\mathsf{Sort}_{\mathsf{func}} & A \times B & (\mathsf{inj}_l\, ar_a) & \mapsto & \mathsf{Sort}_{\mathsf{func}}\, A\, ar_a \\
\mathsf{Sort}_{\mathsf{func}} & A \times B & (\mathsf{inj}_r\, ar_b) & \mapsto & \mathsf{Sort}_{\mathsf{func}}\, B\, ar_b \\
\mathsf{Sort}_{\mathsf{func}} & \sigma\, E\, T & ar & \mapsto & \mathsf{Sort}_{\mathsf{func}}\, (\mathsf{switch}\, T\, (\pi_0\, sh))\, ar \\
\mathsf{Sort}_{\mathsf{func}} & \Pi\, S\, T & (s, ar) & \mapsto & \mathsf{Sort}_{\mathsf{func}}\, (T\, s)\, ar \\
\mathsf{Sort}_{\mathsf{func}} & \Sigma\, S\, T & ar & \mapsto & \mathsf{Sort}_{\mathsf{func}}\, (T\, (\pi_0\, sh))\, ar
\end{array}
$$

Figure 5.4.: From descriptions to containers

Conclusion

(5.77) Following the previous chapter on inductive types, we moved to inductive families. We have extended our type theory with a universe of indexed descriptions, 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

(5.78) 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.

(5.79) 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) In this chapter, we show that enumerations and descriptions can be treated as first-class datatypes (Section 6.1). Indeed, the type EnumU, Desc, and IDesc are nothing but inductives types! Our plan is to code them with descriptions and thus inherit the standard equipment of datatypes: constructor expressions and an induction principle.

We are then able to manipulate these types just like any other datatype. In particular, the generic induction principle applies to them and enables programming over descriptions. This leads to a system where datatype-generic programming, *i.e.* programming over descriptions, is just like programming over any datatype: no extension to the type theory is required. We shall give a few examples of generic programs – such as the catamorphism – and some generic datatype constructions – such as the free monad and its Kleisli category (Section 6.2). In mathematical terms, “generic programming” reads as *reflection*: we reflect the meta-theory of inductive types within the type theory. Consequently, we can develop the theory of inductive types from inside our logic, without any change to the meta-theory.

However, this exercise in self-description is perilously paradoxical. We shall therefore conclude by giving a model of our system (Section 6.3). We build a model of a stratified hierarchy of types à la Palmgren [1995] and prove an isomorphism between the meta-theoretic objects and their reflection. We shall explain in which sense our type theory merely collapses this isomorphism to offer a first-class notion of datatypes.

6.1. The Art of Levitation

(6.2) 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`

(6.3) In Section 2.2, we specified the universe of enumerations EnumU. The formation and introduction rules were given as:

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathsf{EnumU} : \textsc{Set}}$$

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathsf{nilE} : \mathsf{EnumU}} \qquad \frac{\Gamma \vdash t : \mathsf{UId} \quad \Gamma \vdash E : \mathsf{EnumU}}{\Gamma \vdash \mathsf{consE}\ t\ E : \mathsf{EnumU}}$$

However, we remark that the nilE and consE codes are respectively the nil and cons constructors of an ordinary list with elements of type 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 UId, we reflect EnumU as an inductive type in the type theory.

(6.4) Following this intuition, we define:

REFLECTION

$$\begin{array}{lcl}\mathsf{EnumU} & : & \textsc{Set} \\ \mathsf{EnumU} & \mapsto & \mu\left(\Sigma\,\mathsf{EnumT}\begin{Bmatrix}\text{'}\mathsf{nilE} \\ \text{'}\mathsf{consE}\end{Bmatrix}\begin{Bmatrix}\text{'}\mathsf{nilE} & \mapsto 1 \\ \text{'}\mathsf{consE} & \mapsto \Sigma\,\mathsf{UId}\,\lambda - .\,\mathsf{var}\times 1\end{Bmatrix}\right)\end{array}$$

By relying on the constructor expressions (4.67), we can type check nilE and consE against EnumU. These expressions elaborate to the low-level codes:

$$\vdash \mathsf{EnumU} \ni \mathsf{nilE} \overset{Chk}{\leadsto} [\text{'}\mathsf{nilE}]$$

$$t : \mathsf{UId}; E : \mathsf{EnumU} \vdash \mathsf{EnumU} \ni \mathsf{consE}\ t\ E \overset{Chk}{\leadsto} [\text{'}\mathsf{consE}\ t\ E]$$

(6.5) We have thus defined EnumU as a description. Our code clearly satisfies the specification we promised to realise. We are therefore tempted to substitute the meta-level EnumU with this definition. However, this raises two questions. First, we must make sure that this move is *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 *term*. So far, we have given a high-level *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: EnumU is an admissible type.

6.6 Theorem (Admissibility of EnumU)**.** There exists a term EnumUD whose fixpoint satisfies the specification of EnumU, *i.e.* we have:

$$\begin{gathered}\Gamma \vdash \textsc{valid} \Rightarrow \Gamma \vdash \mu\,\mathsf{EnumUD} : \textsc{Set} \\ \Gamma \vdash \textsc{valid} \Rightarrow \Gamma \vdash \mathsf{in}\,(\mathsf{'nilE}, *) : \mu\,\mathsf{EnumUD} \\ \Gamma \vdash t : \mathsf{UId} \wedge \Gamma \vdash e : \mu\,\mathsf{EnumUD} \Rightarrow \Gamma \vdash \mathsf{in}\,(\mathsf{'consE}, (t, (e, *))) : \mu\,\mathsf{EnumUD}\end{gathered}$$

and these are the only inhabitants of that type, encoding respectively nilE and consE.

Proof. Without surprise, we derive the code of EnumUD by type checking our (putative) definition of EnumU (¶ 6.4):

$$\vdash \textsc{Set} \ni \mathsf{EnumU} \overset{Chk}{\leadsto} \mu\,\mathsf{EnumUD}$$

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:

$$\frac{\vdash \mathsf{consE}\,\mathsf{'nilE}\,(\mathsf{consE}\,\mathsf{'consE}\,\mathsf{nilE}) \ni \mathsf{EnumU} \overset{Chk}{\leadsto} t}{\vdash \textsc{Set} \ni \mathsf{EnumU} \overset{Chk}{\leadsto} \mu\,(\Sigma\,(\mathsf{EnumT}\,t)\,(\mathsf{switch}(1, (\Sigma\,\mathsf{UId}\,\lambda - .\,\mathsf{var} \times 1, *))))}$$

Note that the enumeration of constructors relies on the constructors nilE and consE, which do not formally exist! However, recall that EnumT has type $\mathsf{EnumU} \to \textsc{Set}$: the enumeration $\mathsf{consE}\,\mathsf{'nilE}\,(\mathsf{consE}\,\mathsf{'consE}\,\mathsf{nilE})$ is thus *checked* against EnumU. We are back to an elaboration problem:

$$\vdash \mathsf{EnumU} \ni \mathsf{consE}\,\mathsf{'nilE}\,(\mathsf{consE}\,\mathsf{'consE}\,\mathsf{nilE}) \overset{Chk}{\leadsto} ?$$

As pointed out earlier (¶ 6.4), we have access to the constructor expressions for EnumU: the elaboration problem above is thus simply a succession of 3 such elaboration problems. After elaboration, we obtain the following term:

$$\begin{array}{l}\vdash \mathsf{EnumU} \ni \mathsf{consE}\,'\mathsf{nilE}\,(\mathsf{consE}\,'\mathsf{consE}\,\mathsf{nilE}) \overset{\mathit{Chk}}{\leadsto} \\ \qquad \mathsf{in}\,('\mathsf{consE}, (\underline{'\mathsf{nilE}}, \mathsf{in}\,('\mathsf{consE}, (\underline{'\mathsf{consE}}, \mathsf{in}\,('\mathsf{nilE}, *)))))\end{array}$$

For convenience, we underline the tag belonging to the collection. The others come from the elaboration of the 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:

$$\begin{array}{l}\vdash \textsc{Set} \ni \mathsf{EnumU} \overset{\mathit{Chk}}{\leadsto} \mu\, \Sigma\, (\mathsf{EnumT}\,(\mathsf{in}\,('\mathsf{consE}, (\underline{'\mathsf{nilE}}, \mathsf{in}\,('\mathsf{consE}, (\underline{'\mathsf{consE}}, \mathsf{in}\,('\mathsf{nilE}, *))))))) \\ \qquad\qquad (\mathsf{switch}\,(1, (\Sigma\, \mathsf{UId}\, \lambda - .\, \mathsf{var} \times 1, *)))\end{array}$$

We therefore take EnumUD to be

$$\begin{array}{rl}\mathsf{EnumUD} \triangleq & \Sigma\, (\mathsf{EnumT}\,(\mathsf{in}\,('\mathsf{consE}, (\underline{'\mathsf{nilE}}, \mathsf{in}\,('\mathsf{consE}, (\underline{'\mathsf{consE}}, \mathsf{in}\,('\mathsf{nilE}, *))))))) \\ & (\mathsf{switch}\,(1, (\Sigma\, \mathsf{UId}\, \lambda - .\, \mathsf{var} \times 1, *)))\end{array}$$

and *define* EnumU by

$$\begin{array}{lcl}\mathsf{EnumU} & : & \textsc{Set}_l \\ \mathsf{EnumU} & \mapsto & \mu\, \mathsf{EnumUD}\end{array}$$

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.

□

(6.7) 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, $E : \mathsf{EnumU}$ is just like *any* datatype now.

Object	Role	Status
EnumU,nilE, consE	Build enumerations	Reflected
π	Build finite function space	Reflected
EnumT, 0, 1+	Index into enumerations	Meta-theory
switch	Eliminate indices	Meta-theory

Table 6.1.: Status of enumerations

6.8 Definition (Reflected π)**.** We therefore *define* π by induction over E:

REFLECTION

$$\begin{array}{lllll}
\pi & (E : \mathsf{EnumU}) & (P : \mathsf{EnumT}\, E \to \mathrm{SET}) & : & \mathrm{SET} \\
\pi & \quad E & \quad P & \Leftarrow & \mathsf{induction}\, E \\
\quad \pi & \quad \mathsf{nilE} & \quad P & \mapsto & \mathbb{1} \\
\quad \pi & (\mathsf{consE}\, t\, E) & \quad P & \mapsto & P\, 0 \times \pi\, E\, (P \circ 1{+})
\end{array}$$

6.9 Remark (Resulting type theory)**.** Our construction shows that, for a type theory with a universe Desc, we can spare ourselves from extending the meta-theory with EnumU, its constructors nilE and consE, and π. Indeed, we can simply *define* them by a description (for EnumU), by induction over the resulting datatype (for π), 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 π 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.

(6.10) 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`

(6.11) 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

$$\begin{array}{l} \mathsf{T}^{\ell} \;:\; \mathrm{SET}_{\ell} \\ \mathsf{T}^{\ell} \mapsto \cdots \end{array}$$

to define the type T simultaneously at every level of the hierarchy of sets. Similarly, we write

$$\begin{array}{l} \mathsf{t}^{\ell} \;:\; \mathsf{T}^{\ell} \\ \mathsf{t}^{\ell} \mapsto \cdots \end{array}$$

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.

(6.13) 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.

(6.14) The first step is to code the choice of constructors. As usual by now, we represent the choice of constructors by an enumeration:

$$\begin{array}{l}
\mathsf{DescD}_{\ell} \;:\; \mathsf{Desc}_{\ell+1} \\
\mathsf{DescD}_{\ell} \mapsto \Sigma\,\mathsf{EnumT}\left\{\begin{array}{c} '1 \\ '\mathsf{var} \\ '\times \\ '\Sigma \\ '\Pi \end{array}\right\}\left\{\begin{array}{ll} '1 & \mapsto \{?\} \\ '\mathsf{var} & \mapsto \{?\} \\ '\times & \mapsto \{?\} \\ '\Sigma & \mapsto \{?\} \\ '\Pi & \mapsto \{?\} \end{array}\right\}
\end{array}$$

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:

$$\begin{array}{l}
\mathsf{DescD}_{\ell} \;:\; \mathsf{Desc}_{\ell+1} \\
\mathsf{DescD}_{\ell} \mapsto \Sigma\,\mathsf{EnumT}\left\{\begin{array}{c} '1 \\ '\mathsf{var} \\ '\times \\ '\Sigma \\ '\Pi \end{array}\right\}\left\{\begin{array}{ll} '1 & \mapsto 1 \\ '\mathsf{var} & \mapsto 1 \\ '\times & \mapsto \{?\} \\ '\Sigma & \mapsto \{?\} \\ '\Pi & \mapsto \{?\} \end{array}\right\}
\end{array}$$

(6.16) The next step is to encode the binary product (×), which takes two recursive arguments described by var:

$$\begin{array}{l}
\mathsf{DescD}_{\ell} \;:\; \mathsf{Desc}_{\ell+1} \\
\mathsf{DescD}_{\ell} \mapsto \Sigma\,\mathsf{EnumT}\left\{\begin{array}{c} '1 \\ '\mathsf{var} \\ '\times \\ '\Sigma \\ '\Pi \end{array}\right\}\left\{\begin{array}{ll} '1 & \mapsto 1 \\ '\mathsf{var} & \mapsto 1 \\ '\times & \mapsto \mathsf{var} \times \mathsf{var} \times 1 \\ '\Sigma & \mapsto \{?\} \\ '\Pi & \mapsto \{?\} \end{array}\right\}
\end{array}$$

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 Σ and Π. To do so, we use a Σ code to introduce an S of type $\textsc{Set}_\ell$ and then a Π to code the exponential by S:

REFLECTION

$$
\begin{array}{lcl}
\mathsf{DescD}_\ell & : & \mathsf{Desc}_{\ell+1} \\
\mathsf{DescD}_\ell & \mapsto & \Sigma\,\mathsf{EnumT} \left\{ \begin{array}{c} \text{'}1 \\ \text{'}\mathsf{var} \\ \text{'}\times \\ \text{'}\Sigma \\ \text{'}\Pi \end{array} \right\} \left\{ \begin{array}{ll} \text{'}1 & \mapsto 1 \\ \text{'}\mathsf{var} & \mapsto 1 \\ \text{'}\times & \mapsto \mathsf{var} \times \mathsf{var} \times 1 \\ \text{'}\Sigma & \mapsto \Sigma\,\textsc{Set}_\ell\,\lambda S.\,(\Pi\,S\,\lambda - .\,\mathsf{var}) \times 1 \\ \text{'}\Pi & \mapsto \Sigma\,\textsc{Set}_\ell\,\lambda S.\,(\Pi\,S\,\lambda - .\,\mathsf{var}) \times 1 \end{array} \right\}
\end{array}
$$

Following Remark 6.17, we terminate the codes with a 1 code.

(6.19) At first glance, we have achieved our objective. We have described the codes of the universe of descriptions. We are naturally lead to *define* Desc by taking its fixpoint:

REFLECTION

$$
\begin{array}{lcl}
\mathsf{Desc}_\ell & : & \textsc{Set}_{\ell+1} \\
\mathsf{Desc}_\ell & \mapsto & \mu\,\mathsf{DescD}_\ell
\end{array}
$$

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.

(6.20) The definition $\mathsf{Desc}_\ell \mapsto \mu\,\mathsf{DescD}_\ell$ 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 switch reveals the issue:

$$
\begin{array}{lcl}
\mathsf{DescD}_\ell & : & \mathsf{Desc}_{\ell+1} \\
\mathsf{DescD}_\ell & \mapsto & \Sigma\,\mathsf{EnumT}\,\{\text{'}1\,\text{'}\mathsf{var}\,\text{'}\times\,\text{'}\Sigma\,\text{'}\Pi\} \\
 & & \qquad (\mathsf{switch}\,(P := \lambda - .\,\mu\,\mathsf{DescD}_{\ell+1}) \begin{bmatrix} 1 \\ 1 \\ \mathsf{var} \times \mathsf{var} \times 1 \\ \Sigma\,\textsc{Set}_\ell\,\lambda S.\,(\Pi\,S\,\lambda - .\,\mathsf{var}) \times 1 \\ \Sigma\,\textsc{Set}_\ell\,\lambda S.\,(\Pi\,S\,\lambda - .\,\mathsf{var}) \times 1 \end{bmatrix})
\end{array}
$$

The circularity arises from the fact that we must specify the return type of the general-purpose switch eliminator: we claim to return a $\mu\,\mathsf{DescD}_{\ell+1}$ 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 Desc from its own definition.

6.21 Remark (Attempting to prove the admissibility of Desc)**.** In the previous section, we have shown that it is correct to introduce a self-hosted EnumU by giving a low-level term satisfying its specification. It is instructive to attempt a similar construction by elaborating our tentative definition of 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 *choice* and the tuple of codes elaborates to a term *codes*:

$$\vdash \mathsf{EnumU} \ni \{\text{'}1\,\text{'}\mathsf{var}\,\text{'}\times\text{'}\Sigma\,\text{'}\Pi\} \overset{\mathit{Chk}}{\leadsto} \mathit{choice}$$

$$\vdash \mathsf{EnumT}\,\mathit{choice} \to \mathsf{Desc} \ni \begin{bmatrix} 1 \\ 1 \\ \mathsf{var} \times \mathsf{var} \times 1 \\ \Sigma\,\textsc{Set}_\ell\,\lambda S.\,(\Pi\,S\,\lambda - .\,\mathsf{var}) \times 1 \\ \Sigma\,\textsc{Set}_\ell\,\lambda S.\,(\Pi\,S\,\lambda - .\,\mathsf{var}) \times 1 \end{bmatrix} \overset{\mathit{Chk}}{\leadsto} \mathit{codes}$$

We could therefore be tempted to define DescD with the term:

$$\begin{array}{lcl} \mathsf{DescD}_\ell & : & \mathsf{Desc}_{\ell+1} \\ \mathsf{DescD}_\ell & \mapsto & \Sigma\,(\mathsf{EnumT}\,\mathit{choice}) \\ & & \quad (\mathsf{switch}\,(P := \lambda - .\,\mu\,\mathsf{DescD}_{\ell+1})\,\mathit{codes}) \end{array}$$

This definition is invalid! The definition of DescD depends on DescD itself. Adding this term to our type theory would break strong normalisation and its corollary: decidability of type checking.

(6.22) Thus, the problem stands solely in the fact that we must supply a return type for switch. There is an easy way around that: since we know what this return type *must* be, we eliminate the dreaded circularity by *specialising* switch:

META-THEORY

$$\begin{aligned}
&\mathsf{switch}_{\mathsf{D}}\ (ps : \pi\ E\ (\lambda - .\ \mathsf{Desc}_{\ell}))\ (x : \mathsf{EnumT}\ E)\ :\ \mathsf{Desc}_{\ell}\\
&\mathsf{switch}_{\mathsf{D}}\ b\quad 0\qquad \mapsto \pi_0\ b\\
&\mathsf{switch}_{\mathsf{D}}\ b\ (1\!+x)\ \mapsto \mathsf{switch}_{\mathsf{D}}\ (\pi_1\ b)\ x
\end{aligned}$$

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 $\mathsf{Desc}_{\ell+1} : \mathrm{SET}_{\ell+2}$ to describe DescD_{ℓ}, with the intent of defining the object $\mathsf{Desc}_{\ell} : \mathrm{SET}_{\ell+1}$ next. This is the reason why we could not provide a motive to switch: we would need to refer to Desc_{ℓ} while it is not yet defined. By specialising switch to $\mathsf{switch}_{\mathsf{D}}$, we free ourselves from the need to *justify* the existence of a type Desc_{ℓ} (more precisely, in this case, of a tuple of said type). Freed from that necessity, we can define DescD_{ℓ} while avoiding any mention of Desc_{ℓ} – *i.e.* a circularity – altogether.

6.24 Remark. The reader might be concerned about the fact that the *type* of $\mathsf{switch}_{\mathsf{D}}$ mentions Desc. Recall that we must avoid a circular definition of DescD: as long as the *term* defining DescD does not require Desc to pre-exist, we are in a safe territory. A similar situation arose when describing EnumU: the type of EnumT depends on EnumU and EnumT is used to define EnumU.

6.25 Definition (Reflecting descriptions)**.** By replacing switch with a specialised $\mathsf{switch}_{\mathsf{D}}$, we do not need to mention the return type: it is hard-coded in the definition of $\mathsf{switch}_{\mathsf{D}}$. This definition is therefore non-circular:

$$\begin{aligned}
&\mathsf{DescD}_{\ell}\ \ :\ \ \mathsf{Desc}_{\ell+1}\\
&\mathsf{DescD}_{\ell}\ \mapsto\ \Sigma\ \mathsf{EnumT}\ \{\text{'}1\ \text{'}\mathsf{var}\ \text{'}\times\ \text{'}\Sigma\ \text{'}\Pi\}\\
&\qquad\qquad (\mathsf{switch}_{\mathsf{D}}\ \begin{bmatrix} 1\\ 1\\ \mathsf{var}\times\mathsf{var}\times 1\\ \Sigma\,\mathrm{SET}_{\ell}\ \lambda S.\ (\Pi\ S\ \lambda - .\ \mathsf{var})\times 1\\ \Sigma\,\mathrm{SET}_{\ell}\ \lambda S.\ (\Pi\ S\ \lambda - .\ \mathsf{var})\times 1 \end{bmatrix})
\end{aligned}$$

(6.26) We conceal $\mathsf{switch}_\mathsf{D}$ behind a type propagation rule for the finite function spaces of codomain Desc that we apply with higher priority than switch:

ELABORATION

$$\frac{\Gamma \vdash \mathsf{EnumU} \ni \{'l_0 \;\ldots\; 'l_k\} \overset{Chk}{\rightsquigarrow} E \qquad \Gamma \vdash \pi\, E\, (\lambda - .\, \mathsf{Desc}_\ell) \ni [e_0 \ldots e_k] \overset{Chk}{\rightsquigarrow} ts}{\Gamma \vdash \mathsf{EnumT}\, E \to \mathsf{Desc}_\ell \ni \{'l_0 \mapsto e_0\,;\, \ldots\,;\, 'l_k \mapsto e_k\} \overset{Chk}{\rightsquigarrow} \mathsf{switch}_\mathsf{D}\, ts}$$

Our tentative definition (¶ 6.18) now elaborates to the desired, paradox-free term.

(6.27) Thanks to this elaboration rule and the special-purpose $\mathsf{switch}_\mathsf{D}$, 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 Desc. Then, we check that this term is free of self-reference.

6.28 Theorem (Admissibility of DescD)**.** There exists a term DescD whose fixpoint satisfies the specification of Desc, *i.e.* we have:

$$\begin{aligned}
\Gamma \vdash \textsc{valid} &\Rightarrow \Gamma \vdash \mu\, \mathsf{DescD}_\ell : \textsc{Set}_{\ell+1} \\
\Gamma \vdash \textsc{valid} &\Rightarrow \Gamma \vdash \mathsf{in}\, ('1, *) : \mu\, \mathsf{DescD}_\ell \\
\Gamma \vdash \textsc{valid} &\Rightarrow \Gamma \vdash \mathsf{in}\, ('\mathsf{var}, *) : \mu\, \mathsf{DescD}_\ell \\
\Gamma \vdash A : \mu\, \mathsf{DescD}_\ell \wedge \Gamma \vdash B : \mu\, \mathsf{DescD}_\ell &\Rightarrow \Gamma \vdash \mathsf{in}\, ('\times, (A, (B, *))) : \mu\, \mathsf{DescD}_\ell \\
\Gamma \vdash S : \textsc{Set}_l \wedge \Gamma \vdash T : S \to \mu\, \mathsf{DescD}_\ell &\Rightarrow \Gamma \vdash \mathsf{in}\, ('\Sigma, (S, (T, *))) : \mu\, \mathsf{DescD}_\ell \\
\Gamma \vdash S : \textsc{Set}_l \wedge \Gamma \vdash T : S \to \mu\, \mathsf{DescD}_\ell &\Rightarrow \Gamma \vdash \mathsf{in}\, ('\Pi, (S, (T, *))) : \mu\, \mathsf{DescD}_\ell
\end{aligned}$$

and these are the only inhabitants of that type, encoding 1, var, $\times$, Σ, and Π.

Proof. The construction of the term DescD 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 $\mathsf{switch}_\mathsf{D}$ term. Unlike our previous attempt (Remark 6.21), the resulting term does not mention DescD in the motive since we have removed it. Finally, just as EnumU uses its constructors in its own definition, Desc uses the Desc constructors to describe itself: these constructors are interpreted as constructor expressions and elaborate to the low-level terms:

$$\begin{array}{r}
\vdash \mathsf{Desc}_\ell \ni 1 \overset{\mathit{Chk}}{\leadsto} \mathsf{in}\,('1, \ast) \\
\vdash \mathsf{Desc}_\ell \ni \mathsf{var} \overset{\mathit{Chk}}{\leadsto} \mathsf{in}\,('\mathsf{var}, \ast) \\
A\!:\!\mathsf{Desc}_\ell; B\!:\!\mathsf{Desc}_\ell \vdash \mathsf{Desc}_\ell \ni A \times B \overset{\mathit{Chk}}{\leadsto} \mathsf{in}\,('\times, (A, (B, \ast))) \\
S\!:\!\textsc{Set}_\ell; T\!:\!S \to \mathsf{Desc}_\ell \vdash \mathsf{Desc}_\ell \ni \Sigma\, S\, T \overset{\mathit{Chk}}{\leadsto} \mathsf{in}\,('\Sigma, (S, (T, \ast))) \\
S\!:\!\textsc{Set}_\ell; T\!:\!S \to \mathsf{Desc}_\ell \vdash \mathsf{Desc}_\ell \ni \Pi\, S\, T \overset{\mathit{Chk}}{\leadsto} \mathsf{in}\,('\Pi, (S, (T, \ast)))
\end{array}$$

Putting all the pieces together, we obtain a fully-elaborated term:

$$\begin{array}{l}
\mathsf{DescD}_\ell \triangleq (('\Sigma,((\mathsf{EnumT}\ ('\mathsf{consE},('1,(('\mathsf{consE},('\mathsf{var},(('\mathsf{consE},('\times \\
\qquad\qquad\qquad\qquad ,(('\mathsf{consE},('\Sigma,(('\mathsf{consE},('\Pi,(('\mathsf{nilE},\ast),\ast))),\ast))),\ast))),\ast))),\ast)))) \\
\qquad\quad ,((\mathsf{switch}_{\mathsf{D}}\ (('1,\ast), \\
\qquad\qquad\qquad (('1,\ast), \\
\qquad\qquad\qquad (('\times,(('\mathsf{var},\ast),(('\times,(('\mathsf{var},\ast),(('1,\ast),\ast))),\ast))), \\
\qquad\qquad\qquad (('\Sigma,(\textsc{Set}_\ell,(\lambda S.\ ('\times,((('\Pi,(S,(\lambda -.\ ('\mathsf{var},\ast),\ast)))),(('1,\ast),\ast))),\ast))), \\
\qquad\qquad\qquad (('\Sigma,(\textsc{Set}_\ell,(\lambda S.\ ('\times,((('\Pi,(S,(\lambda -.\ ('\mathsf{var},\ast),\ast)))),(('1,\ast),\ast))),\ast))), \\
\qquad\qquad\qquad \ast)))))),\ast))))
\end{array}$$

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 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* Desc to be the fixpoint of that low-level term:

$$\begin{array}{lcl}
\mathsf{Desc}_\ell & : & \textsc{Set}_{\ell+1} \\
\mathsf{Desc}_\ell & \mapsto & \mu\, \mathsf{DescD}_\ell
\end{array}$$

Under this *definition*, we can now formally type check our high-level definition of 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 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 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 Desc constructors (Definition 4.6): we encode them at the term level and we use

Object	Role	Status
Desc	Describe signature functors	Reflected
$[\![-]\!]$	Interpret descriptions	Meta-theory
μ, in	Define, inhabit fixpoints	Meta-theory
induction, $\Box_{-}$, $\Box_{-}^{\rightarrow}$	Induction principle	Meta-theory

Table 6.2.: Summary of constructions on descriptions

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 μ (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 Desc itself: by realising that it already implicitly exists, we derive its raw, non-elaborated form (Theorem 6.28) and denote that term "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 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 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 $\mathsf{IDesc}\ I$ is a plain inductive type, parameterised by I. Therefore, we could simply describe IDesc with a Desc code, as we coded 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

$$\mathsf{switch}_{\mathsf{ID}}\ (ps : \pi\ E\ (\lambda - .\ \mathsf{IDesc}_\ell\ I))(x : \mathsf{EnumT}\ E) : \mathsf{IDesc}\ \ell I$$

Having a specialised $\mathsf{switch}_{\mathsf{ID}}$, we can safely eliminate over the enumeration of constructors and code each description constructor.

In fact, the only requirement is to avoid any mention of IDesc in the definition of its code and interpretation. Alternatively, we could code the choice of constructors choice with a *finite* sum σ:

$$\begin{array}{l}
\mathsf{IDescD}_\ell\ (I : \textsc{Set})\ :\ \mathsf{IDesc}_{\ell+1}\ \mathbb{1} \\
\mathsf{IDescD}_\ell\ I\ \mapsto\ \sigma \begin{Bmatrix} '1 \\ '\mathsf{var} \\ '\times \\ '\sigma \\ '\Sigma \\ '\Pi \end{Bmatrix} \begin{bmatrix} 1 \\ \Sigma\ I\ \lambda - .\ 1 \\ \mathsf{var} * \times \mathsf{var} * \times 1 \\ \Sigma\ \mathsf{EnumU}\ \lambda E.\ \pi\ E \\ \Sigma\ \textsc{Set}_\ell\ \lambda S.\ (\Pi\ S\ \lambda - .\ \mathsf{var} *) \times 1 \\ \Sigma\ \textsc{Set}_\ell\ \lambda S.\ (\Pi\ S\ \lambda - .\ \mathsf{var} *) \times 1 \end{bmatrix}
\end{array}$$

Doing so, we hoist switch from the description of IDesc to the interpretation function of IDesc. It is the interpretation function that must then use the specialised $\mathsf{switch}_{\mathsf{ID}}$ elimination principle, as follows:

$$\begin{array}{lll}
\multicolumn{3}{l}{[\![(D : \mathsf{IDesc}_\ell\ I)]\!]\ (X : I \to \textsc{Set}_\ell)\ :\ \textsc{Set}_\ell} \\
[\![\mathsf{var}\ i]\!] & X \mapsto & X\ i \\
[\![1]\!] & X \mapsto & \mathbb{1} \\
[\![A \times B]\!]\ X & \mapsto & [\![A]\!]\ X \times [\![B]\!]\ X \\
[\![\sigma\ E\ T]\!] & X \mapsto & (e : \mathsf{EnumT}\ E) \times [\![\mathsf{switch}_{\mathsf{ID}}\ T\ e]\!]\ X \\
[\![\Pi\ S\ T]\!] & X \mapsto & (s : S) \to [\![T\ s]\!]\ X \\
[\![\Sigma\ S\ T]\!] & X \mapsto & (s : S) \times [\![T\ s]\!]\ X
\end{array}$$

6.33 Remark. To define the code of the constructor σ, 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 $\mathsf{IDesc}\,(\mathsf{EnumT}\, E)$. 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:

$$\begin{array}{lcl}
\pi\,(E\!:\!\mathsf{EnumU}) & : & \mathsf{IDesc}_{\ell}\,(\mathsf{EnumT}\, E) \\
\pi \quad \mathsf{nilE} & \mapsto & 1 \\
\pi\ \ (\mathsf{consE}\, t\, E) & \mapsto & \mathsf{var}\, 0 \times (\mathsf{compose}\,(\pi\, E)\; D\Delta_{1+})
\end{array}$$

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 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 Desc is not restricted to the meta-language: we have the necessary equipment to program with *data*, *i.e.* program over datatypes. *Generic programming is just programming*.

(6.35) 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`

(6.36) 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 $(X, \alpha\!:\!F\, X \rightarrow X)$ be an F-algebra.

The *catamorphism* (or *fold*) is the unique F-algebra morphism $\llparenthesis\alpha\rrparenthesis$ from the initial algebra $(\mu F, in : F\,\mu F \to \mu F)$ to the algebra α. Put explicitly, $\llparenthesis\alpha\rrparenthesis$ is the unique morphism making the following diagram commute:

$$\begin{tikzcd}
F(\mu F) \arrow[r, dashed, "F\,\llparenthesis\alpha\rrparenthesis"] \arrow[d, "in"'] & FX \arrow[d, "\alpha"] \\
\mu F \arrow[r, dashed, "\llparenthesis\alpha\rrparenthesis"'] & X
\end{tikzcd}$$

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.

(6.39) 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 $\alpha : \llbracket D \rrbracket\; T \to T$, an inhabitant of the fixpoint, and returns a T:

$$\begin{array}{llll}
\llparenthesis(\alpha : \llbracket D \rrbracket\; T \to T)\rrparenthesis & (x : \mu\, D) & : & T \\
\llparenthesis\alpha\rrparenthesis & x & & \{?\}
\end{array}$$

(6.40) Operationally, the catamorphism applies the algebra recursively over the structure of the functor $\llbracket D \rrbracket$. To do so, we proceed by induction over x. Since the return type is non-dependent, setting up the induction is an easy step:

$$\begin{array}{llll}
\llparenthesis(\alpha : \llbracket D \rrbracket\; T \to T)\rrparenthesis & (x : \mu\, D) & : & T \\
\llparenthesis\alpha\rrparenthesis & x & \mapsto & \mathsf{induction}\ \{? : (xs : \llbracket D \rrbracket\; (\mu\, D))(ih : \Box_D\; (\lambda - .\, T)\; xs) \to T\}\ x
\end{array}$$

(6.41) We are then left to implement the inductive step. Given a node xs and the induction hypotheses ih, we must build an element of T. Provided that we know how to make an element of $\llbracket D \rrbracket\; T$, this step will be performed by the algebra α:

$$\begin{array}{l}
\llparenthesis(\alpha : \llbracket D \rrbracket\; T \to T)\rrparenthesis\ (x : \mu\, D)\ :\ T \\
\llparenthesis\alpha\rrparenthesis\ x\ \mapsto\ \mathsf{induction}\ (\lambda xs\; ih.\, \alpha\ \{? : \llbracket D \rrbracket\; T\}\ )\ x
\end{array}$$

(6.42) To complete the final hole, we have $xs : \llbracket D \rrbracket\; \mu\, D$ and $ih : \Box_D\; (\lambda - .\, T)\; xs$ to hand, and we need a $\llbracket D \rrbracket\; T$. The argument xs has the right shape, but its subcomponents have the wrong type. However, for each such component, ih holds the corresponding value in

T. We thus implement a function to replace the former with the latter

$$\begin{array}{llllcl}
\mathsf{replace}\ (D\!:\!\mathsf{Desc}) & (xs\!:\![\![D]\!]\ X) & (ih\!:\!\Box_D\ (\lambda - .\ Y)\ xs) & & : & [\![D]\!]\ Y \\
\mathsf{replace}\quad 1 & * & * & & \mapsto & * \\
\mathsf{replace}\quad \mathsf{var} & x & y & & \mapsto & y \\
\mathsf{replace}\quad (A \times B) & (a, b) & (x, y) & & \mapsto & (\mathsf{replace}\ A\ a\ x, \mathsf{replace}\ B\ b\ y) \\
\mathsf{replace}\quad (\Sigma\, S\, D) & (s, d) & d' & & \mapsto & (s, \mathsf{replace}\ (D\ s)\ d\ d') \\
\mathsf{replace}\quad (\Pi\, S\, D) & f & f' & & \mapsto & \lambda s.\ \mathsf{replace}\ (D\ s)\ (f\ s)\ (f'\ s)
\end{array}$$

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 $\mathsf{replace}\ D\ xs\ ih$, we have implemented a generic catamorphism function:

$$\begin{array}{llcl}
(\!|(\alpha\!:\![\![D]\!]\ T \to T)|\!)\ (x\!:\!\mu\, D) & & : & T \\
(\!|\alpha|\!) & x & \mapsto & \mathsf{induction}\ (\lambda xs\ ih.\ \alpha\ (\mathsf{replace}\ D\ xs\ ih))\ x
\end{array}$$

(6.45) **Indexed catamorphism.** The same construction can be carried on the universe of indexed descriptions, deriving an indexed catamorphism from the elimination principle of IDesc. We shall overload notation and denote $(\!|\ -\ |\!)$ this catamorphism.

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:

$$\begin{array}{llcl}
\mathsf{heightAlg}\ (ts\!:\![\![\mathsf{TreeD}\ A]\!]\ \mathsf{Nat}) & & : & \mathsf{Nat} \\
\mathsf{heightAlg} & [\text{'}\mathsf{leaf}] & \mapsto & 0 \\
\mathsf{heightAlg} & [\text{'}\mathsf{node}\ lh\ a\ rh] & \mapsto & \mathsf{suc}\ (\mathsf{max}\ lh\ rh)
\end{array}$$

By recursively iterating this algebra over the tree, the catamorphism gives us the desired function:

$$\begin{array}{llcl}
\mathsf{height}\ (t\!:\!\mathsf{Tree}\ A) & & : & \mathsf{Nat} \\
\mathsf{height} & t & \mapsto & (\!|\mathsf{heightAlg}|\!)\ t
\end{array}$$

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:

$$\begin{array}{lcl} \mathsf{eval}_{\Downarrow}\ (tm\!:\!\mu\,\mathsf{ExprD}\ ty) & : & \mathsf{Val}\ ty \\ \mathsf{eval}_{\Downarrow}\ \quad tm & \mapsto & (\!|\mathsf{eval}_{\downarrow}|\!)\ tm \end{array}$$

To finish the job, we must supply the algebra that implements a single step of evaluation, given already evaluated subexpressions.

$$\begin{array}{lllcl} \mathsf{eval}_{\downarrow} & (ty\!:\!\mathsf{Ty}) & (xs\!:\![\![\mathsf{ExprD}]\!]\ \mathsf{Val}) & : & \mathsf{Val}\ ty \\ \mathsf{eval}_{\downarrow} & - & [\mathsf{'val}\ x] & \mapsto & x \\ \mathsf{eval}_{\downarrow} & - & [\mathsf{'cond\ true}\ x\ -] & \mapsto & x \\ \mathsf{eval}_{\downarrow} & - & [\mathsf{'cond\ false}\ -\ y] & \mapsto & y \\ \mathsf{eval}_{\downarrow} & \mathsf{'nat} & [\mathsf{'plus}\ x\ y] & \mapsto & x + y \\ \mathsf{eval}_{\downarrow} & \mathsf{'bool} & [\mathsf{'le}\ x\ y] & \mapsto & x \leq y \end{array}$$

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.

(6.48) We have shown how to derive a generic operation, the catamorphism, from a preexisting generic operation, induction, by manipulating descriptions as data: the catamorphism is just a function taking each Desc value to a datatype specific operation. This is polytypic programming [Jansson and Jeuring, 1997] made ordinary.

6.2.2. The generic free monad

MODEL: `Chapter6.Desc.FreeMonad`

(6.49) 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.

(6.50) 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:

```
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 `Op` – and variables – introduced by `Ret`. Provided that `f` is an instance of `Functor`, we may take `Ret` for `return` and use `f`'s `fmap` to define the monadic bind `>>=` as substitution:

```
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 `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 `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:

$$\begin{array}{llll} (D\!:\!\mathsf{tagDesc})^{*} & (X\!:\!\mathrm{SET}) & : & \mathsf{tagDesc} \\ (E, D)^{*} & X & \mapsto & \begin{cases} \text{'ret}: & \Sigma\, X\, \lambda - .\, 1 \\ E \;\; : & D \end{cases} \end{array}$$

Effectively, we simply add a constructor 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:

$$\begin{array}{lllll} (D : \mathsf{tagDesc})^* & (X : \textsc{Set}) & : & \textsc{Set} \\ D^* & X & \mapsto & \mu\,(D^*\,X) \end{array}$$

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 '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:

$$\begin{array}{llll} \Sigma_{\mathsf{List}}\,(A : \textsc{Set}) & : & \mathsf{Desc} \\ \Sigma_{\mathsf{List}} \quad A & \mapsto & \big\{ \text{'cons} : \; \Sigma\,A\,\lambda - .\,\mathsf{var} \times 1 \end{array}$$

We get List A by considering the free monad on the unit set:

$$\mathsf{List}\,A \cong (\Sigma_{\mathsf{List}}\,A)^*\,\mathbb{1}$$

6.56 Remark. This monadic construction of 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 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 $\mathsf{List} : \textsc{Set} \to \textsc{Set}$ as a monad on the category Set. The monadic structure is thus related to the parameter of 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:

$$\begin{array}{lcl} \Sigma_{\mathsf{IO}} & : & \mathsf{tagDesc} \\ \Sigma_{\mathsf{IO}} & \mapsto & \begin{cases} '\mathsf{put}: & \Sigma\, \mathsf{String}\, \lambda - .\, (\Pi\, \mathbb{1}\, \lambda - .\, \mathsf{var}) \times 1 \\ '\mathsf{get}: & (\Pi\, \mathsf{String}\, \lambda - .\, \mathsf{var}) \times 1 \end{cases} \end{array}$$

Following standard terminology [Plotkin and Power, 2003], the constructors $'\mathsf{put}$ and $'\mathsf{get}$ are called the *operations* of the signature. Taking the free monad of this signature, we obtain a TermIO monad:

$$\begin{array}{llcl} \mathsf{TermIO}\, (X : \textsc{Set}) & & : & \textsc{Set} \\ \mathsf{TermIO} & X & \mapsto & \Sigma_{\mathsf{IO}}^{*} X \end{array}$$

Equipped with the following *generic operations*:

$$\begin{array}{llcl} \mathsf{putString}\, (s : \mathsf{String}) & & : & \mathsf{TermIO}\, \mathbb{1} \\ \mathsf{putString} & s & \mapsto & \mathsf{put}\, s\, \mathsf{ret} \end{array}$$

$$\begin{array}{lcl} \mathsf{getString} & : & \mathsf{TermIO}\, \mathsf{String} \\ \mathsf{getString} & \mapsto & \mathsf{get}\, \mathsf{ret} \end{array}$$

(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.

6.59 Definition (Kleisli category)**.** Let (T, η, μ) be a monad over $\mathbb{C}$.

The Kleisli category of T is the category $\mathbb{C}_T$ whose objects are objects of $\mathbb{C}$ and morphisms from C to D in $\mathbb{C}_T$ are morphisms of from C to $T\ D$ in $\mathbb{C}$. In the programming community, the identity is called `return`, while the (postfix) extension operator (which defines composition) is called `bind`, denoted `>>=`, of type $T\ A \to (A \to T\ B) \to T\ B$.

(6.60) 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 $'\mathsf{ret}$ constructor:

$$\begin{array}{llcl} \mathsf{return}\, (x : X) & & : & D^*\ X \\ \mathsf{return} & x & \mapsto & \mathsf{ret}\, x \end{array}$$

Intuitively, return introduces variables.

(6.62) Meanwhile, the postfix extension operator $\mathsf{subst}\ mx\ \sigma$ corresponds to the *substitution* of the variables in mx – introduced by $\mathsf{'ret}$ – with the corresponding term given by σ. We generically implement this substitution using the catamorphism. Let us write the type, and start filling in the blanks:

$$\begin{array}{l}
\mathsf{subst}\,(\sigma\!:\!X \rightarrow D^{*}\;Y)\;(mx\!:\!D^{*}\;X)\;:\;D^{*}\;Y\\
\mathsf{subst}\;\sigma\;mx\;\mapsto\;\llparenthesis\;\{?\!:\!(xs\!:\![\![D^{*}\;X]\!]\;(D^{*}\;Y)) \rightarrow D^{*}\;Y\}\;\rrparenthesis\;mx
\end{array}$$

(6.63) We are left with implementing the algebra of the catamorphism. Its role is to substitute $[\mathsf{'ret}\;x]$ with $\sigma\;x$. This corresponds to the following definition:

$$\begin{array}{llll}
\mathsf{apply}\,(\sigma\!:\!X \rightarrow D^{*}\;Y) & (xs\!:\![\![D^{*}\;X]\!]\;(D^{*}\;Y)) & : & D^{*}\;Y\\
\mathsf{apply}\quad\sigma & [\mathsf{'ret}\;x] & \mapsto & \sigma\;x\\
\mathsf{apply}\quad\sigma & (c, xs) & \mapsto & \mathsf{in}\,(c, xs)
\end{array}$$

6.64 Definition (Extension operator and Composition)**.** We can therefore complete the hole left in ¶ 6.62 and obtain the extension operator:

$$\begin{array}{l}
\mathsf{subst}\,(\sigma\!:\!X \rightarrow D^{*}\;Y)\;(mx\!:\!D^{*}\;X)\;:\;D^{*}\;Y\\
\mathsf{subst}\;\sigma\;mx\;\mapsto\;\llparenthesis\mathsf{apply}\;\sigma\rrparenthesis\;mx
\end{array}$$

And thus composition in the Kleisli category:

$$\begin{array}{lllll}
(\rho\!:\!Y \rightarrow D^{*}\;Z) & \circledast & (\sigma\!:\!X \rightarrow D^{*}\;Y) & : & X \rightarrow D^{*}\;Z\\
\rho & \circledast & \sigma & \mapsto & (\mathsf{subst}\;\rho) \circ \sigma
\end{array}$$

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, $\mathsf{'cons}$. 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 nil constructor of xs:

$$\begin{array}{lllll}
(xs\!:\!\mathsf{List}\;A) & +\!\!+ & (ys\!:\!\mathsf{List}\;A) & : & \mathsf{List}\;A\\
xs & +\!\!+ & ys & \mapsto & \mathsf{subst}\;xs\;\lambda *.\,ys
\end{array}$$

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 TermIO monad, we can denote the program that reads two strings and prints their concatenation:

$$\begin{array}{lcl}\mathsf{example} & : & \mathsf{TermIO}\,\mathbb{1} \\ \mathsf{example} & \mapsto & \mathsf{getString} \mathbin{\texttt{>>=}} \lambda s_1. \\ & & \mathsf{getString} \mathbin{\texttt{>>=}} \lambda s_2. \\ & & \mathsf{putString}(s_1 +\!\!+\, s_2)\end{array}$$

Here, >>= denotes the postfix extension operator, *i.e.* $mx \mathbin{\texttt{>>=}} \sigma \triangleq \mathsf{subst}\,\sigma\, mx$.

(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 (*i.e.* substitution of variables) associated with this monad.

6.2.3. Indexed free monad

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. 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 ′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:

$$\begin{array}{l}(D\,{:}\,\mathsf{tagIDesc}\,I)^{*}\ (X\,{:}\,I \to \mathrm{S{\scriptsize ET}})\ :\ \mathsf{tagIDesc}\,I \\ \left(\begin{cases} i \leftarrow E: & T \\ p \leftarrow F: & U\end{cases}\right)^{*} X \mapsto \begin{cases} i \leftarrow {}'\mathsf{ret}: & \Sigma\,(X\,i)\,\lambda - .\,1 \\ i \leftarrow E & :\ T \\ p \leftarrow F & :\ U\end{cases}\end{array}$$

6.70 Example (Free monad: filesystem interface)**.** 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 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:

$$\begin{array}{llcll}
\{s = \mathsf{open}\} & \mathsf{readFile} & : & \mathsf{String} & \{s = \mathsf{open}\} \\
\{s = \mathsf{closed}\} & \mathsf{openFile}\,(fn\!:\!\mathsf{String}) & : & \mathbb{1} & \{\} \\
\{s = \mathsf{open}\} & \mathsf{closeFile} & : & \mathbb{1} & \{s = \mathsf{closed}\}
\end{array}$$

Following McBride, we implement this specification with the following description:

$$\begin{array}{lcl}
\Sigma_{\mathsf{FS}} & : & \mathsf{IDesc}\,\mathsf{State} \\
\Sigma_{\mathsf{FS}} & \mapsto & \begin{cases}
\mathsf{closed} \leftarrow \mathsf{'openFile} : & \Sigma\,\mathsf{String}\,\lambda -.\,(\Pi\,\mathsf{State}\,\lambda s.\,\Pi\,\mathbb{1}\,\lambda -.\,\mathsf{var}\,s) \times 1 \\
\ \mathsf{open}\ \ \leftarrow \mathsf{'readFile}\ : & (\Pi\,\mathsf{State}\,\lambda s.\,\Pi\,(s = \mathsf{open} \times \mathsf{String})\,\lambda -.\,\mathsf{var}\,s) \times 1 \\
\ \mathsf{open}\ \ \leftarrow \mathsf{'closeFile} : & (\Pi\,\mathsf{State}\,\lambda s.\,\Pi\,(s = \mathsf{closed} \times \mathbb{1})\,\lambda -.\,\mathsf{var}\,s) \times 1
\end{cases}
\end{array}$$

Taking the free monad of that signature, we obtain the FS monad

$$\begin{array}{l}
\mathsf{FS}\,(X\!:\!\mathsf{State} \to \textsc{Set})\ :\ \mathsf{State} \to \textsc{Set} \\
\mathsf{FS}\,X\ \mapsto\ \Sigma_{\mathsf{FS}}^{*}\,X
\end{array}$$

that is equipped with the generic operations:

$$\begin{array}{lcl}
\mathsf{openFile}\,(fn\!:\!\mathsf{String}) & : & \mathsf{FS}\,(\lambda -.\,\mathbb{1})\ \mathsf{open} \\
\mathsf{openFile}\quad fn & \mapsto & \mathsf{openFile}\,fn\,\lambda s.\,\mathsf{ret}
\end{array}$$

$$\begin{array}{lcl}
\mathsf{readFile} & : & \mathsf{FS}\,(\lambda s.\,s = \mathsf{open} \times \mathsf{String})\ \mathsf{open} \\
\mathsf{readFile} & \mapsto & \mathsf{readFile}\,\lambda s.\,\mathsf{ret}
\end{array}$$

$$\begin{array}{lcl}
\mathsf{closeFile} & : & \mathsf{FS}\,(\lambda s.\,s = \mathsf{closed} \times \mathbb{1})\ \mathsf{closed} \\
\mathsf{closeFile} & \mapsto & \mathsf{closeFile}\,\lambda s.\,\mathsf{ret}
\end{array}$$

6.71 Example (Indexed descriptions)**.** An interesting instance of free monad is IDesc itself. Indeed, var is nothing but the return of a free monad. The remaining constructors form the operations, trivially indexed by $\mathbb{1}$. The signature functor is described by:

$$
\begin{array}{lcl}
\mathsf{IDescFreeD} & : & \mathsf{tagIDesc}\,\mathbb{1} \\
\mathsf{IDescFreeD} & \mapsto & \begin{cases} - \leftarrow 1 : & 1 \\ - \leftarrow \times : & \mathsf{var}\,{*} \times \mathsf{var}\,{*} \times 1 \\ - \leftarrow \sigma : & \Sigma\,\mathsf{EnumU}\,\lambda E.\,\pi\,E \times 1 \\ - \leftarrow \Sigma : & \Sigma\,\textsc{Set}\,\lambda S.\,(\Pi\,S\,\lambda -.\,\mathsf{var}\,{*}) \times 1 \\ - \leftarrow \Pi : & \Sigma\,\textsc{Set}\,\lambda S.\,(\Pi\,S\,\lambda -.\,\mathsf{var}\,{*}) \times 1 \end{cases}
\end{array}
$$

We obtain $\mathsf{IDesc}\,I$ by applying the free monad construction on this signature, at constant index I:

$$
\begin{array}{lll}
\mathsf{IDescD}\;(I : \textsc{Set}) & : & \mathsf{tagIDesc}\,\mathbb{1} \\
\mathsf{IDescD}\quad\; I & \mapsto & \mathsf{IDescFreeD}^{*}\,\lambda -.\,I
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Context} : \textsc{Set}\ \textbf{where} \\
\quad \mathsf{Context} \ni [] \\
\quad \mathsf{Context} \ni \mathsf{snoc}\,(\Gamma : \mathsf{Context})(ty : \mathsf{Ty})
\end{array}
$$

An environment realises a context, packing a value for each type:

$$
\begin{array}{llll}
\mathsf{Env} & (\Gamma : \mathsf{Context}) & : & \textsc{Set} \\
\mathsf{Env} & [] & \mapsto & \mathbb{1} \\
\mathsf{Env} & (\mathsf{snoc}\,\Gamma\,S) & \mapsto & \mathsf{Env}\,\Gamma \times \mathsf{Val}\,S
\end{array}
$$

In this setting, we define type variables, Var by

$$
\begin{array}{lllll}
\mathsf{Var} & (\Gamma : \mathsf{Context}) & (T : \mathsf{Ty}) & : & \textsc{Set} \\
\mathsf{Var} & [] & T & \mapsto & \mathbb{0} \\
\mathsf{Var} & (\mathsf{snoc}\,\Gamma\,S) & T & \mapsto & (\mathsf{Var}\,\Gamma\,T) + (S = T)
\end{array}
$$

while Val maps the type to the corresponding host type, Var indexes a value in the context, obtaining a proof that the types match.

The lookup function precisely follows this semantics:

$$\begin{array}{llll} \mathsf{lookup}\ (\gamma : \mathsf{Env}\,\Gamma)\ (v : \mathsf{Var}\,\Gamma\ T) & : & \mathsf{Val}\,T \\ \mathsf{lookup}\quad (\gamma, t) \quad (\mathsf{inj}_r\,\mathsf{refl}) & \mapsto & t \\ \mathsf{lookup}\quad (\gamma, t) \quad (\mathsf{inj}_l\,x) & \mapsto & \mathsf{lookup}\,\gamma\,x \end{array}$$

Consequently, taking the free monad of ExprD by $\mathsf{Var}\,\Gamma$, we obtain the language of open terms in a context Γ:

$$\begin{array}{lll} \mathsf{openTm}\ (\Gamma : \mathsf{Context}) & : & \mathsf{Ty} \to \mathrm{SET} \\ \mathsf{openTm}\quad \Gamma & \mapsto & \mathsf{ExprD}^*\ (\mathsf{Var}\,\Gamma) \end{array}$$

In this setting, the language of closed terms corresponds to the free monad assigning an empty set of values to variables:

$$\begin{array}{lll} \mathsf{closeTm} & : & \mathsf{Ty} \to \mathrm{SET} \\ \mathsf{closeTm} & \mapsto & \mathsf{ExprD}^*\ (\lambda - .\,\mathbb{0}) \end{array}$$

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:

$$\begin{array}{lll} \mathsf{update}\ (tm : \mu\,\mathsf{ExprD}\ ty) & : & \mathsf{closeTm}\ ty \\ \mathsf{update}\quad tm & \mapsto & (\!|(\lambda(tag, tm).\,\mathsf{in}\ (1\!+\ tag, tm))|\!)\ tm \end{array}$$

The other direction of the isomorphism is straightforward, the 'ret case being impossible. We are therefore entitled to reuse the $\mathsf{eval}_{\Downarrow}$ 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:

$$\begin{array}{lll} \mathsf{return}\ (x : X\,i) & : & D^*\ X\ i \\ \mathsf{return}\quad x & \mapsto & \mathsf{ret}\,x \end{array}$$

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

$$\begin{array}{l} \mathsf{substI}\ (\sigma : X \dot{\to} D^*\ Y)\ :\ D^*\ X \dot{\to} D^*\ Y \\ \mathsf{substI}\ \sigma\ \mapsto\ (\!|\mathsf{applyI}\ \sigma|\!) \end{array}$$

where applyI is defined as follows:

$$\begin{array}{llllll} \mathsf{applyI} & (\sigma : X \dot{\to} D^*\ Y) & (xs : [\![D^*\ X]\!]\ D^*\ Y\ i) & : & D^*\ Y\ i \\ \mathsf{applyI} & \sigma & [\text{'}\mathsf{ret}\ x] & \mapsto & \sigma\ i\ x \\ \mathsf{applyI} & \sigma & (c, ys) & \mapsto & \mathsf{in}\ (c, ys) \end{array}$$

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 guarantee that the file is closed when it returns:

$$\begin{array}{llll} \mathsf{example}\ (fn : \mathsf{String}) & : & \mathsf{FS}\ (\lambda s.\, s = \mathsf{closed} \times \mathsf{String})\ \mathsf{closed} \\ \mathsf{example}\quad fn & \mapsto & \mathsf{openFile}\ fn \ {>}{>}{=} \\ & & \lambda \begin{cases} \mathsf{closed}\ * \mapsto \mathsf{ret}\ \texttt{""} & \textit{– File does not exist: silently fail} \\ \mathsf{open}\ * \ \ \mapsto \mathsf{readFile} \ {>}{>}{=}\ \lambda \mathsf{open}\ str. \\ \qquad\qquad \mathsf{closeFile} \ {>}{>}{=}\ \lambda \mathsf{closed}\ *. \\ \qquad\qquad \mathsf{ret}\ str \end{cases} \end{array}$$

Forgetting to call $\mathsf{closeFile}$ would be caught at type checking: the file would be in a state open, and not closed as specified by the post-condition $\lambda s.\, s = \mathsf{closed} \times \mathsf{String}$.

6.77 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 $\mathsf{eval}_{\Downarrow}$, 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 $\mathsf{discharge}$; then, applying this function to every variable in the term. The $\mathsf{discharge}$ function maps values to themselves, and variables to their value in context:

$$\begin{array}{llll} \mathsf{discharge}\,(\gamma\!:\!\mathsf{Env}\,\Gamma) & (v\!:\!\mathsf{Var}\,\Gamma\,T) & : & \mathsf{closeTm}\,T \\ \mathsf{discharge}\quad\gamma & \quad v & \mapsto & \mathsf{val}\,(\mathsf{lookup}\,\gamma\,v) \end{array}$$

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:

$$\begin{array}{l} \mathsf{substExpr}\,(\sigma\!:\!\mathsf{Var}\,\Gamma \dot{\to} \mathsf{closeTm})\,(tm\!:\!\mathsf{openTm}\,\Gamma\,T)\ :\ \mathsf{closeTm}\,T \\ \mathsf{substExpr}\,\sigma\,tm\ \mapsto\ \mathsf{substI}\,\sigma\,tm \end{array}$$

This completes our implementation of the open term's interpreter:

$$\begin{array}{l} \mathsf{evalTerm}_{\Downarrow}\,(\gamma\!:\!\mathsf{Env}\,\Gamma)\,(tm\!:\!\mathsf{openTm}\,\Gamma\,T)\ :\ \mathsf{Val}\,T \\ \mathsf{evalTerm}_{\Downarrow}\,\gamma\,tm\ \mapsto\ \mathsf{eval}_{\Downarrow}\,(\mathsf{substExpr}\,(\mathsf{discharge}\,\gamma)\,tm) \end{array}$$

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 ℓ 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:

$$\begin{tikzcd}
\vdots & & \mathsf{Desc}_{\ell+1} \arrow[dll, "\text{(Def. 6.93)}"'] \arrow[d, dashed, no head, "\text{(Def. 6.87)}"] & : & \mathsf{Set}_{\ell+2} \\
\mu\,\mathsf{DescD}_{\ell} & \underset{\text{(Th. 6.96)}}{\cong} & \mathsf{Desc}_{\ell} & : & \mathsf{Set}_{\ell+1}
\end{tikzcd}$$

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 Op – the collection of nodes – and 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 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

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{Op} : \textsc{Set}\\ \Gamma \vdash \mathsf{Ar} : \mathsf{Op} \to \textsc{Set}\end{array}}{\Gamma \vdash \mathcal{W}\,\mathsf{Op}\,\mathsf{Ar} : \textsc{Set}} \qquad \frac{\begin{array}{l}\Gamma \vdash op : \mathsf{Op}\\ \Gamma \vdash xs : \mathsf{Ar}\,op \to \mathcal{W}\,\mathsf{Op}\,\mathsf{Ar}\end{array}}{\Gamma \vdash \mathsf{sup}\,op\,xs : \mathcal{W}\,\mathsf{Op}\,\mathsf{Ar}}$$

$$\begin{array}{ll}\mathsf{W\text{-}elim} & (ih : (op : \mathsf{Op})(xs : \mathsf{Ar}\,op \to \mathcal{W}\,\mathsf{Op}\,\mathsf{Ar}) \to ((ar : \mathsf{Ar}\,op) \to Q\ (xs\,ar)) \to Q\ (\mathsf{sup}\,op\,xs))\\ & (xs : \mathcal{W}\,\mathsf{Op}\,\mathsf{Ar}) : Q\,xs\\ \multicolumn{2}{l}{\mathsf{W\text{-}elim}\,ih\ (\mathsf{sup}\,op\,xs) \ \mapsto\ ih\,op\,xs\ (\lambda ar.\,\mathsf{W\text{-}elim}\,ih\ (xs\,ar))}\end{array}$$

Figure 6.1.: W-types

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.* Σ-types, Π 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 Π-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:

$$\begin{array}{l}\textbf{data}\ \hat{U}\ [U : \textsc{Set}][El : U \to \textsc{Set}] : \textsc{Set}\ \textbf{where}\\ \quad \hat{U}\,U\,El \ni U`\\ \qquad\qquad \mid\ El`(T : U)\\ \qquad\qquad \mid\ \Pi`(S : \hat{U}\,U\,El)\\ \qquad\qquad\qquad (T : \hat{El}\,S \to \hat{U}\,U\,El)\\ \qquad\qquad \mid\ \Sigma`(S : \hat{U}\,U\,El)\\ \qquad\qquad\qquad (T : \hat{El}\,S \to \hat{U}\,U\,El)\\ \qquad\qquad \mid\ \mathsf{EnumT}`(E : \mathsf{EnumU})\\ \qquad\qquad \mid\ \mathsf{W}`(S : \hat{U}\,U\,El)\\ \qquad\qquad\qquad (P : \hat{El}\,S \to \hat{U}\,U\,El)\end{array}$$

$$\begin{array}{lcl}\hat{El}\ (T : \hat{U}\,U\,El) & : & \textsc{Set}\\ \hat{El}\quad U` & \mapsto & U\\ \hat{El}\quad (El`\,T) & \mapsto & El\,T\\ \hat{El}\quad (\Pi`\,S\,T) & \mapsto & (s : \hat{El}\,S) \to \hat{El}\,(T\,s)\\ \hat{El}\quad (\Sigma`\,S\,T) & \mapsto & (s : \hat{El}\,S) \times \hat{El}\,(T\,s)\\ \hat{El}\ (\mathsf{EnumT}`\,E) & \mapsto & \mathsf{EnumT}\,E\\ \hat{El}\quad (\mathsf{W}`\,S\,P) & \mapsto & \mathcal{W}\,(\hat{El}\,S)\ (\lambda s.\,\hat{El}\,(P\,s))\end{array}$$

6.83 Definition (Hierarchy of types)**.** We obtain the indexed hierarchy of types by iterating the next universe operator over levels, starting from the empty universe :

$$\begin{array}{lcl}\tilde{U}\ (\ell : \mathsf{Level}) & : & \textsc{Set}\\ \tilde{U}\quad 0 & \mapsto & \mathbb{0}\\ \tilde{U}\quad (\mathsf{suc}\,\ell) & \mapsto & \hat{U}\ (\tilde{U}\,\ell)\ \tilde{El}\end{array} \qquad \begin{array}{lcl}\tilde{El}\ (\ell : \mathsf{Level})\ (T : \tilde{U}\,\ell) & : & \textsc{Set}\\ \tilde{El}\quad 0 \qquad T & \mapsto & \mathbb{0}\\ \tilde{El}\quad (\mathsf{suc}\,\ell) \quad T & \mapsto & \hat{El}\,T\end{array}$$

We restrict ourselves to a countable collections of universes, thus defining levels by:

$$\begin{array}{l} \textbf{data}\ \mathsf{Level} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Level} \ni 0 \\ \qquad\qquad\ \mid \mathsf{suc}(\ell : \mathsf{Level}) \end{array}$$

(6.84) 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 EnumT‘. Every level is then closed under W-types, Σ-types and Π-types – thanks to, respectively, W‘, Σ‘, and Π‘.

Since the lowest-level universe is empty, the actual hierarchy starts at level $\mathsf{suc}\,0$. We therefore define

$$\begin{array}{lll|llll} \tilde{U}^+\ (\ell : \mathsf{Level}) & : & \textsc{Set} & \tilde{El}^+\ (\ell : \mathsf{Level}) & (T : \tilde{U}^+\ \ell) & : & \textsc{Set} \\ \tilde{U}^+\quad \ell & \mapsto & \tilde{U}\ (\mathsf{suc}\,\ell) & \tilde{El}^+\quad \ell & T & \mapsto & \tilde{El}\ (\mathsf{suc}\,\ell)\ T \end{array}$$

and work with this hierarchy, whose trivial bottom layer has been removed.

(6.85) In this model, we can formally write definitions inhabiting simultaneously the entire hierarchy (Remark 6.12). To do so, we simply quantify over ℓ : 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 ℓ: Level:

$$\begin{array}{lll} \mathsf{Nat}\ (\ell : \mathsf{Level}) & : & \tilde{U}^+\ \ell \\ \mathsf{Nat}^\ell & \mapsto & \mathsf{W}‘\left(\mathsf{EnumT}‘\left\{\begin{array}{l} ’0 \\ ’\mathsf{suc} \end{array}\right\}\right)\left\{\begin{array}{l} ’0 \mapsto \mathsf{EnumT}‘\ \{\} \\ ’\mathsf{suc} \mapsto \mathsf{EnumT}‘\ \{’*\} \end{array}\right\} \end{array}$$

$$\begin{array}{lll} 0 & : & \tilde{El}^+\ \mathsf{Nat}^\ell \\ 0 & \mapsto & \mathsf{sup}\,’0\ \mathbb{0}\text{-}\mathsf{elim} \end{array} \qquad \begin{array}{lll} \mathsf{suc} & : & \tilde{El}^+\ (\Pi‘\mathsf{Nat}^\ell \lambda - .\,\mathsf{Nat}^\ell) \\ \mathsf{suc} & \mapsto & \lambda n.\,\mathsf{sup}\,’\mathsf{suc}\,\lambda *.\,n \end{array}$$

6.87 Definition (Hard-wired descriptions)**.** Using W-types, we define a hard-wired universe of descriptions. We obtain the datatype Desc by defining its W-type at every level:

$$\begin{array}{lll} \mathsf{Desc}_M\ (\ell : \mathsf{Level}) & : & \tilde{U}^+\ \ell \\ \mathsf{Desc}_M{}^\ell & \mapsto & \mathsf{W}‘\left(\mathsf{EnumT}‘\left\{\begin{array}{c} 1 \\ \mathsf{var} \\ \Sigma \\ \Pi \end{array}\right\}\right)\left\{\begin{array}{l} 1 \quad \mapsto \mathsf{EnumT}‘\ \{’*\} \\ \mathsf{var} \mapsto \mathsf{EnumT}‘\ \{’*\} \\ \Sigma \quad \mapsto \Sigma‘\ U‘\ \lambda S.\,\underline{\Pi‘\ (El‘\ S)\ \lambda - .\,\mathsf{EnumT}‘\ \{’*\}} \\ \Pi \quad \mapsto \Sigma‘\ U‘\ \lambda S.\,\underline{\Pi‘\ (El‘\ S)\ \lambda - .\,\mathsf{EnumT}‘\ \{’*\}} \end{array}\right\} \end{array}$$

(6.88) We automatically obtain the Desc constructors by inhabiting this W-type. We shall skip their definition, which are otherwise unsurprising, and denote them 1_{M}, $\mathsf{var}_{\mathsf{M}}$, Σ_{M}, and Π_{M}, where "M" stands for "Meta". This definition corresponds to our presentation of Desc (Definition 4.6) as a meta-object in Chapter 4. Using the W-types, we conveniently model the formation and introduction rules of 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 Desc_M)**.** Given a code in Desc_M, we can build its least fixpoint, again using a W-type. The idea is to manually unfold the interpretation function $[\![-]\!]$ (Definition 4.6) in the definition of the fixpoint (Definition 4.38).

We obtain the following code:

$$
\begin{array}{lcll}
\mu_M\,(D\!:\!\mathsf{Desc}_M{}^{ell}) & : & \tilde{U}^+\,\ell & \\
\mu_M & D & \mapsto \mathsf{W}`\,\mu_{\mathsf{Sh}}\,\mu_{\mathsf{Pos}} & \textbf{where}
\end{array}
$$

$$
\begin{array}{lcl}
\mu_{\mathsf{Sh}}\,(D\!:\!\tilde{El}^+\,(\mathsf{Desc}_M{}^{\ell})) & : & \tilde{U}^+\,\ell \\
\mu_{\mathsf{Sh}} & 1_{\mathsf{M}} & \mapsto \mathsf{EnumT}`\,\{'\!*\} \\
\mu_{\mathsf{Sh}} & \mathsf{var}_{\mathsf{M}} & \mapsto \mathsf{EnumT}`\,\{'\!*\} \\
\mu_{\mathsf{Sh}} & (\Sigma_{\mathsf{M}}\,S\,T) & \mapsto \Sigma`\,S\,\lambda s.\,\mu_{\mathsf{Sh}}\,(T\,(s,'\!*)) \\
\mu_{\mathsf{Sh}} & (\Pi_{\mathsf{M}}\,S\,T) & \mapsto \Pi`\,S\,\lambda s.\,\mu_{\mathsf{Sh}}\,(T\,(s,'\!*))
\end{array}
$$

$$
\begin{array}{lccl}
\mu_{\mathsf{Pos}}\,(D\!:\!\tilde{El}^+\,(\mathsf{Desc}_M{}^{\ell}))\,(sh\!:\!\tilde{El}^+\,(\mu_{\mathsf{Sh}}\,D)) & & : & \tilde{U}^+\,\ell \\
\mu_{\mathsf{Pos}} & 1_{\mathsf{M}} & '\!* & \mapsto \mathsf{EnumT}`\,\{\} \\
\mu_{\mathsf{Pos}} & \mathsf{var}_{\mathsf{M}} & '\!* & \mapsto \mathsf{EnumT}`\,\{'\!*\} \\
\mu_{\mathsf{Pos}} & (\Sigma_{\mathsf{M}}\,S\,T) & (s,sh) & \mapsto \mu_{\mathsf{Pos}}\,(T\,(s,'\!*))\,sh \\
\mu_{\mathsf{Pos}} & (\Pi_{\mathsf{M}}\,S\,T) & f & \mapsto \Sigma`\,S\,\lambda s.\,\mu_{\mathsf{Pos}}\,(T\,(s,'\!*))\,(f\,s)
\end{array}
$$

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 Desc_M)**.** We define natural numbers exactly as we did in Example 4.18. This time, however, we work in Desc_M:

$$
\begin{aligned}
&\mathsf{NatD}\ (\ell\!:\!\mathsf{Level})\ :\ \tilde{El}^{+}\ (\mathsf{Desc}_M{}^{\ell})\ \ell\\
&\mathsf{NatD}^{\ell}\ \mapsto\ \Sigma_{\mathsf{M}}\ \mathsf{EnumT}`\left\{\begin{matrix}'0\\ '\mathsf{suc}\end{matrix}\right\}\left\{\begin{matrix}'0 & \mapsto 1_{\mathsf{M}}\\ '\mathsf{suc} & \mapsto \mathsf{var}_{\mathsf{M}}\end{matrix}\right\}\\
&\mathsf{Nat}\ (\ell\!:\!\mathsf{Level})\ :\ \tilde{U}^{+}\ \ell\\
&\mathsf{Nat}^{\ell}\ \mapsto\ \mu_M\ \mathsf{NatD}^{\ell}
\end{aligned}
$$

The constructors of the datatype Nat are thus the inhabitants of the W-type μ_M 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 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.

At this stage, we have built a model of the type theory presented in Chapter 4. The description grammar Desc is modelled by Desc_M, while its fixpoint μ is modelled by μ_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 Desc_ℓ using $\mathsf{Desc}_{\ell+1}$.

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 Desc_M and we quantify over sets using the code $U`$:

$$
\begin{aligned}
&\mathsf{DescD}_R\ (\ell\!:\!\mathsf{Level})\ :\ \tilde{El}^{+}\ (\mathsf{Desc}_M{}^{(\mathsf{suc}\,\ell)})\ \ell\\
&\mathsf{DescD}_R{}^{\ell}\ \mapsto\ \Sigma_{\mathsf{M}}\left\{\begin{matrix}'1\\ '\mathsf{var}\\ '\Sigma\\ '\Pi\end{matrix}\right\}\left\{\begin{matrix}'1 & \mapsto 1_{\mathsf{M}}\\ '\mathsf{var} & \mapsto 1_{\mathsf{M}}\\ '\Sigma & \mapsto \Sigma_{\mathsf{M}}\ U`\ \lambda S.\,\Pi_{\mathsf{M}}\ (El`\ S)\ \lambda -.\,\mathsf{var}_{\mathsf{M}}\\ '\Pi & \mapsto \Sigma_{\mathsf{M}}\ U`\ \lambda S.\,\Pi_{\mathsf{M}}\ (El`\ S)\ \lambda -.\,\mathsf{var}_{\mathsf{M}}\end{matrix}\right\}\\
&\mathsf{Desc}_R\ (\ell\!:\!\mathsf{Level})\ :\ \tilde{U}^{+}\ (\mathsf{suc}\ \ell)\\
&\mathsf{Desc}_R{}^{\ell}\ \mapsto\ \mu_M\ \mathsf{DescD}_R{}^{\ell}
\end{aligned}
$$

(6.94) Diagrammatically, the situation is as follows:

$$\begin{tikzcd}
\vdots & \mathsf{Desc}_M{}^{\ell+2} \arrow[dl] \arrow[d, dash, dashed] & : & \tilde{U}^{+}\,(\ell+2) \\
\mathsf{Desc}_R{}^{\ell+1} & \mathsf{Desc}_M{}^{\ell+1} \arrow[dl] \arrow[d, dash, dashed] & : & \tilde{U}^{+}\,(\ell+1) \\
\mathsf{Desc}_R{}^{\ell} & \mathsf{Desc}_M{}^{\ell} & : & \tilde{U}^{+}\,\ell
\end{tikzcd}$$

The dashed lines representing the simultaneous instantiation of Desc_M over all universe levels, while the arrows represents the implementation of the target Desc_R using a code inhabiting its source Desc_M.

(6.95) The levitation trick consists in observing that, at every level ℓ, the meta-level $\mathsf{Desc}_M{}^{\ell}$ and the reflected $\mathsf{Desc}_R{}^{\ell}$ are actually isomorphic.

6.96 Theorem. At every level ℓ, the datatype $\mathsf{Desc}_M{}^{\ell}$ is isomorphic to $\mathsf{Desc}_R{}^{\ell}$ and the bijection is an identity.

Proof. Looking at the constructors of the respective datatype, the isomorphism is trivial:

Constructors of Desc_M :	(unfolds to)	Constructors of Desc_R :
$1_{\mathsf{M}} \triangleq$	$\mathsf{sup}\,(1, {'}{*})\ \mathbb{0}\text{-}\mathsf{elim}$	$\triangleq {'}1_R$
$\mathsf{var}_{\mathsf{M}} \triangleq$	$\mathsf{sup}\,(\mathsf{var}, {'}{*})\ \mathbb{0}\text{-}\mathsf{elim}$	$\triangleq {'}\mathsf{var}_R$
$\Sigma_{\mathsf{M}}\,S\,T \triangleq$	$\mathsf{sup}\,(\Sigma, (S, \lambda - .\, {'}{*}))\ \lambda(s, {'}{*}).\, T\,s$	$\triangleq {'}\Sigma_R\,S\,T$
$\Pi_{\mathsf{M}}\,S\,T \triangleq$	$\mathsf{sup}\,(\Pi, (S, \lambda - .\, {'}{*}))\ \lambda(s, {'}{*}).\, T\,s$	$\triangleq {'}\Pi_R\,S\,T$

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, Desc_M and 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:

$$\begin{tikzcd}
{\mathsf{Desc}_R}^{\ell+2} \arrow[r, "\cong"] & {\mathsf{Desc}_M}^{\ell+2} \arrow[dl] & : & \tilde{U}^+ \, (\ell+2) \\
{\mathsf{Desc}_R}^{\ell+1} \arrow[r, "\cong"] & {\mathsf{Desc}_M}^{\ell+1} \arrow[dl] & : & \tilde{U}^+ \, (\ell+1) \\
{\mathsf{Desc}_R}^{\ell} \arrow[r, "\cong"] & {\mathsf{Desc}_M}^{\ell} & : & \tilde{U}^+ \, \ell
\end{tikzcd}$$

We can therefore dispose entirely of the tower of Desc_M and present Desc_R as a self-supporting tower of definitions.

6.98 Remark (Absence of $\mathsf{switch}_\mathsf{D}$ in the model)**.** Our model does not capture the $\mathsf{switch}_\mathsf{D}$ 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, $\mathsf{switch}_\mathsf{D}$ 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 Desc_M definition)**.** In ¶ 6.89, we noticed that the W-type encoding of 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 Desc_R are the result of the interpretation of DescD_R through μ_M: the resulting W-type contains some unavoidable encoding noise. Knowing this noise, we simply defined Desc_M in such a way that its interpretation would *artificially* generate the same noise. This way, the constructors of Desc_M and the constructors of Desc_R match on the nose: they are trivially (extensionally) equal. If we had been less careful in defining 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 Desc appears only in a set polymorphic sense: the principal type of the encoded description generalises to the type of 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 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

(6.101) 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.

(6.102) 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 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.

(6.103) 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 Desc_ℓ from $\mathsf{Desc}_{\ell+1}$. 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

(6.104) 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.

(6.105) 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-sigmas view, are a natural example.

(6.106) We do not support polykinded programming [Hinze, 2000b], while it is supported by Generic Haskell. Our descriptions are limited to endofunctors on $\textsc{Set}^I$. Whilst indexing is known to be sufficient to *encode* a large class of higher-kinded 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.

(6.107) 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.

(6.108) 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.

(6.109) 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.

(7.4) Our syntax for inductive definitions follows the sum-of-products canon:

$$\begin{array}{l} \textbf{data}\ \mathsf{D}\ \overrightarrow{[p : P]} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{D}\ \vec{p}\ \ni\ \mathsf{c}_0\ \overrightarrow{(a_0 : T_0)} \\ \qquad\quad \mid\ \ldots \\ \qquad\quad \mid\ \mathsf{c}_k\ \overrightarrow{(a_k : T_k)} \end{array}$$

The arguments $\vec{p}$ are parameters. A T_i can be recursive, *i.e.* refer to $\mathsf{D}\ \vec{p}$, 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:

ELABORATION

$$\begin{array}{rcl} \langle \mathit{data} \rangle & ::= & \textbf{data D}\ ([p : \langle T \rangle])^* : \langle T \rangle\ \textbf{where}\ \langle \mathit{choices} \rangle \\ \langle \mathit{choices} \rangle & ::= & \textbf{D}\ (p)^*\ \ni\ \langle \mathit{constructor} \rangle\ ('|'\ \langle \mathit{constructor} \rangle)^* \\ \langle \mathit{constructor} \rangle & ::= & \textbf{con}\ \langle \mathit{arguments} \rangle \\ \langle \mathit{arguments} \rangle & ::= & ('('x : \langle T \rangle')')^* \end{array}$$

The terminals **D** and **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.

(7.6) 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 $\langle \mathit{choices} \rangle$) and is translated to a Σ code over the finite set of constructors. The second level structure consists of the Σ-telescope of arguments (*i.e.* the non-terminal $\langle \mathit{arguments} \rangle$): it translates to right-nested Σ codes. When elaborating arguments, we must make sure that the recursive arguments are valid, and translate them to the var code.

7.1.1. Description labels

(7.7) 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 $\lbag l \rbag$ 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 Desc code with a high-level presentation, *i.e.* the name and parameters of the datatype being defined. The elaborative steps are ran 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

$$\langle l \rangle ::= \mathbf{D}\,(\langle t \rangle)^{*} \qquad\qquad \langle T \rangle ::= \ \ldots \quad | \quad \lbag \langle l \rangle \rbag$$

We then define a set former $\lbag l \rbag$ (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$\lbag l \rbag$, 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

(7.10) 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.

META-THEORY

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \lbag l \rbag : \textsc{Set}_{l+1}}$$

$$\frac{\Gamma \vdash D : \mathsf{tagDesc}}{\Gamma \vdash \mathsf{return}\, D : \lbag l \rbag} \qquad \begin{array}{lcl} \mathsf{call} \lbag l \rbag\, (t : \lbag l \rbag) & : & \mathsf{tagDesc} \\ \mathsf{call} \lbag l \rbag\, (\mathsf{return}\, D) & \mapsto & D \end{array}$$

Figure 7.1.: Description label

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 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:

$$(1) \vdash (2) \overset{(3)}{\leadsto} (4)$$

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:

$$\begin{array}{l} \textbf{data}\ \mathsf{Tree}\,[A : \textsc{Set}] : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Tree}\, A \ni \mathsf{leaf} \\ \qquad\qquad\ \mid\ \mathsf{node}\,(a : A)(b : \mathsf{Bool} \to \mathsf{Tree}\, A) \end{array}$$

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:

$$\begin{array}{l}\mathsf{Tree}\,(A\!:\!\mathrm{SET})\ :\ \mathrm{SET}\\ \mathsf{Tree}\,A\ \mapsto\ \mu\ \left(\Sigma\,\mathsf{EnumT}\left\{\begin{array}{l}\mathsf{'leaf}\\ \mathsf{'node}\end{array}\right\}\left\{\begin{array}{l}\mathsf{'leaf}\ \ \mapsto 1\\ \mathsf{'node}\mapsto \Sigma\,A\,\lambda-.\,(\Pi\,\mathsf{Bool}\,\lambda-.\,\mathsf{var})\times 1\end{array}\right\}\right)\end{array}$$

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 LET keyword.

(7.14) **Elaboration of an inductive definition (Figure 7.2a.)**

$$\boxed{\Gamma \vdash \textbf{data}\ D\ \overrightarrow{(p\!:\!P)}\!:\!\mathrm{SET}\ \textbf{where}\ \mathit{choices} \overset{D}{\leadsto} \Delta}$$

This judgment reads as: in context Γ, the definition **data** $D\ \overrightarrow{(p\!:\!P)}\!:\!\mathrm{SET}$ **where** *choices* extends the original context to a context Δ 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 $\overset{Cs}{\leadsto}$ – introducing a description label in the process.

7.15 Example (Elaborating Tree)**.** Applied to our example, we obtain:

LET: Γ a valid context

$\langle 1\rangle 1.$ $\vdash$ **data** $\mathsf{Tree}\,[A\!:\!\mathrm{SET}]\!:\!\mathrm{SET}$ **where** $[\mathit{choices}]$
$\overset{D}{\leadsto} \Gamma[\mathsf{Tree} \mapsto \lambda A.\,\mu\,(\mathsf{call}\Lbag\textbf{Tree}\ A\Rbag\,[\mathit{code}])\!:\!A \to \mathrm{SET}]$

BY: Example 7.18 applied to rule (DATA)

$$\boxed{\langle\Gamma\rangle \vdash \langle \mathit{data}\rangle \overset{D}{\leadsto} \langle\Gamma\rangle}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash \textsc{Set}_1 \ni \overrightarrow{(p:P)} \to \textsc{Set} \overset{\mathit{Chk}}{\leadsto} \overrightarrow{(p:P')} \to \textsc{Set} \\ \Gamma; \overrightarrow{p:P'} \vdash \mathbf{D}\ \vec{p} \ni \mathit{choices} \overset{\mathit{Cs}}{\leadsto} \mathit{code}\end{array}}{\begin{array}{c}\Gamma \vdash \textsf{data}\ \ D\ [\overrightarrow{p:P}] : \textsc{Set}\ \textsf{where}\ \mathit{choices} \overset{D}{\leadsto} \\ \Gamma[D \mapsto \lambda\vec{p}.\, \mu\, (\mathsf{call}\lbag \mathbf{D}\ \vec{p} \rbag\, \mathit{code}) : \overrightarrow{(p:P')} \to \textsc{Set}]\end{array}}\ (\textsc{Data})$$

(a) Elaboration of definition

$$\boxed{\langle\Gamma\rangle \vdash \langle l\rangle \ni \langle \mathit{choices}\rangle \overset{\mathit{Cs}}{\leadsto} \langle t\rangle}$$

$$\dfrac{\begin{array}{c} T \rhd l \\ \Gamma \vdash l \ni c_i \overset{C}{\leadsto} [t_i \mapsto \mathit{code}_i]\end{array} \qquad \begin{array}{c}\Gamma \vdash \mathsf{EnumU} \ni \{t_i\} \overset{\mathit{Chk}}{\leadsto} E \\ \Gamma \vdash \pi\, E\, \lambda - .\, \mathsf{Desc} \ni \{t_i \mapsto \mathit{code}_i\} \overset{\mathit{Chk}}{\leadsto} T\end{array}}{\Gamma \vdash l \ni T \ \backepsilon c_0 | \ldots | c_n \overset{\mathit{Cs}}{\leadsto} \mathsf{return}\ (E, T)}\ (\textsc{Choices})$$

(b) Elaboration of constructor choices

$$\boxed{\langle\Gamma\rangle \vdash \langle l\rangle \ni \langle \mathit{constructor}\rangle \overset{C}{\leadsto} [\langle t\rangle \mapsto \langle t\rangle]}$$

$$\dfrac{\begin{array}{c}\Gamma \vdash \mathsf{UId} \ni {'t} \overset{\mathit{Chk}}{\leadsto} t' \\ \Gamma \vdash l \ni \mathit{args} \overset{A}{\leadsto} \mathit{code}\end{array}}{\Gamma \vdash l \ni t\ \mathit{args} \overset{C}{\leadsto} [t' \mapsto \mathit{code}]}\ (\textsc{Constructor})$$

(c) Elaboration of a constructor

Figure 7.2.: Elaboration of inductive types

$$\boxed{\langle\Gamma\rangle \vdash \langle l\rangle \ni \langle \mathit{arguments}\rangle \overset{A}{\leadsto} \langle t\rangle}$$

$$\frac{\Gamma \vdash \textsc{Set} \ni T \overset{\mathit{Chk}}{\leadsto} T' \qquad \Gamma; x : T' \vdash l \ni \Delta \overset{A}{\leadsto} \mathit{code}_\Delta}{\Gamma \vdash l \ni (x : T)\Delta \overset{A}{\leadsto} \Sigma\, T'\, \lambda x.\, \mathit{code}_\Delta}\ (\textsc{Arg-Sig})$$

$$\frac{\Gamma \vdash l \ni \nabla \overset{R}{\leadsto} \mathit{code}_\nabla \qquad \Gamma \vdash l \ni \Delta \overset{A}{\leadsto} \mathit{code}_\Delta}{\Gamma \vdash l \ni (x : \nabla)\Delta \overset{A}{\leadsto} \mathit{code}_\nabla \times \mathit{code}_\Delta}\ (\textsc{Arg-Rec})$$

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash l \ni \epsilon \overset{A}{\leadsto} 1}\ (\textsc{Arg-End})$$

(d) Elaboration of arguments

$$\boxed{\langle\Gamma\rangle \vdash \langle l\rangle \ni \langle T\rangle \overset{R}{\leadsto} \langle t\rangle}$$

$$\frac{\Gamma \vdash \textsc{valid} \qquad T \rhd l}{\Gamma \vdash l \ni T \overset{R}{\leadsto} \mathsf{var}}\ (\textsc{Arg-Rec-Var})$$

$$\frac{\Gamma \vdash \textsc{Set} \ni T \overset{\mathit{Chk}}{\leadsto} T' \qquad \Gamma; t : T' \vdash l \ni \nabla \overset{R}{\leadsto} \mathit{code}_\nabla}{\Gamma \vdash l \ni (t : T) \to \nabla \overset{R}{\leadsto} \Pi\, T'\, \lambda t.\, \mathit{code}_\nabla}\ (\textsc{Arg-Rec-Exp})$$

$$\frac{\Gamma \vdash \textsc{Set} \ni T \overset{\mathit{Chk}}{\leadsto} T' \qquad \Gamma; t : T' \vdash l \ni \nabla \overset{R}{\leadsto} \mathit{code}_\nabla}{\Gamma \vdash l \ni (t : T) \times \nabla \overset{R}{\leadsto} \Sigma\, T'\, \lambda t.\, \mathit{code}_\nabla}\ (\textsc{Arg-Rec-Sig})$$

(e) Elaboration of recursive arguments

$$\boxed{\langle T\rangle \rhd \langle l\rangle}$$

$$\frac{}{D \rhd \mathbf{D}}\ (\textsc{Match-Name}) \qquad\qquad \frac{T \rhd l}{T\, p \rhd l\, p}\ (\textsc{Match-Param})$$

(f) Matching label

Figure 7.1.: Elaboration of inductive types

where

$$\begin{aligned} \mathit{choices} &\triangleq \mathsf{Tree}\,A \ni \mathsf{leaf} \mid \mathsf{node}\,(a\!:\!A)(b\!:\!\mathsf{Bool} \to \mathsf{Tree}\,A) \\ \mathit{code} &\triangleq \mathsf{return} \left\{ \begin{array}{l} '\mathsf{leaf} \\ '\mathsf{node} \end{array} \right\} \left\{ \begin{array}{l} '\mathsf{leaf} \;\; \mapsto 1 \\ '\mathsf{node} \mapsto \Sigma\,A\,\lambda - .\,(\Pi\,\mathsf{Bool}\,\lambda_{.}\,\mathsf{var}) \times 1 \end{array} \right\} \end{aligned}$$

(7.16) **Elaboration of constructor choices (Figure 7.2b.)**

$$\boxed{\Gamma \vdash l \ni \mathit{choices} \overset{Cs}{\rightsquigarrow} \mathit{code}}$$

This judgment reads as: in a context Γ, the sum of constructors *choices* defining the datatype l elaborates to a description *code*. To elaborate the choice of constructors, we elaborate each individual constructor – via judgment $\overset{\mathsf{C}}{\rightsquigarrow}$ – 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:

7.17 Lemma. If $\Gamma \vdash l \ni \mathit{choices} \overset{Cs}{\rightsquigarrow} \mathit{code}$, then $\Gamma \vdash \mathit{code} : \wr l \wr$

7.18 Example (Elaborating Tree)**.** Applied to our example, we obtain:

LET: 1. Γ a valid context
 2. A : SET

$\langle 1 \rangle$1. $\vdash$ **Tree** $A \ni [\mathit{choices}] \overset{Cs}{\rightsquigarrow} [\mathit{code}]$
BY: By Example 7.21 applied to rule (CHOICES)

Where *choices* and *code* have been defined above.

(7.19) **Elaboration of a constructor (Figure 7.2c.)**

$$\boxed{\Gamma \vdash l \ni c \overset{\mathsf{C}}{\rightsquigarrow} [t \mapsto \mathit{code}]}$$

This judgment reads as: in context Γ, the constructor c defining a datatype l elaborates to a tag t, the constructor name, and a description *code*. The role of this elaboration step is twofold. First, we extract the constructor name and elaborate it to a tag – via type checking against UId. Second, we elaborate the arguments of that constructor – via judgment $\overset{A}{\rightsquigarrow}$ – 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 $\Gamma \vdash l \ni c \overset{C}{\rightsquigarrow} [t \mapsto \mathit{code}]$, then $\begin{cases} \Gamma \vdash t : \mathsf{UId} \\ \Gamma \vdash \mathit{code} : \mathsf{Desc} \end{cases}$

7.21 Example (Elaborating Tree)**.** Since our datatype has two constructors, there are two instances of constructor elaboration.

LET: 1. Γ a valid context
2. $A : \text{SET}$

SKETCH: The first one to elaborate the leaf constructors:

$\langle 1 \rangle 1.$ $\vdash \mathbf{Tree}\, A \ni \mathsf{leaf} \overset{C}{\rightsquigarrow} [\mathsf{'leaf} \mapsto 1]$

BY: Example 7.24 applied to rule (CONSTRUCTOR)

SKETCH: The second one to elaborate the node constructor:

$\langle 1 \rangle 2.$ $\vdash \mathbf{Tree}\, A \ni \mathsf{node}\, (a : A)(b : \mathsf{Bool} \to \mathsf{Tree}\, A)$
$\overset{C}{\rightsquigarrow} [\mathsf{'node} \mapsto \Sigma\, A\, \lambda - .\, (\Pi\, \mathsf{Bool}\, \lambda - .\, \mathsf{var}) \times 1]$

BY: Example 7.24 applied to rule (CONSTRUCTOR)

(7.22) **Elaboration of arguments (Figure 7.2d.)**

$$\boxed{\Gamma \vdash l \ni \mathit{args} \overset{A}{\rightsquigarrow} \mathit{code}}$$

This judgment reads as: in context Γ, a constructor's arguments *args* defining the datatype *l* elaborate to a description *code*. The arguments are interpreted as telescopes of Σ-types terminated by a unit type. Hence our translation to Σ 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 Σ code, while in the second case, we elaborate the recursive arguments – via judgment $\overset{R}{\rightsquigarrow}$ – 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 $\Gamma \vdash l \ni \mathit{args} \overset{A}{\rightsquigarrow} \mathit{code}$, then $\Gamma \vdash \mathit{code} : \mathsf{Desc}$

7.24 Example (Elaborating Tree)**.**

LET: 1. Γ a valid context
2. $A : \text{SET}$

SKETCH: Elaborating the arguments of the leaf constructor is trivial:

$\langle 1 \rangle 1.$ $\vdash \mathbf{Tree}\, A \ni \epsilon \overset{A}{\rightsquigarrow} 1$

BY: rule (ARG-END)

SKETCH: As for the node constructor, we obtain its code by elaborating $(a : A)$ to a Σ code, then elaborating the recursive argument, and terminating with the 1 code:

⟨1⟩2. $\vdash \mathbf{Tree}\, A \ni (a\!:\!A)(b\!:\!\mathsf{Bool} \to \mathsf{Tree}\, A) \overset{A}{\leadsto} \Sigma\, A\, \lambda -.\, (\Pi\, \mathsf{Bool}\, \lambda -.\, \mathsf{var}) \times 1$

LET: $a\!:\!A$

⟨2⟩1. $\vdash \mathbf{Tree}\, A \ni (b\!:\!\mathsf{Bool} \to \mathsf{Tree}\, A) \overset{A}{\leadsto} (\Pi\, \mathsf{Bool}\, \lambda -.\, \mathsf{var}) \times 1$

⟨3⟩1. $\vdash \mathbf{Tree}\, A \ni \epsilon \overset{A}{\leadsto} 1$

BY: rule (ARG-END)

⟨3⟩2. Q.E.D.

BY: Example 7.27 and ⟨3⟩1 applied to rule (ARG-REC)

⟨2⟩2. Q.E.D.

BY: ⟨2⟩1 applied to rule (ARG-SIG)

(7.25) **Elaboration of recursive arguments (Figure 7.2e.)**

$$\boxed{\Gamma \vdash l \ni rec \overset{R}{\leadsto} code}$$

This judgment reads as: in context Γ, a recursive argument *rec* defining the datatype *l* elaborates to a description *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 $T \triangleright l$ – and, if so, we generate a var code. We also support constant exponentials of recursive arguments, translating them to the Π 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 $\Gamma \vdash l \ni rec \overset{R}{\leadsto} code$, then $\Gamma \vdash code\!:\!\mathsf{Desc}$

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. Γ a valid context
2. $A\!:\!\mathrm{SET}$
3. $a\!:\!A$

⟨1⟩1. $\vdash \mathbf{Tree}\, A \ni (b\!:\!\mathsf{Bool} \to \mathsf{Tree}\, A) \overset{R}{\leadsto} \Pi\, \mathsf{Bool}\, \lambda -.\, \mathsf{var}$

LET: $-\!:\!\mathsf{Bool}$

⟨2⟩1. $\mathsf{Tree}\, A \triangleright \mathbf{Tree}\, A$

BY: definition of $- \triangleright -$

⟨2⟩2. $\vdash \mathbf{Tree}\, A \ni \mathsf{Tree}\, A \overset{R}{\leadsto} \mathsf{var}$

BY: ⟨2⟩1 and rule (ARG-REC-VAR)

⟨2⟩3. Q.E.D.

BY: ⟨2⟩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 Γ returns an extended context Δ that is valid.

7.29 Theorem (Soundness of elaboration)**.** If $\Gamma \vdash \textbf{data}\ D\ \overrightarrow{(p:P)} : \textsc{Set}\ \textbf{where}\ \mathit{choices} \overset{D}{\rightsquigarrow} \Delta$, then $\Delta \vdash \textsc{valid}$.

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 $\rhd$ 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 $\rhd$ 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:

$$\begin{array}{ll}
\textbf{data}\ \mathsf{Tree}\ [A : \textsc{Set}] : \textsc{Set}\ \textbf{where} & \textbf{data}\ \mathsf{Tree}\ [A : \textsc{Set}] : \textsc{Set}\ \textbf{where} \\
\quad \mathsf{Tree}\ B \ni \mathsf{leaf} & \quad \mathsf{Tree}\ A \ni \mathsf{leaf} \\
\qquad\qquad \mid \mathsf{node}\ (a : A)(b : \mathsf{Bool} \to \mathsf{Tree}\ A) & \qquad\qquad \mid \mathsf{node}\ (a : A)(b : \mathsf{Bool} \to \mathsf{Tree}\ B)
\end{array}$$

In effect, the relation $\rhd$ 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 $\mathbf{Ind}(X : \textsc{Set})\langle \mathcal{C}_0 \mid \ldots \mid \mathcal{C}_n \rangle$ in Coq, the corresponding inductive definition **data** $X : \textsc{Set}$ **where** $X \ni \mathcal{C}_0 \mid \cdots \mid \mathcal{C}_n$ in our system elaborates to a code D having an extensionally equal elimination principle.

Proof. To prove this result, we compute the elaboration of a constructor form $\mathcal{C}$ (Definition 2.2, [Giménez, 1995]). This merely consists in applying the judgment $\overset{\mathsf{C}}{\rightsquigarrow}$: we denote $\lfloor - \rfloor$ the result of this elaboration step. We proceed by induction over the syntax of a constructor form and obtain:

$$\begin{aligned} \lfloor X \rfloor &\mapsto 1 \\ \lfloor (x : M) \to \mathcal{C} \rfloor &\mapsto \Sigma\, M\, \lambda x.\, \lfloor \mathcal{C} \rfloor \\ \lfloor (x : M \to X) \to \mathcal{C} \rfloor &\mapsto (\Pi\, M\, \lambda - .\, \mathsf{var}) \times \lfloor \mathcal{C} \rfloor \end{aligned}$$

We thus get a translation from Giménez's recursive type declarations to a code in our universe:

$$\lfloor \mathbf{Ind}(X : \textsc{Set})\langle \mathcal{C}_0 \mid \ldots \mid \mathcal{C}_n \rangle \rfloor \mapsto \Sigma\, (\mathsf{Fin}\, n)\, \{ i \mapsto \lfloor \mathcal{C}_i \rfloor \}$$

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 $\Box_D\, X \mathrel{\dot{\to}} X \circ \mathsf{in}$ in our system.

□

(7.33) A corollary of this lemma amounts to the completeness of our syntax of datatypes, *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

$$\mathbf{Ind}(X : \textsc{Set})\langle \mathcal{C}_0 \mid \ldots \mid \mathcal{C}_n \rangle$$

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.

Proof. We must show that any `Fixpoint` definition can be implemented using our induction principle. To this end, we use Giménez reduction of `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 *defines* the semantics of our datatypes. This is why the soundness result is crucial.

7.2. Inductive Families

(7.36) 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 though this simpler system. Our syntax is strongly inspired by the one used by Agda and Coq. However, to support computation over indices, we add support for the Epigram-style *by* ($\Leftarrow$) gadget. Our language of inductive definitions is therefore more complex, following the skeleton:

$$\begin{array}{l}
\textbf{data}\ \mathsf{D}\ \overrightarrow{[p\!:\!P]}\overrightarrow{(i\!:\!I)}:\textsc{Set}\ \textbf{where} \\
\quad \begin{array}{llll}
\mathsf{D} & \vec{p}\ \overrightarrow{(i = t_0)} & \ni & c_{0,0}\overrightarrow{(a_{0,0}:T_{0,0})} \\
 & & \mid & \ldots \\
 & \vdots & & \\
\mathsf{D} & \quad \vec{p}\,\vec{i} & \Leftarrow & \mathsf{elim}\ i_m \\
\quad \mathsf{D} & \vec{p}\,\vec{k_0} & \ni & \ldots \\
 & \quad \vdots & &
\end{array}
\end{array}$$

7.37 Definition (Grammar of inductive families)**.** Formally, this notation is captured by the grammar of Figure 7.2. Following convention, $\langle t \rangle$ ranges over terms while $\langle T \rangle$ ranges over types. We artificially segregate parameters from indices, putting parameters first. This is not strictly necessary since our syntax syntactically distinguishes parameters, declared by writing $[p : P]$, from indices, declared by writing $(i : I)$. How-

Elaboration

$$\begin{aligned}
\langle \mathit{data}\rangle &::= \mathbf{data}\ \mathbf{D}\ ([p\!:\!\langle T\rangle])^*('('i\!:\!\langle T\rangle')')^*:\langle T\rangle\ \mathsf{where}\ \langle \mathit{patterns}\rangle \\
\langle \mathit{patterns}\rangle &::= (\langle \mathit{choices}\rangle)^*(\langle \mathit{match}\rangle)^? \\
\langle \mathit{choices}\rangle &::= \langle \mathit{pattern}\rangle \ni \langle \mathit{constructor}\rangle\ ('|'\ \langle \mathit{constructor}\rangle)^* \\
\langle \mathit{match}\rangle &::= \langle \mathit{pattern}\rangle\ \Leftarrow\ \langle t\rangle\{\langle \mathit{patterns}\rangle\} \\
\langle \mathit{pattern}\rangle &::= \mathbf{D}\ (p)^*(\langle t\rangle\ |\ '('i = \langle t\rangle')')^* \\
\langle \mathit{constructor}\rangle &::= \mathbf{con}\ \langle \mathit{arguments}\rangle \\
\langle \mathit{arguments}\rangle &::= ('('x\!:\!\langle T\rangle')')^*
\end{aligned}$$

Figure 7.2.: Grammar of indexed inductive definitions

ever, this choice simplifies the syntax and, later, its elaboration. Our definition of the non-terminal $\langle \mathit{patterns}\rangle$ follows the structure of tagged description (Definition 5.44): it starts with $\langle \mathit{choices}\rangle$ – the constructors available at any index – and ends with $\langle \mathit{match}\rangle$ – 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 $\langle \mathit{match}\rangle$, 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 $\langle \mathit{match}\rangle$ from the grammar. With this minor adjustment, our formalisation of elaboration gives a translation semantics for the inductive definitions of these mainstream theorem provers.

7.2.1. Description labels

(7.39) 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 $\mathsf{IDesc}\ I$, 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 $\langle l \rangle$:

SYNTAX

$$\begin{array}{rcl} \langle l \rangle & ::= & \mathbf{D}\ ([\langle T \rangle])^{*} \langle \mathit{indices} \rangle \qquad\qquad \langle T \rangle ::= \ \ldots \quad | \quad \wr \langle l \rangle \int \\ \langle \mathit{indices} \rangle & ::= & '('i : \langle T \rangle ')' \langle \mathit{indices} \rangle \\ & | & '('i = \langle t \rangle : \langle T \rangle ')' \langle \mathit{indices} \rangle \end{array}$$

We then define the set former $\wr l \int$, the description label of the datatype l (Figure 7.3). Note that indices are presented in a Church-style: they carry a type. We also introduce a meta-operation $[\![-]\!]_D$ that computes the (dependent) product of these types. It computes the index of the description from the label:

META-THEORY

$$[\![\mathbf{D}\ [p_k] \ldots (i_l : T_j) \ldots (i_m = t_m : T_m) \ldots]\!]_D \ \mapsto \ (i_0 : T_0) \times \ldots \times (i_m : T_m) \times \ldots \mathbb{1}$$

The type $\wr l \int$ is inhabited by a return constructor, which takes an enumeration of constructors and their codes. It is then eliminated by call$\wr l \int$ 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

(7.42) 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 7.4c), followed by individual constructors (Figure 7.4d), and finally process the telescope of arguments (Figure 7.4e).

The presence of indices introduces new steps. We constrain and compute over indices through a new top-level judgment (Figure 7.4b). Besides, we must translate the constraints to actual equalities (Figure 7.3h) and pass the correct indices when elaborating a recursive argument (Figure 7.3i).

META-THEORY

$$\frac{\Gamma \vdash [\![l]\!]_D : \textsc{Set}_{\ell}}{\Gamma \vdash \wr l \int : \textsc{Set}_{l+1}}$$

$$\frac{\begin{array}{l}\Gamma \vdash E : \mathsf{EnumU} \\ \Gamma \vdash T : \pi\, E\, \lambda - .\, \mathsf{IDesc}\, [\![l]\!]_D\end{array}}{\Gamma \vdash \mathsf{return}\, E\, T : \wr l \int} \qquad \begin{array}{lcl}\mathsf{call}\wr l \int\, (t : \wr l \int) & : & \mathsf{IDesc}\, [\![l]\!]_D \\ \mathsf{call}\wr l \int\, (\mathsf{return}\, E\, T) & \mapsto & (E, T)\end{array}$$

Figure 7.3.: Description label (indexed)

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:

$$\begin{array}{l}\textbf{data}\ \mathsf{Vec}\ [A : \textsc{Set}](n : \mathsf{Nat}) : \textsc{Set}\ \textbf{where} \\ \quad \begin{array}{lll}\mathsf{Vec}\, A\ \ (n = \mathsf{0}) & \ni & \mathsf{nil} \\ \mathsf{Vec}\, A\ (n = \mathsf{suc}\, n') & \ni & \mathsf{cons}\, (n' : \mathsf{Nat})(a : A)(vs : \mathsf{Vec}\, A\, n')\end{array}\end{array}$$

Our second example consists of the alternative definition of vector, where we compute over the index to determine which constructor to offer:

$$\begin{array}{l}\textbf{data}\ \mathsf{Vec}\ [A : \textsc{Set}](n : \mathsf{Nat}) : \textsc{Set}\ \textbf{where} \\ \quad \begin{array}{lll}\mathsf{Vec}\, A\ \quad n & \Leftarrow & \mathsf{Nat\text{-}case}\, n \\ \quad \mathsf{Vec}\, A\ \quad \mathsf{0} & \ni & \mathsf{nil} \\ \quad \mathsf{Vec}\, A\ (\mathsf{suc}\, m) & \ni & \mathsf{cons}\, (a : A)(vs : \mathsf{Vec}\, A\, m)\end{array}\end{array}$$

(7.44) **Elaboration of inductive families (Figure 7.4a.)**

$$\boxed{\Gamma \vdash \textbf{data}\ D\ \overrightarrow{[p : P]}\overrightarrow{(i : I)} : \textsc{Set}\ \textbf{where}\ patts \overset{D}{\leadsto} \Delta}$$

This judgment reads as: in context Γ, the definition of datatype D, with parameters $\vec{p}$, indices $\vec{i}$, and constructors *choices*, elaborates to a context Δ 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 $\overset{Patts}{\leadsto}$ – by elaborating against the label type corresponding to the given inductive type.

$$\boxed{\langle\Gamma\rangle \vdash \langle \mathit{data}\rangle \overset{D}{\leadsto} \langle\Gamma\rangle}$$

$$\dfrac{\begin{array}{l}\Gamma \vdash \textsc{Set}_1 \ni \overrightarrow{(p : P)}\,\overrightarrow{(i : I)} \to \textsc{Set} \overset{\mathit{Chk}}{\leadsto} \overrightarrow{(p : P')}\,\overrightarrow{(i : I')} \to \textsc{Set} \\ \Gamma; \overrightarrow{p : P'}; \overrightarrow{i : I'} \vdash \mathbf{D}\ \overrightarrow{[p]}\ \overrightarrow{(i)} \ni \mathit{patts} \overset{\mathit{Patts}}{\leadsto} \mathit{code}\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{data}\ D\ \overrightarrow{[p : P]}\,\overrightarrow{(i : I)} : \textsc{Set}\ \textbf{where}\ \mathit{patts} \overset{D}{\leadsto} \\ \qquad \Gamma[D \mapsto \lambda \vec{p} : \vec{P'}.\, \mu\, (\lambda \vec{i} : \vec{I'}.\, \mathsf{call}\lbag D\ \overrightarrow{[p]}\ \overrightarrow{(i)} \rbag\, \mathit{code}) : \overrightarrow{(p : P')}\,\overrightarrow{(i : I')} \to \textsc{Set}]\end{array}}\ (\textsc{Data})$$

(a) Elaboration of definition

$$\boxed{\langle\Gamma\rangle \vdash \langle l\rangle \ni \langle \mathit{patterns}\rangle \overset{\mathit{Patts}}{\leadsto} \langle t\rangle}$$

$$\dfrac{\begin{array}{l} l \supseteq pt_i \overset{I}{\leadsto} l_i \\ \Gamma \vdash l_i \ni cs_i \overset{\mathit{Cs}}{\leadsto} [\{\vec{c_{i,j}}\} \mapsto \{\overrightarrow{c_{i,j} \mapsto as_{i,j}}\}] \\ \\ l \supseteq pt_{i+1} \overset{I}{\leadsto} l_{i+1} \\ \Gamma \vdash e \overset{\mathit{ewm}}{\leadsto} e' \in (\overrightarrow{(\overrightarrow{(x_k : X_k)} \to \lbag l_k \rbag)} \to \lbag l_{i+1} \rbag \\ \Gamma; \overrightarrow{x_k : X_k} \vdash l_k \ni p_k \overset{\mathit{Patts}}{\leadsto} \mathit{code}_k \end{array}}{\Gamma \vdash l \ni \begin{array}{l} pt_0 \ni cs_0 \\ \vdots \\ pt_i \ni cs_i \\ pt_{i+1} \Leftarrow e\,\{\vec{p_k}\} \end{array} \overset{\mathit{Patts}}{\leadsto}\ \mathsf{return}\ \begin{array}{l} \left\{\begin{array}{l} \vec{c_{i,j}} \\ {}'\mathsf{elim} \end{array}\right\} \\ \left\{\begin{array}{l} c_{i,j} \mapsto as_{i,j} \\ {}'\mathsf{elim} \mapsto e'\ \overrightarrow{(\lambda \vec{x_k} : \vec{X_k}.\, \mathsf{call}\lbag l_k \rbag\, \mathit{code}_k)} \end{array}\right\} \end{array}}\ (\textsc{Patterns})$$

(b) Elaboration of patterns

$$\boxed{\langle\Gamma\rangle \vdash \langle l\rangle \ni \langle \mathit{choices}\rangle \overset{\mathit{Cs}}{\leadsto} [\langle t\rangle \mapsto \langle t\rangle]}$$

$$\dfrac{\Gamma \vdash l \ni c_i \overset{C}{\leadsto} [t_i \mapsto \mathit{code}_i]}{\Gamma \vdash l \ni c_0 | \ldots | c_n \overset{\mathit{Cs}}{\leadsto} [\{\vec{t_i}\} \mapsto \left\{\overrightarrow{t_i \mapsto \mathit{code}_i}\right\}]}\ (\textsc{Choices})$$

(c) Elaboration of choices

Figure 7.4.: Elaboration of inductive families

$$\boxed{\langle\Gamma\rangle \vdash \langle l\rangle \ni \langle \mathit{constructor}\rangle \overset{C}{\leadsto} [\langle t\rangle \mapsto \langle t\rangle]}$$

$$\frac{\Gamma \vdash \mathsf{UId} \ni {'}\mathsf{c} \overset{\mathit{Chk}}{\leadsto} c' \qquad \Gamma \vdash l \ni \mathit{args} \overset{A}{\leadsto} \mathit{code}}{\Gamma \vdash l \ni c\, \mathit{args} \overset{C}{\leadsto} [c' \mapsto \mathit{code}]}\ (\textsc{Constructor})$$

(d) Elaboration of constructor

$$\boxed{\langle\Gamma\rangle \vdash \langle l\rangle \ni \langle \mathit{arguments}\rangle \overset{A}{\leadsto} \langle t\rangle}$$

$$\frac{\Gamma \vdash \textsc{Set} \ni T \overset{\mathit{Chk}}{\leadsto} T' \qquad \Gamma; x : T' \vdash l \ni \Delta \overset{A}{\leadsto} \mathit{code}_{\Delta}}{\Gamma \vdash l \ni (x : T)\ \Delta \overset{A}{\leadsto} \Sigma\, T\, \lambda x.\, \mathit{code}_{\Delta}}\ (\textsc{Arg-Sig})$$

$$\frac{\Gamma \vdash l \ni \nabla \overset{R}{\leadsto} \mathit{code}_{\nabla} \qquad \Gamma \vdash l \ni \Delta \overset{A}{\leadsto} \mathit{code}_{\Delta}}{\Gamma \vdash l \ni (x : \nabla)\ \Delta \overset{A}{\leadsto} \mathit{code}_{\nabla} \times \mathit{code}_{\Delta}}\ (\textsc{Arg-Rec})$$

$$\frac{\Gamma \vdash l \overset{\mathit{Eq}}{\leadsto} q}{\Gamma \vdash l \ni \epsilon \overset{A}{\leadsto} q}\ (\textsc{Arg-End})$$

(e) Elaboration of arguments

$$\boxed{\langle\Gamma\rangle \vdash \langle l\rangle \ni \langle \mathit{arguments}\rangle \overset{R}{\leadsto} \langle t\rangle}$$

$$\frac{\Gamma \vdash l \ni T \overset{\mathit{Idx}}{\leadsto} \mathit{is}}{\Gamma \vdash l \ni (x : T) \overset{R}{\leadsto} \mathsf{var}\, \mathit{is}}\ (\textsc{Arg-Rec-Var})$$

$$\frac{\Gamma \vdash \textsc{Set} \ni T \overset{\mathit{Chk}}{\leadsto} T' \qquad \Gamma; x : T' \vdash l \ni \nabla \overset{R}{\leadsto} \mathit{code}_{\nabla}}{\Gamma \vdash l \ni (x : T) \to \nabla \overset{R}{\leadsto} \Pi\, T'\, \lambda x.\, \mathit{code}_{\nabla}}\ (\textsc{Arg-Rec-Exp})$$

$$\frac{\Gamma \vdash \textsc{Set} \ni T \overset{\mathit{Chk}}{\leadsto} T' \qquad \Gamma; x : T' \vdash l \ni \nabla \overset{R}{\leadsto} \mathit{code}_{\nabla}}{\Gamma \vdash l \ni (x : T) \times \nabla \overset{R}{\leadsto} \Sigma\, T'\, \lambda x.\, \mathit{code}_{\nabla}}\ (\textsc{Arg-Rec-Sig})$$

(f) Elaboration of recursive arguments

Figure 7.3.: Elaboration of inductive families

$$\boxed{\langle l\rangle \supseteq \langle T\rangle \overset{I}{\leadsto} \langle l\rangle}$$

$$\frac{}{\mathbf{D} \supseteq D \overset{I}{\leadsto} \mathbf{D}} \qquad \frac{l \supseteq T \overset{I}{\leadsto} l_T}{l\,[p] \supseteq T\,p \overset{I}{\leadsto} l_T\,[p]} \qquad \frac{l \supseteq T \overset{I}{\leadsto} l_T}{l\,(i\!:\!I) \supseteq T\,i \overset{I}{\leadsto} l_T\,(i\!:\!I)}$$

$$\frac{l \supseteq T \overset{I}{\leadsto} l_T}{l\,(i\!:\!I) \supseteq T\,(i\!=\!t\!:\!I) \overset{I}{\leadsto} l_T\,(i\!=\!t\!:\!I)} \qquad \frac{l \supseteq T \overset{I}{\leadsto} l_T}{l\,(i\!=\!t\!:\!I) \supseteq T\,(i\!=\!t\!:\!I) \overset{I}{\leadsto} l_T\,(i\!=\!t\!:\!I)}$$

(g) Pattern validation

$$\boxed{\langle\Gamma\rangle \vdash \langle l\rangle \overset{Eq}{\leadsto} \langle l\rangle}$$

$$\frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \mathbf{D} \overset{Eq}{\leadsto} 1}\ (\textsc{Eq-End}) \qquad \frac{\Gamma \vdash l \overset{Eq}{\leadsto} q}{\Gamma \vdash l\,[p] \overset{Eq}{\leadsto} q}\ (\textsc{Eq-Param})$$

$$\frac{\Gamma \vdash l \overset{Eq}{\leadsto} q}{\Gamma \vdash l\,(i\!:\!I) \overset{Eq}{\leadsto} q}\ (\textsc{Eq-Index}) \qquad \frac{\Gamma \vdash l \overset{Eq}{\leadsto} q \qquad \begin{array}{c} i\!:\!I \in \Gamma \\ \Gamma \vdash I \ni t \overset{Chk}{\leadsto} t' \end{array}}{\Gamma \vdash l\,(i\!=\!t\!:\!I) \overset{Eq}{\leadsto} \Sigma\,(i = t')\,\lambda - .\,q}\ (\textsc{Eq-Cstr})$$

(h) Elaboration of constraints

$$\boxed{\langle\Gamma\rangle \vdash \langle l\rangle \ni \langle T\rangle \overset{Idx}{\leadsto} \langle t\rangle}$$

$$\frac{\Gamma \vdash l \ni t \overset{Idx}{\leadsto} is}{\Gamma \vdash \mathbf{D}\,l \ni D\,t \overset{Idx}{\leadsto} is}\ (\textsc{Eq-Start}) \qquad \frac{\Gamma \vdash \textsc{valid}}{\Gamma \vdash \epsilon \ni \epsilon \overset{Idx}{\leadsto} *}\ (\textsc{Eq-End})$$

$$\frac{\Gamma \vdash l \ni t \overset{Idx}{\leadsto} is}{\Gamma \vdash [p]\,l \ni p\,t \overset{Idx}{\leadsto} is}\ (\textsc{Eq-Param})$$

$$\frac{\begin{array}{c} \Gamma \vdash I \ni i \overset{Chk}{\leadsto} i' \\ \Gamma \vdash l \ni t \overset{Idx}{\leadsto} is \end{array}}{\Gamma \vdash (j\!:\!I)\,l \ni i\,t \overset{Idx}{\leadsto} (i', is)}\ (\textsc{Eq-Index}) \qquad \frac{\begin{array}{c} \Gamma \vdash I \ni i \overset{Chk}{\leadsto} i' \\ \Gamma \vdash l \ni t \overset{Idx}{\leadsto} is \end{array}}{\Gamma \vdash (j\!=\!t\!:\!I)\,l \ni i\,t \overset{Idx}{\leadsto} (i', is)}\ (\textsc{Eq-Cstr})$$

(i) Extraction of indices

Figure 7.2.: Elaboration of inductive families

7.45 Example (Vector, constrained)**.** The elaboration of constraint-based vectors starts as follows:

$$\begin{array}{ll}
\textsc{Let:} & \Gamma \text{ a valid context} \\
& \vdash \textbf{data}\ \mathsf{Vec}\,[A\!:\!\textsc{Set}](n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where}\ [\mathit{patts}_=] \overset{D}{\rightsquigarrow} \\
\langle 1\rangle 1. & \quad \Gamma[\mathsf{Vec} \mapsto \lambda A\!:\!\textsc{Set}.\,\mu\,(\lambda n\!:\!\mathsf{Nat}.\,\mathsf{call}\lbag \mathbf{Vec}\,[A]\ (n\!:\!\mathsf{Nat})\rbag\,[\mathit{code}_=])\ ] \\
& \qquad\qquad : (A\!:\!\textsc{Set})(n\!:\!\mathsf{Nat}) \to \textsc{Set}
\end{array}$$

By: Example 7.51 applied to rule (Data)

where

$$\mathit{patts}_= \triangleq \begin{array}{llll}
\mathsf{Vec}\,A & (n=0) & \ni & \mathsf{nil} \\
\mathsf{Vec}\,A & (n=\mathsf{suc}\,m) & \ni & \mathsf{cons}\,(m\!:\!\mathsf{Nat})(a\!:\!A)(vs\!:\!\mathsf{Vec}\,A\,m)
\end{array}$$

$$\begin{array}{rcl}
\mathit{code}_= & \triangleq & \mathsf{return} \left\{\begin{array}{l} '\mathsf{nil} \\ '\mathsf{cons} \end{array}\right\} \\
& & \left\{\begin{array}{ll} '\mathsf{nil} & \mapsto \Sigma\,(n=0)\,\lambda-.\,1 \\ '\mathsf{cons} & \mapsto \Sigma\,\mathsf{Nat}\,\lambda m.\,\Sigma\,A\,\lambda-.\,\mathsf{var}\,(m,*) \times \Sigma\,(n=\mathsf{suc}\,m)\,\lambda-.\,1 \end{array}\right\}
\end{array}$$

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:

$$\mathit{patts}_\to \triangleq \begin{array}{llll}
\mathsf{Vec}\,A & n & \Leftarrow & \mathsf{Nat\text{-}case}\,n \\
\quad\mathsf{Vec}\,A & 0 & \ni & \mathsf{nil} \\
\quad\mathsf{Vec}\,A & (\mathsf{suc}\,m) & \ni & \mathsf{cons}\,(a\!:\!A)(vs\!:\!\mathsf{Vec}\,A\,m)
\end{array}$$

$$\begin{array}{rcl}
\mathit{code}_\to & \triangleq & \mathsf{return}\,\{'\mathsf{elim}\} \\
& & \left\{ '\mathsf{elim} \mapsto \begin{array}{l}
\quad\mathsf{Nat\text{-}case}\,n\ (\lambda n.\,\lbag \mathbf{Vec}\,[A](n\!:\!\mathsf{Nat})\rbag) \\
(\mathsf{call}\lbag \mathbf{Vec}\,[A]\ (0\!:\!\mathsf{Nat})\rbag\,(\mathsf{return}\,\{'\mathsf{nil}\}\,\{'\mathsf{nil} \mapsto 1\})) \\
(\lambda m.\,\mathsf{call}\lbag \mathbf{Vec}\,[A]\ (\mathsf{suc}\,m\!:\!\mathsf{Nat})\rbag \\
\qquad (\mathsf{return}\,\{'\mathsf{cons}\}\,\{'\mathsf{cons} \mapsto \Sigma\,A\,\lambda-.\,\mathsf{var}\,(m,*) \times 1\}))
\end{array}\right\}
\end{array}$$

(7.47) **Elaboration of patterns (Figure 7.4b.)**

$$\boxed{\Gamma \vdash l \ni \mathit{patts} \overset{\mathit{Patts}}{\rightsquigarrow} \mathit{code}}$$

This judgment reads as: in context Γ, the index patterns *patts* defining the datatype *l* elaborate to a description *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 $\overset{Cs}{\rightsquigarrow}$. The resulting labels are used to elaborate 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 $\overset{ewm}{\rightsquigarrow}$ – 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 $\Gamma \vdash l \ni patts \overset{Patts}{\rightsquigarrow} code$, then $\Gamma \vdash code : \wr l \wr$

To prove this lemma, we need to show that pattern validation respects types:

7.49 Lemma. If $l \supseteq T \overset{I}{\rightsquigarrow} l_T$, then $[\![l]\!]_D = [\![l_T]\!]_D$.

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 $\overset{I}{\rightsquigarrow}$: 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 proceeds over the patterns of constructors, triggering the elaboration of patterns on nil and cons:

LET: 1. Γ a valid context
2. $A : \textsc{Set}$
3. $n : \mathsf{Nat}$

$\langle 1\rangle 1.$ $\vdash \mathbf{Vec}\,[A](n : \mathsf{Nat}) \ni [patts_{=}] \overset{Patts}{\rightsquigarrow} [code_{=}]$

$\langle 2\rangle 1.$ $\mathbf{Vec}\,[A](n : \mathsf{Nat}) \supseteq \mathsf{Vec}\,A\;(n = 0) \overset{I}{\rightsquigarrow} \mathbf{Vec}\,[A](n = 0 : \mathsf{Nat})$
BY: definition of $\overset{I}{\rightsquigarrow}$

$\langle 2\rangle 2.$ $\mathbf{Vec}\,[A](n : \mathsf{Nat}) \supseteq \mathsf{Vec}\,A\;(n = \mathsf{suc}\,m) \overset{I}{\rightsquigarrow} \mathbf{Vec}\,[A](n = \mathsf{suc}\,m : \mathsf{Nat})$
BY: definition of $\overset{I}{\rightsquigarrow}$

$\langle 2\rangle 3.$ Q.E.D.
BY: Example 7.55, $\langle 2\rangle 1$, and $\langle 2\rangle 2$ 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. Γ a valid context

2. $A : \textsc{Set}$

3. $n : \mathsf{Nat}$

$\langle 1\rangle 1.$ $\vdash \mathbf{Vec}\,[A](n : \mathsf{Nat}) \ni [patts_{\rightarrow}] \overset{Patts}{\leadsto} [code_{\rightarrow}]$

$\langle 2\rangle 1.$ $\mathbf{Vec}\,[A](n : \mathsf{Nat}) \supseteq \mathsf{Vec}\,A\,n \overset{I}{\leadsto} \mathbf{Vec}\,[A](n : \mathsf{Nat})$

BY: definition of $\overset{I}{\leadsto}$

$\vdash \mathsf{Nat\text{-}case}\,n \overset{ewm}{\leadsto}$

$\langle 2\rangle 2.$ $\lambda ih_0\, ih_n.\, \mathsf{Nat\text{-}case}\,n\,ih0\,ih_n$

$\in \mathbf{Vec}\,[A](0 : \mathsf{Nat}) \rightarrow ((m : \mathsf{Nat}) \rightarrow \mathbf{Vec}\,[A](\mathsf{suc}\,m : \mathsf{Nat})) \rightarrow \mathbf{Vec}\,[A](n : \mathsf{Nat})$

BY: definition of $\overset{ewm}{\leadsto}$

SKETCH: In turn, this triggers the elaboration of constructor patterns for the nil and cons patterns:

$\langle 2\rangle 3.$ $\vdash \mathbf{Vec}\,[A](0 : \mathsf{Nat}) \ni \mathsf{nil} \overset{Patts}{\leadsto} \mathsf{return}\,\{\mathsf{'nil}\}\,\{\mathsf{'nil} \mapsto 1\}$

$\langle 3\rangle 1.$ $\mathbf{Vec}\,[A](n : \mathsf{Nat}) \supseteq \mathsf{Vec}\,A\,0 \overset{I}{\leadsto} \mathbf{Vec}\,[A](0 : \mathsf{Nat})$

BY: definition of $\overset{I}{\leadsto}$

$\langle 3\rangle 2.$ Q.E.D.

BY: Example 7.56 and $\langle 3\rangle 1$ applied to rule (PATTERNS)

LET: $m : \mathsf{Nat}$

$\langle 2\rangle 4.$ $\vdash \mathbf{Vec}\,[A](\mathsf{suc}\,m : \mathsf{Nat}) \ni \mathsf{cons}\,(a : A)(vs : \mathsf{Vec}\,A\,m)$
$\overset{Patts}{\leadsto} \mathsf{return}\,\{\mathsf{'cons}\}\,\{\mathsf{'cons} \mapsto \Sigma\,A\,\lambda -.\,\mathsf{var}\,(m, *) \times 1\}$

$\langle 3\rangle 1.$ $\mathbf{Vec}\,[A](n : \mathsf{Nat}) \supseteq \mathsf{Vec}\,A\,\mathsf{suc}\,m \overset{I}{\leadsto} \mathbf{Vec}\,[A](\mathsf{suc}\,m : \mathsf{Nat})$

BY: definition of $\overset{I}{\leadsto}$

$\langle 3\rangle 2.$ Q.E.D.

BY: Example 7.56 and $\langle 3\rangle 1$ applied to rule (PATTERNS)

$\langle 2\rangle 5.$ Q.E.D.

BY: $\langle 2\rangle 1$, $\langle 2\rangle 2$, $\langle 2\rangle 3$, and $\langle 2\rangle 4$ applied to rule (PATTERNS)

with $patts_{\rightarrow}$ and $code_{\rightarrow}$ were previously defined.

(7.53) **Elaboration of choices (Figure 7.4c.)**

$$\boxed{\Gamma \vdash l \ni choices \overset{Cs}{\leadsto} [E \mapsto T]}$$

This judgment reads as: in context Γ, the sum of constructor *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 $\Gamma \vdash l \ni \mathit{choices} \overset{Cs}{\rightsquigarrow} [E \mapsto T]$, then $\begin{cases} \Gamma \vdash E : \mathsf{EnumU} \\ \Gamma \vdash T : \mathsf{EnumT}\; E \to \mathsf{IDesc}\; [\![l]\!]_D \end{cases}$

7.55 Example (Vector, constrained)**.** In the particular example of vector, there is only one choice of constructor when $(n = 0)$ – namely nil – and when $(n = \mathsf{suc}\, m)$ – namely cons. Therefore, we obtain the elaboration of choices from the elaboration of the unique constructor, in both cases:

LET: 1. Γ a valid context
2. $A : \mathrm{SET}$
3. $n : \mathsf{Nat}$

$\langle 1 \rangle 1.$ $\vdash \mathbf{Vec}\, [A] (n = 0 : \mathsf{Nat}) \ni \mathsf{nil} \overset{Cs}{\rightsquigarrow} [\{\text{'}\mathsf{nil}\} \mapsto \{\text{'}\mathsf{nil} \;\mapsto\; \Sigma\, (n\, =\, 0)\, \lambda - .\, 1\}]$
BY: Example 7.59 applied to rule (CHOICES)

$\langle 1 \rangle 2.$ $\vdash \mathbf{Vec}\, [A] (n = \mathsf{suc}\, m : \mathsf{Nat}) \ni \mathsf{cons}\, (m : \mathsf{Nat}) (a : A) (vs : \mathsf{Vec}\, A\, m)$
$\overset{Cs}{\rightsquigarrow} [\{\text{'}\mathsf{cons}\} \mapsto \{\text{'}\mathsf{cons} \;\mapsto\; \Sigma\, \mathsf{Nat}\, \lambda m.\, \Sigma\, A\, \lambda - .\, \mathsf{var}\, (m, *) \times \Sigma\, (n\, =\, \mathsf{suc}\, m)\, \lambda - .\, 1\}]$
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. Γ a valid context
2. $A : \mathrm{SET}$
3. $n : \mathsf{Nat}$

$\langle 1 \rangle 1.$ $\vdash \mathbf{Vec}\, [A] (0 : \mathsf{Nat}) \ni \mathsf{nil} \overset{Cs}{\rightsquigarrow} [\{\text{'}\mathsf{nil}\} \mapsto \{\text{'}\mathsf{nil} \;\mapsto\; 1\}]$
BY: Example 7.60 applied to rule (CHOICES)

LET: $m : \mathsf{Nat}$

$\langle 1 \rangle 2.$ $\vdash \mathbf{Vec}\, [A] (\mathsf{suc}\, m : \mathsf{Nat}) \ni \mathsf{cons}\, (a : A) (vs : \mathsf{Vec}\, A\, m)$
$\overset{Cs}{\rightsquigarrow} [\{\text{'}\mathsf{cons}\} \mapsto \{\text{'}\mathsf{cons} \;\mapsto\; \Sigma\, A\, \lambda - .\, \mathsf{var}\, (m, *) \times 1\}]$
BY: Example 7.60 applied to rule (CHOICES)

(7.57) **Elaboration of a constructor (Figure 7.4d.)**

$$\boxed{\Gamma \vdash l \ni c \overset{C}{\rightsquigarrow} [t \mapsto \mathit{code}]}$$

This judgment reads as: in context Γ, the constructor c participating in the definition of datatype l is named t and its arguments are coded by *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 $\Gamma \vdash l \ni c \overset{\mathsf{C}}{\leadsto} [t \mapsto \mathit{code}]$, then $\begin{cases} \Gamma \vdash t : \mathsf{UId} \\ \Gamma \vdash \mathit{code} : \mathsf{IDesc}\, [\![l]\!]_D \end{cases}$

7.59 Example (Vector, constrained)**.** Elaboration of the constructors is straightforward, simply switching to the elaboration of the arguments:

LET: 1. Γ a valid context
2. $A : \textsc{Set}$
3. $n : \mathsf{Nat}$

$\langle 1 \rangle 1.$ $\vdash \mathbf{Vec}\, [A]\ (n = 0 : \mathsf{Nat}) \ni \mathsf{nil} \overset{\mathsf{C}}{\leadsto} [{}'\mathsf{nil} \mapsto \Sigma\, (n = 0)\ \lambda - .\, 1]$

BY: Example 7.63 applied to rule (CONSTRUCTOR)

$\langle 1 \rangle 2.$ $\vdash \mathbf{Vec}\, [A]\ (n = \mathsf{suc}\, m : \mathsf{Nat}) \ni \mathsf{cons}\, (m : \mathsf{Nat})(a : A)(vs : \mathsf{Vec}\, A\, m)$
$\overset{\mathsf{C}}{\leadsto} [{}'\mathsf{cons} \mapsto \Sigma\, \mathsf{Nat}\, \lambda m.\, \Sigma\, A\, \lambda - .\, \mathsf{var}\, (m, *) \times \Sigma\, (n = \mathsf{suc}\, m)\ \lambda - .\, 1]$

BY: Example 7.63 applied to rule (CONSTRUCTOR)

7.60 Example (Vector, computed)**.** Similarly for the alternative definition, we have:

LET: 1. Γ a valid context
2. $A : \textsc{Set}$
3. $n : \mathsf{Nat}$

$\langle 1 \rangle 1.$ $\vdash \mathbf{Vec}\, [A](0 : \mathsf{Nat}) \ni \mathsf{nil} \overset{\mathsf{C}}{\leadsto} [{}'\mathsf{nil} \mapsto 1]$

BY: Example 7.64 applied to rule (CONSTRUCTOR)

LET: $m : \mathsf{Nat}$

$\langle 1 \rangle 2.$ $\vdash \mathbf{Vec}\, [A](\mathsf{suc}\, m : \mathsf{Nat}) \ni \mathsf{cons}\, (a : A)(vs : \mathsf{Vec}\, A\, m)$
$\overset{\mathsf{C}}{\leadsto} [{}'\mathsf{cons} \mapsto \Sigma\, A\, \lambda - .\, \mathsf{var}\, (m, *) \times 1]$

BY: Example 7.64 applied to rule (CONSTRUCTOR)

(7.61) **Elaboration of arguments (Figure 7.4e.)**

$$\boxed{\Gamma \vdash l \ni \mathit{args} \overset{A}{\leadsto} \mathit{code}}$$

This judgment reads as: in context Γ, the telescope of arguments *args* participating in the definition of the datatype l elaborates to a description *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 $\Gamma \vdash l \ni \mathit{args} \overset{A}{\leadsto} \mathit{code}$, then $\Gamma \vdash \mathit{code} : \mathsf{IDesc}\ [\![l]\!]_D$

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. Γ a valid context
2. $A : \textsc{Set}$
3. $n : \mathsf{Nat}$

Sketch: We do so immediately in the nil case:

$\langle 1 \rangle 1.$ $\vdash \mathbf{Vec}\,[A]\ (n = 0 : \mathsf{Nat}) \ni \epsilon \overset{A}{\leadsto} \Sigma\,(n = 0)\ \lambda - .\,1$

By: Example 7.71 applied to rule (Arg-End)

Sketch: While a few steps are necessary to elaborate the arguments in the cons case, including the elaboration of the recursive call:

$\langle 1 \rangle 2.$ $\vdash \mathbf{Vec}\,[A]\ (n = \mathsf{suc}\, m : \mathsf{Nat}) \ni (m : \mathsf{Nat})(a : A)(\mathit{vs} : \mathsf{Vec}\ A\ m)$
$\overset{A}{\leadsto} \Sigma\, \mathsf{Nat}\ \lambda m.\, \Sigma\, A\ \lambda - .\, \mathsf{var}\,(m, *) \times \Sigma\,(n = \mathsf{suc}\, m)\ \lambda - .\,1$

Let: $m : \mathsf{Nat}$

$\langle 2 \rangle 1.$ $\vdash \mathbf{Vec}\,[A]\ (n = \mathsf{suc}\, m : \mathsf{Nat}) \ni (a : A)(\mathit{vs} : \mathsf{Vec}\ A\ m)$
$\overset{A}{\leadsto} \Sigma\, A\ \lambda - .\, \mathsf{var}\,(m, *) \times \Sigma\,(n = \mathsf{suc}\, m)\ \lambda - .\,1$

$\langle 3 \rangle 1.$ $\vdash \mathbf{Vec}\,[A]\ (n = \mathsf{suc}\, m : \mathsf{Nat}) \ni (\mathit{vs} : \mathsf{Vec}\ A\ m)$
$\overset{A}{\leadsto} \mathsf{var}\,(m, *) \times \Sigma\,(n = \mathsf{suc}\, m)\ \lambda - .\,1$

$\langle 4 \rangle 1.$ $\vdash \mathbf{Vec}\,[A]\ (n = \mathsf{suc}\, m : \mathsf{Nat}) \ni \epsilon \overset{A}{\leadsto} \Sigma\,(n = \mathsf{suc}\, m)\ \lambda - .\,1$

By: Example 7.71 applied to rule (Arg-End)

$\langle 4 \rangle 2.$ Q.E.D.

By: Example 7.67, and $\langle 4 \rangle 1$ applied to rule (Arg-Rec)

$\langle 3 \rangle 2.$ Q.E.D.

By: $\langle 3 \rangle 1$ applied to rule (Arg-Sig)

$\langle 2 \rangle 2.$ Q.E.D.

By: $\langle 2 \rangle 1$ applied to rule (Arg-Sig)

7.64 Example (Vector, computed)**.** In the alternative definition, the process is similar:

Let: 1. Γ a valid context
2. $A : \textsc{Set}$
3. $n : \mathsf{Nat}$

$\langle 1 \rangle 1.$ $A : \textsc{Set}, n : \mathsf{Nat} \vdash \mathbf{Vec}\,[A](0 : \mathsf{Nat}) \ni \epsilon \overset{A}{\leadsto} 1$

By: Example 7.72 applied to rule (Arg-End)

Let: $m : \mathsf{Nat}$

$\langle 1 \rangle 2.$ $\vdash \mathbf{Vec}\,[A](\mathsf{suc}\, m : \mathsf{Nat}) \ni (a : A)(\mathit{vs} : \mathsf{Vec}\ A\ m) \overset{A}{\leadsto} \Sigma\, A\ \lambda - .\, \mathsf{var}\,(m, *) \times 1$

$\langle 2 \rangle 1.$ $\vdash \mathbf{Vec}\,[A](\mathsf{suc}\, m : \mathsf{Nat}) \ni (\mathit{vs} : \mathsf{Vec}\ A\ m) \overset{A}{\leadsto} \mathsf{var}\,(m, *) \times 1$

$\langle 3\rangle 1.$ $\vdash$ **Vec** $[A](\mathsf{suc}\, m : \mathsf{Nat}) \ni \epsilon \overset{A}{\leadsto} 1$

BY: Example 7.72 applied to rule (ARG-END)

$\langle 3\rangle 2.$ Q.E.D.

BY: Example 7.68, and $\langle 3\rangle 1$ applied to rule (ARG-REC)

$\langle 2\rangle 2.$ Q.E.D.

BY: $\langle 2\rangle 1$ applied to rule (ARG-SIG)

(7.65) **Elaboration of recursive arguments (Figure 7.4f.)**

$$\boxed{\Gamma \vdash l \ni arg \overset{R}{\leadsto} code}$$

This judgment reads as: in context Γ, the recursive argument *arg* participating in the definition of the datatype l elaborates to a description *code*. Again, the elaboration of recursive arguments follows the one for inductive types (¶ 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 $\overset{Idx}{\leadsto}$.

This step is subject to the following soundness property:

7.66 Lemma. If $\Gamma \vdash l \ni args \overset{R}{\leadsto} code$, then $\Gamma \vdash code : \mathsf{IDesc}\, [\![l]\!]_D$

7.67 Example (Vector, constrained)**.** There is no recursive argument in the nil case. The elaboration of the recursive call in the cons case is a direct application of (ARG-REC-VAR):

LET: 1. Γ a valid context
2. $A : \mathrm{SET}$
3. $n : \mathsf{Nat}$
4. $m : \mathsf{Nat}$

$\langle 1\rangle 1.$ $\vdash$ **Vec** $[A]\ (n = \mathsf{suc}\, m : \mathsf{Nat}) \ni (vs : \mathsf{Vec}\, A\, m) \overset{R}{\leadsto} \mathsf{var}\,(m, *)$

BY: Example 7.75 applied to rule (ARG-REC-VAR)

7.68 Example (Vector, computed)**.** In the alternative definition, the process is similar: no recursive argument in the nil case and a direct appeal to (ARG-REC-VAR) in the cons case:

LET: 1. Γ a valid context
2. $A : \mathrm{SET}$
3. $n : \mathsf{Nat}$
4. $m : \mathsf{Nat}$

$\langle 1\rangle 1.$ $\vdash$ **Vec** $[A](\mathsf{suc}\, m : \mathsf{Nat}) \ni (vs : \mathsf{Vec}\, A\, m) \overset{R}{\leadsto} \mathsf{var}\,(m, *)$

BY: Example 7.76 applied to rule (ARG-REC-VAR)

(7.69) **Elaboration of constraints (Figure 7.3h.)**

$$\boxed{\Gamma \vdash l \overset{Eq}{\rightsquigarrow} q}$$

This judgment reads as: in context Γ, the label l codes the equality constraints q. In order to generate the equality constraints, we simply traverse the label. On an index constraint, we generate the corresponding equation, using whatever propositional equality the system has to offer. On parameters and (unconstrained) indices, we simply go through.

This step satisfies the following property:

7.70 Lemma. If $\Gamma \vdash l \overset{Eq}{\rightsquigarrow} q$, then $\Gamma \vdash q : \mathsf{IDesc}\ [\![l]\!]_D$.

7.71 Example (Vector, constrained)**.** We elaborate the constraints in the nil case – constraining n to 0 – and in the cons case – constraining n to $\mathsf{suc}\, m$:

LET: 1. Γ a valid context
2. $A : \mathrm{SET}$
3. $n : \mathsf{Nat}$

⟨1⟩1. $\vdash \mathbf{Vec}\,[A]\ (n = 0 : \mathsf{Nat}) \overset{Eq}{\rightsquigarrow} \Sigma\ (n = 0)\ \lambda - .\, 1$
  ⟨2⟩1. $\vdash \mathbf{Vec}\,[A] \overset{Eq}{\rightsquigarrow} 1$
    ⟨3⟩1. $\vdash \mathbf{Vec} \overset{Eq}{\rightsquigarrow} 1$
      BY: rule (EQ-END)
    ⟨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-CSTR)

LET: $m : \mathsf{Nat}$

⟨1⟩2. $\vdash \mathbf{Vec}\,[A]\ (n = \mathsf{suc}\, m : \mathsf{Nat}) \overset{Eq}{\rightsquigarrow} \Sigma\ (n = \mathsf{suc}\, m)\ \lambda - .\, 1$
  ⟨2⟩1. $\vdash \mathbf{Vec}\,[A] \overset{Eq}{\rightsquigarrow} 1$
    ⟨3⟩1. $\vdash \mathbf{Vec} \overset{Eq}{\rightsquigarrow} 1$
      BY: rule (EQ-END)
    ⟨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-CSTR)

7.72 Example (Vector, computed)**.** No equations are generated and, indeed, needed for the alternative definition of vectors:

LET: 1. Γ a valid context
2. $A : \textsc{Set}$
3. $n : \mathsf{Nat}$

$\langle 1 \rangle 1.\ \vdash \mathbf{Vec}\,[A](0 : \mathsf{Nat}) \overset{Eq}{\leadsto} 1$
$\langle 2 \rangle 1.\ \vdash \mathbf{Vec}\,[A] \overset{Eq}{\leadsto} 1$
$\langle 3 \rangle 1.\ \vdash \mathbf{Vec} \overset{Eq}{\leadsto} 1$
BY: rule (EQ-END)
$\langle 3 \rangle 2.$ Q.E.D.
BY: $\langle 3 \rangle 1$ applied to rule (EQ-PARAM)
$\langle 2 \rangle 2.$ Q.E.D.
BY: $\langle 2 \rangle 1$ applied to rule (EQ-INDEX)

LET: $m : \mathsf{Nat}$

$\langle 1 \rangle 2.\ \vdash \mathbf{Vec}\,[A](\mathsf{suc}\, m : \mathsf{Nat}) \overset{Eq}{\leadsto} 1$
$\langle 2 \rangle 1.\ \vdash \mathbf{Vec}\,[A] \overset{Eq}{\leadsto} 1$
$\langle 3 \rangle 1.\ \vdash \mathbf{Vec} \overset{Eq}{\leadsto} 1$
BY: rule (EQ-END)
$\langle 3 \rangle 2.$ Q.E.D.
BY: $\langle 3 \rangle 1$ applied to rule (EQ-PARAM)
$\langle 2 \rangle 2.$ Q.E.D.
BY: $\langle 2 \rangle 1$ applied to rule (EQ-INDEX)

(7.73) **Extraction of indices (Figure 7.3i.)**

$$\boxed{\Gamma \vdash l \ni T \overset{Idx}{\leadsto} is}$$

This judgment reads as: in context Γ, the recursive argument T participating in the definition of datatype l is indexed by 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 $\Gamma \vdash l \ni T \overset{Idx}{\leadsto} is$, then $\Gamma \vdash is : [\![l]\!]_D$.

7.75 Example (Vector, constrained)**.** There is only one instance of recursive definition, in the cons case. Its elaboration goes as follows:

LET: 1. Γ a valid context
2. $A : \textsc{Set}$
3. $n : \mathsf{Nat}$

$\langle 1\rangle 1.\ \vdash \mathbf{Vec}\,[A]\,(n = \mathsf{suc}\,m : \mathsf{Nat}) \ni \mathsf{Vec}\,A\,m \overset{\mathit{Idx}}{\rightsquigarrow} (m, *)$

$\langle 2\rangle 1.\ \vdash [A]\,(n = \mathsf{suc}\,m : \mathsf{Nat}) \ni A\,m \overset{\mathit{Idx}}{\rightsquigarrow} (m, *)$

$\langle 3\rangle 1.\ \vdash (n = \mathsf{suc}\,m : \mathsf{Nat}) \ni m \overset{\mathit{Idx}}{\rightsquigarrow} (m, *)$

$\langle 4\rangle 1.\ \vdash \epsilon \ni \epsilon \overset{\mathit{Idx}}{\rightsquigarrow} *$

BY: rule (EQ-END)

$\langle 4\rangle 2.$ Q.E.D.

BY: $\langle 4\rangle 1$ applied to rule (EQ-CSTR)

$\langle 3\rangle 2.$ Q.E.D.

BY: $\langle 3\rangle 1$ applied to rule (EQ-PARAM)

$\langle 2\rangle 2.$ Q.E.D.

BY: $\langle 2\rangle 1$ 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. Γ a valid context
2. $A : \textsc{Set}$
3. $n : \mathsf{Nat}$
4. $m : \mathsf{Nat}$

$\langle 1\rangle 1.\ \vdash \mathbf{Vec}\,[A](\mathsf{suc}\,m : \mathsf{Nat}) \ni \mathsf{Vec}\,A\,m \overset{\mathit{Idx}}{\rightsquigarrow} (m, *)$

$\langle 2\rangle 1.\ \vdash [A]\,(\mathsf{suc}\,m : \mathsf{Nat}) \ni A\,m \overset{\mathit{Idx}}{\rightsquigarrow} (m, *)$

$\langle 3\rangle 1.\ \vdash (\mathsf{suc}\,m : \mathsf{Nat}) \ni m \overset{\mathit{Idx}}{\rightsquigarrow} (m, *)$

$\langle 4\rangle 1.\ \vdash \epsilon \ni \epsilon \overset{\mathit{Idx}}{\rightsquigarrow} *$

BY: rule (EQ-END)

$\langle 4\rangle 2.$ Q.E.D.

BY: $\langle 4\rangle 1$ applied to rule (EQ-CSTR)

$\langle 3\rangle 2.$ Q.E.D.

BY: $\langle 3\rangle 1$ applied to rule (EQ-PARAM)

$\langle 2\rangle 2.$ Q.E.D.

BY: $\langle 2\rangle 1$ 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 $\Gamma \vdash \textbf{data}\ D\ \overrightarrow{[p:P]}\overrightarrow{(i:I)} : \textsc{Set}\ \textbf{where}\ \mathit{choices} \overset{D}{\rightsquigarrow} \Delta$, then $\Delta \vdash \textsc{valid}$.

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 $\Gamma \vdash e \overset{ewm}{\rightsquigarrow} e' \in T$, then $\Gamma \vdash e' : T$), we obtain Lemma 7.48. This gives the desired result.

□

(7.79) **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

$$\mathbf{Ind}(X : \vec{I} \to \textsc{Set})\langle \mathcal{C}_0 \mid \ldots \mid \mathcal{C}_n \rangle$$

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 7.4, Figure 7.3, and Figure 7.2) *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.

7.83 Remark. For simplicity and conciseness, we shall define these mechanisms over our universe of inductive types, 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`

(7.84) 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,

$$
\frac{\begin{array}{l}\Gamma \vdash \textsc{Set}_1 \ni \overrightarrow{(p\!:\!P)} \to \textsc{Set} \overset{Chk}{\leadsto} \overrightarrow{(p\!:\!P')} \to \textsc{Set} \\ \Gamma; \overrightarrow{p\!:\!P'} \vdash \mathbf{D}\,\vec{p} \ni \mathit{choices} \overset{Cs}{\leadsto} \mathit{code}\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{data}\ D\ \overrightarrow{[p\!:\!P]}\!:\!\textsc{Set}\ \textbf{where}\ \mathit{choices} \\ \quad \overset{Case}{\leadsto}\ \Gamma[D\text{-}\mathsf{Desc} \mapsto \mathsf{call} \wr \mathbf{D}\,\vec{p} \int \mathit{code}\!:\!\mathsf{Desc}] \\ \qquad [D\text{-}\mathsf{case} \mapsto \mathsf{case}\, D\text{-}\mathsf{Desc}\!:\!(P\!:\!\mu D\text{-}\mathsf{Desc} \to \textsc{Set}) \to \\ \qquad\qquad\qquad\qquad (\mathit{cases}\!:\!((d\!:\![\![D\text{-}\mathsf{Desc}]\!]\ (D\text{-}\mathsf{Desc})) \to P(\mathsf{in}d))) \\ \qquad\qquad\qquad\qquad (x\!:\!\mu D\text{-}\mathsf{Desc}) \to P x]\end{array}}
$$

Figure 7.3.: Elaboration of case analysis

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:

$$
\begin{array}{l}
\mathsf{case}\,(D\!:\!\mathsf{Desc})(P\!:\!\mu D \to \textsc{Set})(\mathit{cases}\!:\!((d\!:\![\![D]\!]\ (\mu D)) \to P(\mathsf{in}d)))(x\!:\!\mu D)\ :\ P x \\
\mathsf{case}\, D\ P\ \mathit{cases}\ x\ \mapsto\ \mathsf{induction}\ (\lambda d.\, \lambda -.\, \mathit{cases}\ d)\ x
\end{array}
$$

(7.86) To account for case analysis during elaboration, we simply add a judgment $\overset{Case}{\leadsto}$ to our system (Figure 7.3). The judgment has the form

$$\boxed{\Gamma \vdash \textbf{data}\ D\ \overrightarrow{(p\!:\!P)}\!:\!\textsc{Set}\ \textbf{where}\ \mathit{choices} \overset{Case}{\leadsto} \Delta}$$

This judgment reads as: in context Γ, the datatype declaration of D elaborates to an extended context Δ, which defines the code of this declaration, denoted D-Desc, together with a case analysis operator, denoted D-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 case function to this datatype by partial application.

7.87 Remark. In Figure 7.3, we extend the context with a definition D-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-Desc is in context for a datatype definition D.

(7.88) 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 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):

$$\begin{array}{llll}
\mathsf{NoConfusion} & (x : \mu D) \quad\;\; (y : \mu D) & : \textsc{Set}_1 & \\
\mathsf{NoConfusion} & (\mathsf{in}\,(c_x, a_x))\;(\mathsf{in}\,(c_y, a_y)) & \mid \mathsf{decideEq\text{-}EnumT}\; c_x\; c_y & \\
\mathsf{NoConfusion} & (\mathsf{in}\,(c_x, a_x))\;(\mathsf{in}\,(c_x, a_y)) & \mid \;\; \mathsf{equal\; refl} & \mapsto (P : \textsc{Set}) \to (a_x = a_y \to P) \to P \\
\mathsf{NoConfusion} & (\mathsf{in}\,(c_x, a_x))\;(\mathsf{in}\,(c_y, a_y)) & \mid \mathsf{not\text{-}equal}\; q & \mapsto (P : \textsc{Set}) \to P
\end{array}$$

The function 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 decideEq-EnumT $c_x\, c_y$ under the scrutiny of 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:

$$\begin{array}{lllll}
\mathsf{noConfusion} & (x : \mu D)\;(y : \mu D)\;(q : x = y) & : \mathsf{NoConfusion}\; x\; y & & \\
\mathsf{noConfusion} & (\mathsf{in}\,(c_x, a_x))\;(\mathsf{in}\,(c_y, a_y))\;\; q & \mid \mathsf{decideEq\text{-}EnumT}\; c_x\; c_y & & \\
\mathsf{noConfusion} & (\mathsf{in}\,(c_x, a_x))\;(\mathsf{in}\,(c_x, a_y))\;\; q & \mid & \mathsf{equal\; refl} & \mapsto \lambda P.\, \lambda rec.\, rec\; q \\
\mathsf{noConfusion} & (\mathsf{in}\,(c_x, a_x))\;(\mathsf{in}\,(c_y, a_y))\;\; q & \mid & \mathsf{not\text{-}equal}\; neq & \mapsto \lambda P.\, \mathbb{0}\text{-}\mathsf{elim}\;(neq\; q)
\end{array}$$

□

(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.

7.3.2. Deriving operations on datatypes

MODEL: `Chapter7.Derivable`

(7.91) Another extension of our system is a generic `deriving` mechanism. In the Haskell language, we can write a definition such as

$$\begin{array}{l} \textbf{data}\ \mathsf{Nat}\!:\!\textsc{Set}\ \textbf{where} \\ \quad \mathsf{Nat}\ \ni\ 0 \\ \qquad\quad \mid\ \mathsf{suc}\,(n\!:\!\mathsf{Nat}) \\ \qquad \textbf{deriving}\ \mathsf{Eq} \end{array}$$

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.

(7.92) 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:

$$\begin{array}{l} \mathsf{Derivable}\,(P\!:\!\mathsf{Desc}\to\textsc{Set})\ :\ \textsc{Set}_1 \\ \mathsf{Derivable}\,P\ \mapsto\ \left\{\begin{array}{ll} \mathsf{subDesc} & :\mathsf{Desc}\to\textsc{Set}_1 \\ \mathsf{membership} & :(D\!:\!\mathsf{Desc})\to\mathsf{Decidable}\,(\mathsf{subDesc}\,D) \\ \mathsf{derive} & :\mathsf{subDesc}\,D\to P\,D \end{array}\right. \end{array}$$

That is, one must specify a subuniverse of descriptions, subDesc. The membership of a description to this subuniverse must be a decidable property, as witnessed by membership. Finally, in order to actually derive the property P, one must provide a 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

$$\mathit{decide}_P : (x : X) \to \mathsf{Decidable}\,(P\; x)$$

where Decidable is defined by:

$$\begin{array}{l} \textbf{data}\; \mathsf{Decidable}\,[A : \textsc{Set}] : \textsc{Set}\; \textbf{where} \\ \quad \mathsf{Decidable}\; A \ni \mathsf{yes}\,(a : A) \\ \qquad\qquad\qquad \mid\; \mathsf{no}\,(na : A \to \mathbb{0}) \end{array}$$

(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 → $\textsc{Set}$, 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 → $\textsc{Set}_1$. 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

$$\mathsf{membershipEq} : (D : \mathsf{Desc}) \to \mathsf{Decidable}\,(\mathsf{eqDesc}\; D)$$

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

$$\mathsf{deriveEq} : \mathsf{eqDesc}\; D \to (x\; y : \mu\; D) \to \mathsf{Decidable}\,(x = y)$$

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

[1] We refer our reader to `Chapter7.Derivable.Equality` for the implementation details.

objects of finite cardinality for equality. Put together, this populates a record:

$$\mathsf{Eq} : \mathsf{Derivable}\ (\lambda D.\ (x\, y : \mu\, D) \to \mathsf{Decidable}\ (x = y))$$

(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 decision procedure

$$\begin{array}{lll} \mathsf{Nat\text{-}eq}\ (x\, y : \mathsf{Nat}) & : & \mathsf{Decidable}\ (x = y) \\ \mathsf{Nat\text{-}eq}\quad x\, y & \mapsto & \mathsf{deriveEq}\ (\mathsf{witness}\ (\mathsf{membershipEq}\ \mathsf{NatD})\ *) \end{array}$$

without any input from the user but the `deriving` Eq clause. Here, witness is a library function that extracts the witness from a true decidable property:

$$\begin{array}{llll} \mathsf{witness}\ (w : \mathsf{Decidable}\ A)\ (t : \mathsf{True}\ w) & : & A & \\ \mathsf{witness}\quad (\mathsf{yes}\ a)\quad * & \mapsto & a & \textbf{where} \\ \quad \mathsf{True}\ (w : \mathsf{Decidable}\ A) & : & \mathrm{S}\small{\mathrm{ET}} & \\ \quad \mathsf{True}\quad (\mathsf{yes}\ a) & \mapsto & \mathbb{1} & \\ \quad \mathsf{True}\quad (\mathsf{no}\ na) & \mapsto & \mathbb{0} & \end{array}$$

Applied to NatD, the function membershipEq computes a positive witness that we can simply extract.

(7.99) 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:

$$\langle \mathit{data} \rangle ::= \textbf{data D}\ \langle \mathit{paramTel} \rangle : \langle T \rangle\ \textbf{where}\ \langle \mathit{choices} \rangle\ \textbf{deriving}\ \langle t \rangle$$

We process these definitions through the judgment:

$$\boxed{\Gamma \vdash \textbf{data}\ D\ \overrightarrow{(p : P)} : \mathrm{SET}\ \textbf{where}\ \mathit{choices}\ \textbf{deriving}\ P \overset{\mathit{Derive}}{\rightsquigarrow} \Delta}$$

This judgment reads as: in context Γ, the datatype definition deriving a property P elaborates to a context Δ, which either defines a function decide-P that decides P, or is

$$\frac{\Gamma \vdash \mathit{prop} \overset{\mathit{Syn}}{\rightsquigarrow} \mathit{prop}' \in \mathsf{Derivable}\, P}{\begin{array}{l} \Gamma \vdash \textbf{data}\ D\ [\overrightarrow{p : P}] : \textsc{Set}\ \textbf{where}\ \mathit{choices}\ \textbf{deriving}\ \mathit{prop} \\ \overset{\mathit{Derive}}{\rightsquigarrow} \Gamma\ \begin{array}{lll} [\mathsf{decide\text{-}P} & \mapsto & \mathit{prop}'.\mathsf{derive}\, (\mathsf{witness}\, (\mathit{prop}'.\mathsf{membership}\, D\text{-}\mathsf{Desc})\, *) \\ & : & (x : \mu\, D\text{-}\mathsf{Desc}) \to P\, x] \end{array} \end{array}}$$

Figure 7.4.: Elaboration of `deriving`

invalid – *i.e.* $\Delta \nvdash \textsc{valid}$. The inference rule (Figure 7.4) consists simply in extending Γ with a function extracting the decidability witness, as we did in Example 7.98, thus obtaining our decision procedure.

7.100 Remark (type checking and error management)**.** We remark that the witness extraction might fail: if the decision procedure rejects the datatype, the witness function asks for an inhabitant of $\mathbb{0}$. Thus, our generated term, providing an inhabitant of $\mathbb{1}$, does not type check. This behavior is the expected one: we want to reject **deriving** statements 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.

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 language 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.

8. Ornaments

(8.1) 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).

(8.2) 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 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.

(8.3) 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:

$$\begin{array}{l} \textbf{data}\ \mathsf{Bool} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Bool} \ni \mathsf{true} \\ \qquad\quad\ \mid \mathsf{false} \end{array} \quad \overset{\mathsf{MaybeO}\ A}{\Rightarrow} \quad \begin{array}{l} \textbf{data}\ \mathsf{Maybe}\ [A : \textsc{Set}] : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Maybe}\ A \ni \mathsf{just}\ (a : A) \\ \qquad\qquad\quad \mid \mathsf{nothing} \end{array}$$

8.5 Example (Ornament: Extending numbers to lists)**.** Or we can extend natural numbers to lists by inserting an $a : A$ at each successor node:

$$\begin{array}{l} \textbf{data}\ \mathsf{Nat} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Nat} \ni 0 \\ \qquad\quad \mid \mathsf{suc}\ (n : \mathsf{Nat}) \end{array} \quad \overset{\mathsf{ListO}\ A}{\Rightarrow} \quad \begin{array}{l} \textbf{data}\ \mathsf{List}\ [A : \textsc{Set}] : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{List}\ A \ni \mathsf{nil} \\ \qquad\qquad \mid \mathsf{cons}\ (a : A)(as : \mathsf{List}\ A) \end{array}$$

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:

$$\begin{array}{l} \textbf{data}\ \mathsf{Nat} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Nat} \ni 0 \\ \qquad\quad \mid \mathsf{suc}\,(n : \mathsf{Nat}) \end{array} \quad \overset{\mathsf{ContextO}}{\Rightarrow} \quad \begin{array}{l} \textbf{data}\ \mathsf{Context} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Context} \ni \epsilon \\ \qquad\qquad\quad \mid (\Gamma : \mathsf{Context})\,;(T : \mathsf{Type}) \end{array}$$

(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:

$$\begin{array}{l} \textbf{data}\ \mathsf{Nat} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Nat} \ni 0 \\ \qquad\quad \mid \mathsf{suc}\,(n : \mathsf{Nat}) \end{array} \quad \overset{\mathsf{FinO}}{\Rightarrow} \quad \begin{array}{l} \textbf{data}\ \mathsf{Fin}\,(n : \mathsf{Nat}) : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Fin}\,(n = \mathsf{suc}\,n') \ni \mathsf{f0}\,(n' : \mathsf{Nat}) \\ \qquad\qquad\qquad\quad \mid \mathsf{fsuc}\,(n' : \mathsf{Nat})(k : \mathsf{Fin}\,n') \end{array}$$

The datatype $\mathsf{Fin}\,n$ 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 and refining numbers to vectors)**.** We extend natural numbers to lists while refining the index to represent the length of the list thus constructed:

$$\begin{array}{l} \textbf{data}\ \mathsf{Nat} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Nat} \ni 0 \\ \qquad\quad \mid \mathsf{suc}\,(n : \mathsf{Nat}) \end{array} \quad \overset{\mathsf{VecO}\,A}{\Longrightarrow} \quad \begin{array}{l} \textbf{data}\ \mathsf{Vec}\,[A : \textsc{Set}](n : \mathsf{Nat}) : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Vec}\,A\ \ (n = 0)\ \ \ni \mathsf{nil} \\ \quad \mathsf{Vec}\,A\,(n = \mathsf{suc}\,n') \ni \mathsf{cons}\,(n' : \mathsf{Nat})(a : A)(vs : \mathsf{Vec}\,A\,n') \end{array}$$

(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:

$$\begin{array}{l} \textbf{data}\ \mathsf{Nat} : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Nat} \ni 0 \\ \qquad\quad \mid \mathsf{suc}\,(n : \mathsf{Nat}) \end{array} \quad \not\Rightarrow \quad \begin{array}{l} \textbf{data}\ \mathsf{Tree}\,[A : \textsc{Set}] : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Tree}\,A \ni \mathsf{leaf} \\ \qquad\qquad \mid \mathsf{node}\,(lb : \mathsf{Tree}\,A)(a : A)(rb : \mathsf{Tree}\,A) \end{array}$$

(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 $[\text{SET}/I, \text{SET}]^J$ (Definition 8.20).

(8.16) Our grammar of ornaments is similar to the original one: we can *copy* the base datatype (with the codes 1, Π, Σ, $\times$, and σ), *extend* it by inserting sets (with the code insert), and *refine* the indexing subject to the relation imposed by u (with the code 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 insert codes into Σ codes. In the delete case, no Σ code is generated: we use the witness to compute the extension of the rest of the ornament.

8.18 Remark (Notation)**.** We overload the IDesc constructor names in the definition of

$$
\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ \mathsf{Orn}\ (D\,{:}\,\mathsf{IDesc}\,K)[u\,{:}\,I \to K]\,{:}\,\textsc{Set}_1\ \textbf{where}}\\
\multicolumn{4}{l}{\quad \textit{– Extend with S:}}\\
\mathsf{Orn}\quad D & u & \ni & \mathsf{insert}\,(S\,{:}\,\textsc{Set})(D^{+}\,{:}\,S \to \mathsf{Orn}\,D\,u)\\
\multicolumn{4}{l}{\quad \textit{– Refine the index:}}\\
\mathsf{Orn}\ (\mathsf{var}\,k) & u & \ni & \mathsf{var}\,(i\,{:}\,u^{\ -1}\,k)\\
\multicolumn{4}{l}{\quad \textit{– Copy the original:}}\\
\mathsf{Orn}\quad 1 & u & \ni & 1\\
\mathsf{Orn}\ (A \times B) & u & \ni & (A^{+}\,{:}\,\mathsf{Orn}\,A\,u) \times (B^{+}\,{:}\,\mathsf{Orn}\,B\,u)\\
\mathsf{Orn}\ (\sigma\,E\,T) & u & \ni & \sigma\,(T^{+}\,{:}\,(e\,{:}\,E) \to \mathsf{Orn}\,(\mathsf{switch}\,T\,e)\,u)\\
\mathsf{Orn}\ (\Pi\,S\,T) & u & \ni & \Pi\,(T^{+}\,{:}\,(s\,{:}\,S) \to \mathsf{Orn}\,(T\,s)\,u)\\
\mathsf{Orn}\ (\Sigma\,S\,T) & u & \ni & \Sigma\,(T^{+}\,{:}\,(s\,{:}\,S) \to \mathsf{Orn}\,(T\,s)\,u)\\
\multicolumn{4}{l}{\quad \textit{– Delete S:}}\\
 & & \mid & \mathsf{delete}\,(s\,{:}\,S)(T^{+}\,{:}\,\mathsf{Orn}\,(T\,s)\,u)
\end{array}
$$

(a) Code

$$
\begin{array}{lcl}
[\![(O\,{:}\,\mathsf{Orn}\,D\,u)]\!]_{\mathsf{orn}} & : & \mathsf{IDesc}\,I\\
[\![\mathsf{insert}\,S\,D^{+}]\!]_{\mathsf{orn}} & \mapsto & \Sigma\,S\,\lambda s.\,[\![D^{+}\,s]\!]_{\mathsf{orn}}\\
[\![\mathsf{var}\,(\mathsf{inv}\,i)]\!]_{\mathsf{orn}} & \mapsto & \mathsf{var}\,i\\
[\![1]\!]_{\mathsf{orn}} & \mapsto & 1\\
[\![A^{+} \times B^{+}]\!]_{\mathsf{orn}} & \mapsto & [\![A^{+}]\!]_{\mathsf{orn}} \times [\![B^{+}]\!]_{\mathsf{orn}}\\
[\![\sigma\,T^{+}]\!]_{\mathsf{orn}} & \mapsto & \sigma\,E\,\lambda e.\,[\![\mathsf{switch}\,T^{+}\,e]\!]_{\mathsf{orn}}\\
[\![\Pi\,T^{+}]\!]_{\mathsf{orn}} & \mapsto & \Pi\,S\,\lambda s.\,[\![T^{+}\,s]\!]_{\mathsf{orn}}\\
[\![\Sigma\,T^{+}]\!]_{\mathsf{orn}} & \mapsto & \Sigma\,S\,\lambda s.\,[\![T^{+}\,s]\!]_{\mathsf{orn}}\\
[\![\mathsf{delete}\,s\,T^{+}]\!]_{\mathsf{orn}} & \mapsto & [\![T^{+}]\!]_{\mathsf{orn}}
\end{array}
$$

(b) Interpretation

Figure 8.1.: Universe of ornaments

the Orn universe. Indeed, the ornamental codes 1, ×, σ, Π, and Σ merely duplicate the underlying description. Similarly, the 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.

(8.19) We then define the ornament of an indexed description by lifting the ornament code to a set indexed by J. An ornament is defined upon a datatype – specified by a description $D : \mathsf{func}\, K\, L$ – and indices are then refined relatively to two reindexing functions $u : I \to K$ and $v : J \to L$.

8.20 Definition (Ornament)**.** An ornament is the lifting of an Orn code to a J-indexed set:

$$\begin{array}{ll}
\mathsf{orn}\; (D : \mathsf{func}\, K\, L)\; (u : I \to K)\; (v : J \to L) \;:\; \textsc{Set}_1 & [\![(o : \mathsf{orn}\, D\, u\, v)]\!]_{\mathsf{orn}} \;:\; \mathsf{func}\, I\, J \\
\mathsf{orn}\, D\, u\, v \;\mapsto\; (j : J) \to \mathsf{Orn}\, (D\, (v\, j))\, u & [\![o]\!]_{\mathsf{orn}} \;\mapsto\; \lambda j.\, [\![o\, j]\!]_{\mathsf{orn}}
\end{array}$$

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 $[x \triangleq s]$. We now rephrase our earlier examples using this language.

Just as the datatype declaration syntax was elaborated to 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.

from the Booleans by inserting an extra $a : A$ in the true case:

$$\begin{array}{l} \textbf{data}\ \mathsf{Maybe}\ [A : \textsc{Set}]\ \textbf{from}\ \mathsf{Bool}\ \textbf{where} \\ \quad \mathsf{Maybe}\ A\ \ni \mathsf{just}\ [a : A] \\ \qquad\qquad\quad \mid \mathsf{nothing} \end{array}$$

$$\wr$$

$$\begin{array}{lll} \mathsf{MaybeO}\ (A : \textsc{Set}) & : & \mathsf{orn}\ \mathsf{BoolD}\ \mathsf{id}\ \mathsf{id} \\ \mathsf{MaybeO}\quad A & \mapsto & \lambda *.\, \Sigma \left\{ \begin{array}{l} \mathsf{'true} \mapsto \mathsf{insert}\ A\ \lambda - .\, 1 \\ \mathsf{'false} \mapsto 1 \end{array} \right\} \end{array}$$

The reader will check that the interpretation of this ornament (by $\llbracket - \rrbracket_{\mathsf{orn}}$) followed by the interpretation of the resulting description (by $\llbracket - \rrbracket$) yields the signature functor of the option type:

$$\llbracket \llbracket \mathsf{MaybeO}\ A \rrbracket_{\mathsf{orn}} \rrbracket\ X \cong 1 + A$$

8.23 Remark. The astute reader will have noticed that the constructor names of the Maybe type have been lost in the ornamentation. In the present system, its constructors are false : Maybe A and true : $A \rightarrow$ 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 $\llbracket - \rrbracket_{\mathsf{orn}}$ to denote both the description and the interpretation of that description. For instance, we write the above isomorphism as follows:

$$\llbracket \mathsf{MaybeO}\ A \rrbracket_{\mathsf{orn}} [X] \cong 1 + A$$

8.25 Example (Ornamenting natural numbers to lists)**.** We obtain lists from natural numbers with the following ornament:

$$\begin{array}{l}
\textbf{data}\ \mathsf{List}\ [A\!:\!\textsc{Set}]\ \textbf{from}\ \mathsf{Nat}\ \textbf{where} \\
\quad \mathsf{List}\ A\ \ni\ \mathsf{nil} \\
\qquad\qquad \mid\ \mathsf{cons}\ [a\!:\!A](as\!:\!\mathsf{List}\ A)
\end{array}$$

$$\wr$$

$$\begin{array}{lll}
\mathsf{ListO}\ (A\!:\!\textsc{Set}) & : & \mathsf{orn}\ \mathsf{NatD}\ \mathsf{id}\ \mathsf{id} \\
\mathsf{ListO}\qquad A & \mapsto & \lambda *.\ \Sigma \left\{ \begin{array}{ll} '0 & \mapsto 1 \\ '\mathsf{suc} & \mapsto \mathsf{insert}\ A\ \lambda - .\ \mathsf{var} * \end{array} \right\}
\end{array}$$

We check that we obtain the signature functor of lists:

$$[\![\mathsf{ListO}\ A]\!]_{\mathsf{orn}}[X] \cong 1 + A \times X$$

8.26 Example (Ornamenting natural numbers to finite sets)**.** We obtain finite sets by inserting a number $n' : \mathsf{Nat}$, constraining the index n to $\mathsf{suc}\ n'$, and – in the recursive case – indexing at n':

$$\begin{array}{l}
\textbf{data}\ \mathsf{Fin}\ (n\!:\!\mathsf{Nat})\ \textbf{from}\ \mathsf{Nat}\ \textbf{where} \\
\quad \mathsf{Fin}\ n\ \ni\ \mathsf{f0}\ [n'\!:\!\mathsf{Nat}][q\!:\!n = \mathsf{suc}\ n'] \\
\qquad\qquad \mid\ \mathsf{fsuc}\ [n'\!:\!\mathsf{Nat}][q\!:\!n = \mathsf{suc}\ n'](k\!:\!\mathsf{Fin}\ n')
\end{array}$$

$$\wr$$

$$\begin{array}{lll}
\mathsf{FinO} & : & \mathsf{orn}\ \mathsf{NatD}\ (\lambda n.\ *)\ (\lambda n.\ *) \\
\mathsf{FinO} & \mapsto & \lambda n.\ \mathsf{insert}\ \mathsf{Nat}\ \lambda n'. \\
 & & \qquad \mathsf{insert}\ (n = \mathsf{suc}\ n')\lambda - . \\
 & & \qquad \Sigma \left\{ \begin{array}{ll} '0 & \mapsto 1 \\ '\mathsf{suc} & \mapsto \mathsf{var}\ n' \end{array} \right\}
\end{array}$$

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:

$$\begin{array}{lcll}
\Box_{(D : \mathsf{IDesc}\,K)} & (xs : [\![D]\!]\, X) & : & \mathsf{Orn}\, D\, \pi_0 \\
\Box_{1} & * & \mapsto & 1 \\
\Box_{\mathsf{var}\, k} & x & \mapsto & \mathsf{var}\,(\mathsf{inv}\,(k, x)) \\
\Box_{A \times B} & (a, b) & \mapsto & \Box_{A}\, a \times \Box_{B}\, b \\
\Box_{\sigma\, E\, T} & (e, xs) & \mapsto & \mathsf{delete}\, e\, (\Box_{\mathsf{switch}\, T\, e}\, xs) \\
\Box_{\Sigma\, S\, T} & (s, xs) & \mapsto & \mathsf{delete}\, s\, (\Box_{T\, s}\, xs) \\
\Box_{\Pi\, S\, T} & f & \mapsto & \Pi\, \lambda s.\, \Box_{T\, s}\, (f\, s)
\end{array}$$

This lifts pointwise to indexed descriptions, giving an ornament of type:

$$\Box_{(D : \mathsf{func}\, K\, L)} : \mathsf{orn}\, D\, (\pi_0 : (k : K) \times X\, k \to K)\, (\pi_0 : (l : L) \times [\![D]\!]\, X\, l \to L)$$

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 $\Box_D$ 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-preserving.

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:

$$\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ \mathsf{Vec'}\ [A\!:\!\textsc{Set}](n\!:\!\mathsf{Nat})\ \textbf{from}\ \mathsf{Vec}\ A\ n\ \textbf{where}} \\
\quad \mathsf{Vec'}\ A & n & \Leftarrow & \mathsf{Nat\text{-}case}\ n \\
\quad\quad \mathsf{Vec'}\ A & 0 & \ni & \mathsf{nil} \\
\quad\quad \mathsf{Vec'}\ A & (\mathsf{suc}\ n) & \ni & \mathsf{cons}\ [n' \triangleq n](a\!:\!A)(vs\!:\!\mathsf{Vec'}\ A\ n)
\end{array}$$

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:

$$\begin{array}{llll}
\multicolumn{4}{l}{\textbf{data}\ \mathsf{Fin'}\ (n\!:\!\mathsf{Nat})\ \textbf{from}\ \mathsf{Fin}\ n\ \textbf{where}} \\
\quad \mathsf{Fin'}\ \ 0 & \ni & [b\!:\!\mathbb{0}] & \quad\textit{– no constructor} \\
\quad \mathsf{Fin'}\ (\mathsf{suc}\ n) & \ni & \mathsf{f0}\ [n' \triangleq n] & \\
 & \mid & \mathsf{fsuc}\ [n' \triangleq n](k\!:\!\mathsf{Fin'}\ n') &
\end{array}$$

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`

(8.31) 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 $O : \mathsf{Orn}\ D\ u$, there is a function – actually, a natural transformation – projecting the ornamented functor down to the original, unornamented functor:

$$\begin{array}{lccl}
\mathsf{forgetNT}\ (O\!:\!\mathsf{Orn}\ D\ u)\ (xs\!:\![\![O]\!]_{\mathsf{orn}}\ (X\circ u)) & & : & [\![D]\!]\ X \\
\mathsf{forgetNT}\ (\mathsf{insert}\ S\ D^{+}) & (s, xs) & \mapsto & \mathsf{forgetNT}\ (D^{+}\ s)\ xs \\
\mathsf{forgetNT}\ \ (\mathsf{var}\ (\mathsf{inv}\ i)) & xs & \mapsto & xs \\
\mathsf{forgetNT}\ \qquad 1 & * & \mapsto & * \\
\mathsf{forgetNT}\ \ (O_1 \times O_2) & (t_1, t_2) & \mapsto & (\mathsf{forgetNT}\ O_1\ t_1, \mathsf{forgetNT}\ O_2\ t_2) \\
\mathsf{forgetNT}\ \quad (\sigma\, O) & (k, xs) & \mapsto & (k, \mathsf{forgetNT}\ (\mathsf{switch}\ O\ k)\ xs) \\
\mathsf{forgetNT}\ \quad (\Sigma\, O) & (s, xs) & \mapsto & (s, \mathsf{forgetNT}\ (O\ s)\ xs) \\
\mathsf{forgetNT}\ \quad (\Pi\, O) & f & \mapsto & \lambda s.\, \mathsf{forgetNT}\ (O\ s)\ (f\ s) \\
\mathsf{forgetNT}\ \ (\mathsf{delete}\ s\ O) & xs & \mapsto & (s, \mathsf{forgetNT}\ O\ xs)
\end{array}$$

This function then lifts pointwise to a function on ornaments:

$$\mathsf{forgetNT} : (o\!:\!\mathsf{orn}\ D\ u\ v) \to [\![o]\!]\ (X\circ u) \mathrel{\dot{\to}} [\![D\circ v]\!]\ X$$

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 in, this Cartesian morphism induces the ornamental algebra:

$$\begin{array}{l}
\mathsf{forgetAlg}\ (o\!:\!\mathsf{orn}\ D\ u\ u)\ (xs\!:\![\![o]\!]\ (\mu\, D\circ u)\ i)\ :\ (\mu\, D\circ u)\ i \\
\mathsf{forgetAlg}\ o\ xs\ \mapsto\ \mathsf{in}\ (\mathsf{forgetNT}\ o\ xs)
\end{array}$$

8.35 Definition (Forgetful map)**.** In turn, this algebra induces a forgetful map from the ornamented type to its original form:

$$\begin{array}{lcl}
\mathsf{forget}\ (o\!:\!\mathsf{orn}\ D\ u\ u) & : & \mu\ [\![o]\!]_{\mathsf{orn}} \mathrel{\dot{\to}} \mu\, D\circ u \\
\mathsf{forget}\qquad o & \mapsto & (\!|\mathsf{forgetAlg}\ o|\!)
\end{array}$$

8.36 Example (From lists back to natural numbers)**.** Applied to the ornament ListO, the Cartesian morphism removes the extra information added through insert, *i.e.* the inhabitant of A. The resulting algebra thus takes nil to 0, and cons a to 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 FinO, the Cartesian morphism removes the equations introduced by insert and forgets the indexing discipline enforced by the var code. The resulting forgetful map computes the cardinality – a natural number – of a finite set.

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 $D : \mathsf{func}\, K\, K$ and an algebra $\alpha : [\![D]\!]\ X \dot{\to} X$, we define an ornament D^{α} indexed by $(k\,{:}\,K) \times X\, k$ satisfying the following coherence property:

$$\mu\, D^{\alpha}\ (k, x) \cong (t\,{:}\,\mu\, D\, k) \times (\!|\alpha|\!)\ t = x$$

Seen as a set comprehension, this states that $\mu\, D^{\alpha}\ (k, x)$ is an inductive definition equivalent to the refinement type $\{t \in \mu\, D\, k \mid (\!|\alpha|\!)\ t = x\}$. 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 $m : \mathsf{Nat}$, the addition $m + - : \mathsf{Nat} \to \mathsf{Nat}$ is definable by folding the algebra:

$$\begin{array}{lcl}
\mathsf{add}_m\ (xn\,{:}\,[\![\mathsf{NatD}]\!]\ \mathsf{Nat}) & : & \mathsf{Nat} \\
\mathsf{add}_m \qquad [\text{'}0] & \mapsto & m \\
\mathsf{add}_m \quad\ [\text{'}\mathsf{suc}\, n] & \mapsto & \mathsf{suc}\, n
\end{array}$$

By algebraically ornamenting Nat by this algebra, we obtain the type of intervals $[m, -\,] : \mathsf{Nat} \to \mathrm{SET}$ that is characterised by the isomorphism:

$$[m,n] \cong (k\,{:}\,\mathsf{Nat}) \times (m + k = n)$$

Put explicitly, the datatype computed by the algebraic ornament corresponds to

$$\begin{array}{l}
\textbf{data}\ [[m\,{:}\,\mathsf{Nat}], -\,]\ (n\,{:}\,\mathsf{Nat}) : \mathrm{SET}\ \textbf{where} \\
\quad [m, -\,] \quad (n = m) \quad \ni 0 \\
\quad [m, -\,]\ (n = \mathsf{suc}\, n') \ \ni \mathsf{suc}\,(k\,{:}\,[m,n'])
\end{array}$$

(8.40) The type-theoretic construction of D^{α} 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 α 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:

$$\begin{array}{ll} (D : \mathsf{func}\, K\, K)^{(\alpha : [\![D]\!]\, X \dot{\to} X)} & : \mathsf{orn}\, D\, \pi_0\, \pi_0 \\ D^{\alpha} \mapsto \lambda(k, x).\ & \mathsf{insert}\, ([\![D]\!]\, X\, k)\, \lambda xs. \\ & \mathsf{insert}\, (\alpha\ xs = x)\, \lambda - . \\ & \Box_D\ (k, xs) \end{array}$$

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^{α} 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^{in}. Note that in is an isomorphism (by Lambek's lemma 4.35). The definition of D^{in} does indeed reduce to $\Box_D$: 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 Nat:

$$\begin{array}{lll} \textbf{data}\ \mathsf{Expr}\!:\!\textsc{Set}\ \textbf{where} & & \\ \quad \mathsf{Expr}\ \ni\ \mathsf{const}\,(n\!:\!\mathsf{Nat}) & & \\ \qquad\quad \mid\ \mathsf{add}\,(d\!:\!\mathsf{Expr})(e\!:\!\mathsf{Expr}) & & \end{array} \qquad \begin{array}{llcl} \mathsf{evalAlg} & (es\!:\![\![\mathsf{Expr\text{-}Desc}]\!]\,\mathsf{Nat}) & : & \mathsf{Nat} \\ \mathsf{evalAlg} & [\text{'}\mathsf{const}\,n] & \mapsto & n \\ \mathsf{evalAlg} & [\text{'}\mathsf{add}\,m\,n] & \mapsto & m+n \end{array}$$

Using the algebra evalAlg, we construct the algebraic ornament of Expr and obtain expressions indexed by their semantics:

$$\begin{array}{l} \textbf{data}\ \mathsf{Expr}^{\mathsf{evalAlg}}\,(k\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where} \\ \quad \mathsf{Expr}^{\mathsf{evalAlg}}\quad (k=n)\quad\ \ni\ \mathsf{const}\,(n\!:\!\mathsf{Nat}) \\ \quad \mathsf{Expr}^{\mathsf{evalAlg}}\,(k=m+n)\ \ni\ \mathsf{add}\,(m\,n\!:\!\mathsf{Nat})(d\!:\!\mathsf{Expr}^{\mathsf{evalAlg}}\,m)(e\!:\!\mathsf{Expr}^{\mathsf{evalAlg}}\,n) \end{array}$$

We can now enforce the preservation of semantics by typing. For example, let us optimise away all additions of the form "$0 + e$":

$$\begin{array}{lccl} \mathsf{optimise\text{-}0\!+} & (e\!:\!\mathsf{Expr}^{\mathsf{evalAlg}}\,n) & : & \mathsf{Expr}^{\mathsf{evalAlg}}\,n \\ \mathsf{optimise\text{-}0\!+} & (\mathsf{const}\,n) & \mapsto & \mathsf{const}\,n \\ \mathsf{optimise\text{-}0\!+} & (\mathsf{add}\,0\,n\,(\mathsf{const}\,0)\,e) & \mapsto & \mathsf{optimise\text{-}0\!+}\,e \\ \mathsf{optimise\text{-}0\!+} & (\mathsf{add}\,m\,n\,d\,e) & \mapsto & \mathsf{add}\,m\,n\,d\,e \end{array}$$

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 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 $\mu\, D^{\alpha}(i,x) \to (t : \mu\, D\, i) \times (\!|\alpha|\!)\ t = x$ relies on the generic forget D^{α} function to compute the first component of the pair, and gives us the following theorem:

$$\mathsf{coherentOrn}\!:\!\forall t^{\alpha}\!:\!\mu\, D^{\alpha}(i,x).\ (\!|\alpha|\!)\ (\mathsf{forget}\,D^{\alpha}\ t^{\alpha}) = x$$

This corresponds to the Recomputation theorem of McBride [2013, §8].

(8.49) **Computational interpretation.** From an inhabitant of D^{α}, we can extract its computational component with 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^{α} was enforcing.

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 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:

$$(t : \mu\, D\, i) \times (\!|\alpha|\!)\; t = x \to \mu\, D^{\alpha}(i, x)$$

Put in full and simplifying the equation, this corresponds to a function:

$$D^{\alpha}\text{-}\mathsf{make} : (t : \mu\, D\, i) \to \mu\, D^{\alpha}(i, (\!|\alpha|\!)\; t)$$

This corresponds to the remember function of McBride [2013, §7].

(8.52) **Computational interpretation.** This operation lets us lift an inhabitant of D to its algebraic refinement D^{α}. Computationally, this function is an identity function: its role is purely logical, unfolding the index-level computation $(\!|\alpha|\!)$ across the inhabitant of D.

8.1.4. Reornaments

MODEL: `Chapter8.Reornament`

(8.53) 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 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 $[\![o_D]\!]_{\mathsf{orn}}$ using the ornamental algebra 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 $\lceil o_D \rceil$. For brevity, we call it the *reornament* of o_D. As usual, we write $\lceil o_D \rceil$ to denote both the ornament and the resulting description.

8.55 Example (Reornament: vectors)**.** Following Example 8.44, a standard example of reornament is Vec: it is the reornament of ListO. Explicitly, a vector is the algebraic ornament of List by the algebra computing its length, *i.e.* the ornamental algebra from List to Nat. This corresponds to the definition of vectors by equality constraints:

$$\begin{array}{l} \textbf{data}\ \mathsf{Vec}\,[A\!:\!\textsc{Set}](n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where} \\ \quad \mathsf{Vec}\,A\quad (n = 0) \quad \ni \mathsf{nil} \\ \quad \mathsf{Vec}\,A\ (n = \mathsf{suc}\,n') \ni \mathsf{cons}\,(n'\!:\!\mathsf{Nat})(a\!:\!A)(vs\!:\!\mathsf{Vec}\,A\,n') \end{array}$$

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:

$$\begin{array}{l} \textbf{data}\ \mathsf{IMaybe}\,[A\!:\!\textsc{Set}](b\!:\!\mathsf{Bool})\!:\!\textsc{Set}\ \textbf{where} \\ \quad \mathsf{IMaybe}\,A\ (b = \mathsf{true}) \ni \mathsf{just}\,(a\!:\!A) \\ \quad \mathsf{IMaybe}\,A\ (b = \mathsf{false}) \ni \mathsf{nothing} \end{array}$$

This datatype comes – by folding its ornamental algebra – with the forgetful map:

$$\begin{array}{lll} \mathsf{forgetIMaybe}\,(mba\!:\!\mathsf{IMaybe}\,A\,b) & : & (ma\!:\!\mathsf{Maybe}\,A) \times \mathsf{isJust}\,ma = b \\ \mathsf{forgetIMaybe}\quad (\mathsf{just}\,a) & \mapsto & (\mathsf{just}\,a, \mathsf{refl}) \\ \mathsf{forgetIMaybe}\quad \mathsf{nothing} & \mapsto & (\mathsf{nothing}, \mathsf{refl}) \end{array}$$

(8.57) 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.

8.58 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 $n\!:\!\mathsf{Nat}$;
- 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 $n = \mathsf{suc}\,n'$, 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 $\mathsf{suc}\, n'$ is a vector of size n'. The recursive structure is denoted $\mathsf{Structure}$.

Let us generalise this construction to any ornament.

8.59 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

$$\{t^{+} : \mu\, [\![o_D]\!]_{\mathsf{orn}}\, i \mid \mathsf{forget}\, o_D\, t^{+} = t\}$$

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:

$$\begin{array}{lccl}
\multicolumn{4}{l}{\mathsf{Extension}\ (O : \mathsf{Orn}\, D\, u)\ (xs : [\![D]\!]\, \mu\, D) \ :\ \textsc{Set}} \\
\multicolumn{4}{l}{\quad \textit{-- Do not duplicate the original data, it is already in xs:}} \\
\mathsf{Extension} & (\mathsf{var}\, (\mathsf{inv}\, i)) & t & \mapsto \mathbb{1} \\
\mathsf{Extension} & 1 & * & \mapsto \mathbb{1} \\
\mathsf{Extension} & (\Pi\, T^{+}) & f & \mapsto (s : S) \to \mathsf{Extension}\, (T^{+}\, s)\, (f\, s) \\
\mathsf{Extension} & (A^{+} \times B^{+}) & (a, b) & \mapsto \mathsf{Extension}\, A^{+}\, a \times \mathsf{Extension}\, B^{+}\, b \\
\mathsf{Extension} & (\sigma\, T^{+}) & (e, xs) & \mapsto \mathsf{Extension}\, (\mathsf{switch}\, T^{+}\, e)\, xs \\
\mathsf{Extension} & (\Sigma\, T^{+}) & (s, xs) & \mapsto \mathsf{Extension}\, (T^{+}\, s)\, xs \\
\multicolumn{4}{l}{\quad \textit{-- Ask for freshly inserted data, it is the data missing from xs:}} \\
\mathsf{Extension} & (\mathsf{insert}\, S\, D^{+}) & xs & \mapsto (s : S) \times \mathsf{Extension}\, (D^{+}\, s)\, xs \\
\multicolumn{4}{l}{\quad \textit{-- Deleted data must be consistent with xs:}} \\
\mathsf{Extension} & (\mathsf{delete}\, s\, T^{+}) & (s', xs) & \mapsto (q : s = s') \times \mathsf{Extension}\, T^{+}\, xs
\end{array}$$

(8.60) 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.

8.61 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 the subnode of t we have reached. We can delete Σ and insert codes to avoid information duplication: the information is already provided by the index in the case of Σ, and by the extension in the case of insert. We proceed as follows:

$$\begin{array}{lllll}
\multicolumn{5}{l}{\mathsf{Structure}\ (O\!:\!\mathsf{Orn}\,D\,u)\ (xs\!:\![\![D]\!]\ \mu\,D)\ (e\!:\!\mathsf{Extension}\,O\,xs)\ :\ \mathsf{Orn}\ [\![O]\!]_{\mathsf{orn}}\ \pi_0} \\
\multicolumn{5}{l}{\textit{– Extract index from the ornament and the index:}} \\
\mathsf{Structure} & (\mathsf{var}\,(\mathsf{inv}\,i)) & t & * & \mapsto \mathsf{var}\,(\mathsf{inv}\,(i, t)) \\
\multicolumn{5}{l}{\textit{– Duplicate only the recursive structure:}} \\
\mathsf{Structure} & 1 & * & * & \mapsto 1 \\
\mathsf{Structure} & (\Pi\,T^{+}) & f & ext & \mapsto \Pi\,\lambda s.\,\mathsf{Structure}\,(T^{+}\,s)\,(f\,s)\,(ext\,s) \\
\mathsf{Structure} & (A^{+} \times B^{+}) & (a, b) & (ext_a, ext_b) & \mapsto \mathsf{Structure}\,A^{+}\,a\,ext_a \times \mathsf{Structure}\,B^{+}\,b\,ext_b \\
\mathsf{Structure} & (\sigma\,T^{+}) & (e, xs) & ext & \mapsto \mathsf{delete}\,e\,(\mathsf{Structure}\,(\mathsf{switch}\,T^{+}\,e)\,xs\,ext) \\
\mathsf{Structure} & (\Sigma\,T^{+}) & (s, xs) & ext & \mapsto \mathsf{delete}\,s\,(\mathsf{Structure}\,(T^{+}\,s)\,xs\,ext) \\
\mathsf{Structure} & (\mathsf{insert}\,S\,D^{+}) & xs & (s, ext) & \mapsto \mathsf{delete}\,s\,(\mathsf{Structure}\,(D^{+}\,s)\,xs\,ext) \\
\mathsf{Structure} & (\mathsf{delete}\,s\,T^{+}) & (s, xs) & (\mathsf{refl}, ext) & \mapsto \mathsf{Structure}\,T^{+}\,xs\,ext
\end{array}$$

8.62 Definition (Reornament)**.** A reornament is thus the $\mathsf{Extension}$ of its index followed by the arguments specified by its recursive $\mathsf{Structure}$. We define the reornament at index $t = \mathsf{in}\,xs\!:\!\mu\,D$ by inserting the valid extensions of t, followed by its recursive structure:

$$\begin{array}{lll}
\mathsf{reornament}\ (o\!:\!\mathsf{orn}\,D\,u\,u) & : & \mathsf{orn}\ [\![o]\!]_{\mathsf{orn}}\ \pi_0\ \pi_0 \\
\mathsf{reornament}\quad o & \mapsto & \lambda(i, \mathsf{in}\,xs).\,\mathsf{insert}\,(\mathsf{Extension}\,(o\,i)\,xs)\,\lambda e. \\
 & & \qquad\qquad \mathsf{Structure}\,(o\,i)\,xs\,e
\end{array}$$

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:

$$\begin{array}{l}
\textbf{data}\ \mathsf{Vec}\,[A\!:\!\mathrm{SET}](n\!:\!\mathsf{Nat})\!:\!\mathrm{SET}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathsf{Vec}\,A & 0 & \ni \mathsf{nil} \\
\mathsf{Vec}\,A & (\mathsf{suc}\,n) & \ni \mathsf{cons}\,(a\!:\!A)(vs\!:\!\mathsf{Vec}\,A\,n)
\end{array}
\end{array}$$

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

$$\begin{array}{l}
\textbf{data}\ \mathsf{IMaybe}\,[A\!:\!\textsc{Set}](b\!:\!\mathsf{Bool})\!:\!\textsc{Set}\ \textbf{where}\\
\quad \mathsf{IMaybe}\,A\ \mathsf{true}\ \ni\ \mathsf{just}\,(a\!:\!A)\\
\quad \mathsf{IMaybe}\,A\ \mathsf{false}\ \ni\ \mathsf{nothing}
\end{array}$$

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. 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 coherentOrn theorem in one direction and the $\lceil o_D \rceil$-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 Length : Nat → List A → SET. Reornamenting 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 Extension and Structure. We proceed by case analysis on the ornament. On a 1, $\times$, Π, and var code, the reornamentation proceeds purely structurally, hence introducing no information. The reornamentation *deletes* Σ 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 Σ code.

In the next iteration of the reornament, the insert codes of the ornament (*e.g.* list from nat) turn into Σ codes in its reornament (*e.g.* vector). In the subsequent iteration, these Σ codes in the reornament are in turn deleted by the re-reornament (*e.g.* the 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.

(8.67) 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

(8.68) 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: *ornaments are Cartesian morphisms of containers*. In this section, we substantiate this claim.

8.2.1. Cartesian morphisms of containers

MODEL: `Chapter8.Container.Morphism`
`Chapter8.Container.BaseChange`
`Chapter8.Container.CobaseChange`

(8.69) 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 $\mathsf{Op}' \triangleleft^{\mathsf{Sort}'} \mathsf{Ar}' : \mathsf{ICont}\, I\, J$ and $\mathsf{Op} \triangleleft^{\mathsf{Sort}} \mathsf{Ar} : \mathsf{ICont}\, K\, L$ be two containers, indexed respectively by (I, J) and (K, L). Let $u : I \rightarrow K$ and $v : J \rightarrow L$ be two functions mapping the indices of the former to the latter.

A Cartesian morphism from $\mathsf{Op}' \triangleleft^{\mathsf{Sort}'} \mathsf{Ar}'$ to $\mathsf{Op} \triangleleft^{\mathsf{Sort}} \mathsf{Ar}$ framed by u and v corresponds to a triple

$$\begin{cases} \omega : \mathsf{Op}'\, j \rightarrow \mathsf{Op}\ (v\, j) \\ \rho : \forall\, op' : \mathsf{Op}'\, j.\ \mathsf{Ar}\ (\omega\, op') = \mathsf{Ar}'\ op' \\ q : \forall\, op' : \mathsf{Op}'\, j.\ \forall\, ar : \mathsf{Ar}\ (\omega\, op').\ u\ (\mathsf{Sort}'\ ar) = \mathsf{Sort}\, ar \end{cases}$$

where ω maps the operations of $\mathsf{Op}' \triangleleft^{\mathsf{Sort}'} \mathsf{Ar}'$ to operations of $\mathsf{Op} \triangleleft^{\mathsf{Sort}} \mathsf{Ar}$, while ρ enforces that the operations mapped through ω 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 ω, leaving it to the reader to verify that the side-conditions ρ and q are satisfied.

Following 2-categorical conventions [Shulman, 2008], the hom-set of Cartesian morphisms (*i.e.* 2-cell) from $\mathsf{Op}'{\lhd}^{\mathsf{Sort}'}\mathsf{Ar}'$ to $\mathsf{Op}{\lhd}^{\mathsf{Sort}}\mathsf{Ar}$ framed by u and v is denoted

$$\mathsf{Op}'{\lhd}^{\mathsf{Sort}'}\mathsf{Ar}' \underset{v}{\overset{u}{\Longrightarrow}}{}^{c} \mathsf{Op}{\lhd}^{\mathsf{Sort}}\mathsf{Ar}$$

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 $[\textsc{Set}^I, \textsc{Set}^J]$ [Gambino and Kock, 2013, Proposition 2.4]. The category $\textsc{Set}^J$ being pullback-complete, and $\textsc{Set}^I$ having a terminal object $\mathbb{1}$, we may consider the fibration

$$\begin{tikzcd}
{[\textsc{Set}^I, \textsc{Set}^J]} \arrow[d, "-\mathbb{1}"] \\
\textsc{Set}^J
\end{tikzcd}$$

The Cartesian morphisms we have just defined correspond *exactly* to the Cartesian natural transformations – natural transformations whose naturality square forms a 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 $\mathsf{ListCont}_A$ (Example 5.56) to $\mathsf{NatCont}$ (Example 5.55) by mapping operations of $\mathsf{ListCont}_A$ to operations of $\mathsf{NatCont}$:

$$\begin{array}{llll}
\multicolumn{4}{l}{\omega : \mathsf{ListCont}_A \underset{\mathsf{id}}{\overset{\mathsf{id}}{\Longrightarrow}}{}^{c} \mathsf{NatCont} \quad \textbf{where}} \\
\quad \omega\ (op_l : \mathsf{Op}_{\mathsf{List}}\ *) & : & \mathsf{Op}_{\mathsf{Nat}}\ * & \\
\quad \omega\ \quad (\mathsf{inj}_l\ *) & \mapsto & \mathsf{inj}_l\ * & -\ \mathsf{nil}\ \textit{to}\ 0 \\
\quad \omega\ \quad (\mathsf{inj}_r\ a) & \mapsto & \mathsf{inj}_r\ * & -\ \mathsf{cons}\ a\ \textit{to}\ \mathsf{suc}
\end{array}$$

We are then left to check that arities are isomorphic: this is indeed true, since, in the $\mathsf{nil}/0$ case, the arity is nil while, in the $\mathsf{cons}/\mathsf{suc}$ case, the arity is one. The coherence condition is trivially satisfied, since both containers are indexed by $\mathbb{1}$. 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 Op' to operations Op covariantly using ω:

$$\begin{array}{l}
[\![(m : \mathsf{Op}' \lhd^{\mathsf{Sort}'} \mathsf{Ar}' \overset{u}{\underset{v}{\Longrightarrow}}{}^{c} \mathsf{Op} \lhd^{\mathsf{Sort}} \mathsf{Ar})]\!]_{\mathsf{Cont}}\ (xs : [\![\mathsf{Op}' \lhd^{\mathsf{Sort}'} \mathsf{Ar}']\!]_{\mathsf{Cont}}\ (X \circ u)\ j)\ :\ [\![\mathsf{Op} \lhd^{\mathsf{Sort}} \mathsf{Ar}]\!]_{\mathsf{Cont}}\ X\ (v\ j) \\
[\![\omega]\!]_{\mathsf{Cont}}\ (op', xs) \mapsto (\omega\ op', xs)
\end{array}$$

(8.74) For a pair of indices $I, J : \textsc{Set}$, we can – quite unsurprisingly – organise $\mathsf{ICont}\ I\ J$ into a category [Abbott, 2003], with the morphisms defined to be the Cartesian morphisms framed by the identity ($u \triangleq \mathrm{id}_I$ and $v \triangleq \mathrm{id}_J$).

More interestingly, we can organise ICont 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 $\mathsf{ICont}\ I\ J$. 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 $\mathsf{Op} \lhd^{\mathsf{Sort}} \mathsf{Ar} : \mathsf{ICont}\ K\ L$ 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 $\mathsf{Op} \lhd^{\mathsf{Sort}} \mathsf{Ar}$ by u and v, written $(u, v)^{*} \mathsf{Op} \lhd^{\mathsf{Sort}} \mathsf{Ar}$, is a container indexed by I and J, defined as follows:

$$(u,v)^{*}\mathsf{Op}\lhd^{\mathsf{Sort}}\mathsf{Ar} \triangleq \begin{cases} \mathsf{Op}^{*}\ (j\,{:}\,J) & : \ \textsc{Set} \\ \mathsf{Op}^{*}\quad j & \mapsto \ (l\,{:}\,v^{-1}\,j) \times \mathsf{Op}\,l \\ \mathsf{Ar}^{*}\ (op\,{:}\,\mathsf{Op}^{*}\,j) & : \ \textsc{Set} \\ \mathsf{Ar}^{*}\quad op & \mapsto \ \mathsf{Ar}\,(\pi_1\,op) \\ \mathsf{Sort}^{*}\ (ar\,{:}\,\mathsf{Ar}^{*}\,op) & : \ I \\ \mathsf{Sort}^{*}\quad ar & \mapsto \ v\,(\mathsf{Sort}\,ar) \end{cases}$$

(8.77) **Intuition.** In a sense, the base-change container $(u,v)^{*}\mathsf{Op}\lhd^{\mathsf{Sort}}\mathsf{Ar}$ is the "closest" container to $\mathsf{Op}\lhd^{\mathsf{Sort}}\mathsf{Ar}$ 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 $\mathsf{Op}\lhd^{\mathsf{Sort}}\mathsf{Ar} : \mathsf{ICont}\,I\,J$ 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 $\mathsf{Op}\lhd^{\mathsf{Sort}}\mathsf{Ar}$ by u and v, written $(u,v)_{!}\mathsf{Op}\lhd^{\mathsf{Sort}}\mathsf{Ar}$, is a container indexed by K and L, defined as follows:

$$(u,v)_{!}\mathsf{Op}\lhd^{\mathsf{Sort}}\mathsf{Ar} \triangleq \begin{cases} \mathsf{Op}_{!}\ (l\,{:}\,L) & : \ \textsc{Set} \\ \mathsf{Op}_{!}\quad l & \mapsto \ (op\,{:}\,\mathsf{Op}\,(v\,l)) \times ((ar\,{:}\,\mathsf{Ar}\,op) \to (k\,{:}\,K) \times u\,k = \mathsf{Sort}\,ar) \\ \mathsf{Ar}_{!}\ (op\,{:}\,\mathsf{Op}_{!}\,l) & : \ \textsc{Set} \\ \mathsf{Ar}_{!}\quad op & \mapsto \ \mathsf{Ar}\,(\pi_0\,op) \\ \mathsf{Sort}_{!}\ (ar\,{:}\,\mathsf{Ar}_{!}\,op) & : \ K \\ \mathsf{Sort}_{!}\quad ar & \mapsto \ \pi_0\,((\pi_1\,op)\,ar) \end{cases}$$

(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 $(u,v)_{!}\mathsf{Op}\lhd^{\mathsf{Sort}}\mathsf{Ar}$ is thus the more precisely-indexed container derived from $\mathsf{Op}\lhd^{\mathsf{Sort}}\mathsf{Ar}$. 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.

8.2.2. 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 $\mathsf{Op} \lhd^{\mathsf{Sort}} \mathsf{Ar}$, an ornament can be modelled as an extension *ext*, and a refined indexing Sort^{+} subject to a coherence condition q with respect to the original indexing:

$$\begin{cases} ext : \mathsf{Op}\ (v\ l) \to \mathrm{SET} \\ \mathsf{Sort}^{+} : ext\ op \to \mathsf{Ar}\ op \to K \\ q : \forall\ ar : \mathsf{Ar}\ op.\ u\ (\mathsf{Sort}^{+}\ e\ ar) = \mathsf{Sort}\ ar \end{cases}$$

(8.83) Equivalently, the family of set *ext* can be understood as the inverse image of a function $\omega : \mathsf{Op}^{+}\ l \to \mathsf{Op}\ (v\ l)$. The function Sort^{+} is then the arguments' sort of a container with operations Op^{+} and arities $\mathsf{Ar} \circ \omega$. Put otherwise, the morphism on operations ω together with the coherence condition q form a Cartesian morphism from $\mathsf{Op}^{+} \lhd^{\mathsf{Sort}^{+}} \mathsf{Ar} \circ \omega$ to $\mathsf{Op} \lhd^{\mathsf{Sort}} \mathsf{Ar}$!

8.84 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

$$\mathsf{orn}\, D\; u\; v \cong \mathsf{ICont}(-, \langle D \rangle)_{u,v}$$

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 $[\![o]\!]_{\mathsf{orn}}$ to the container described by D. By definition of Cartesian morphisms, we simply have to give a map from the operations of $[\![o]\!]_{\mathsf{orn}}$ 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 Π 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.

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:

$$\begin{array}{lll}
\psi\, (m : F \overset{u}{\underset{v}{\Longrightarrow}}{}^{c} G) & : & \mathsf{orn}\, \langle G \rangle^{-1}\, u\; v \\
\psi \quad (\mathit{forget}) & \mapsto & \lambda j.\, \Sigma\, \lambda sh.\, \mathsf{insert}\, (\mathit{forget}^{-1}\, sh)\, \lambda ext.\, \Pi\, \lambda ps.\, \mathsf{var}\, (\mathsf{inv}\, (n_F\; ps))
\end{array}$$

Indeed, the description of G is a Σ of its operations, followed by a Π of its arities, terminated by a var at the desired sort. To ornament G into F, we simply have to insert the inverse image of *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 ϕ and ψ 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.

□

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 ϕ, followed by the interpretation of the resulting Cartesian morphism. The Cartesian morphism $\mathsf{forgetNT}$ is indeed natural and Cartesian. This natural transformation was already present in the original presenta-

$$
\begin{array}{llll}
\phi\ (o : \mathsf{orn}\ D\ u\ v) & : & \langle [\![o]\!]_{\mathsf{orn}} \rangle \overset{u}{\underset{v}{\Longrightarrow}}{}^{c} \langle D \rangle & \\
\phi\ \quad o & \mapsto & (\lambda i.\, \mathsf{forget}\ (o\ i)\ (u\ i)) & \textbf{where}
\end{array}
$$

$$
\begin{array}{llcll}
\mathsf{forget} & (O : \mathsf{Orn}\ D\ u) & (sh : \mathsf{Op}_{\mathsf{func}}\ [\![O]\!]_{\mathsf{orn}}) & : & \mathsf{Op}_{\mathsf{func}}\ D \\
\mathsf{forget} & (\mathsf{insert}\ a\ D^{+}) & (a, sh) & \mapsto & \mathsf{forget}\ (D^{+}\ a)\ sh \\
\mathsf{forget} & (\mathsf{var}\ (\mathsf{inv}\ j)) & * & \mapsto & * \\
\mathsf{forget} & 1 & * & \mapsto & * \\
\mathsf{forget} & (A^{+} \times B^{+}) & (a, b) & \mapsto & (\mathsf{forget}\ A^{+}\ a, \mathsf{forget}\ B^{+}\ b) \\
\mathsf{forget} & (\sigma\ T^{+}) & (e, sh) & \mapsto & (e, \mathsf{forget}\ (\mathsf{switch}\ T^{+}\ e)\ sh) \\
\mathsf{forget} & (\Pi\ T^{+}) & f & \mapsto & \lambda a.\, \mathsf{forget}\ (T^{+}\ a)\ (f\ a) \\
\mathsf{forget} & (\Sigma\ T^{+}) & (a, sh) & \mapsto & (a, \mathsf{forget}\ (T^{+}\ a)\ sh) \\
\mathsf{forget} & (\mathsf{delete}\ s\ O) & sh & \mapsto & (s, \mathsf{forget}\ O\ sh)
\end{array}
$$

Figure 8.2.: From ornament to Cartesian morphism

tion [McBride, 2013, §4], in the form of the helper function 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*.

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

MODEL: `Chapter8.Container.Morphism.Cartesian,`
`Chapter8.Container.Morphism.Contornament`

(8.88) 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

(8.89) 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 $o : F \overset{u}{\underset{v}{\Longrightarrow}}{}^{c} G$, we obtain the *ornamental algebra* $\mathsf{forgetAlg}\, o : F\ (\mu G \circ v) \to \mu G \circ u$.

Proof. We apply the natural transformation o at μG and post-compose by *in*:

$$\mathsf{forgetAlg}\, o : F\ (\mu G \circ v) \xrightarrow{o_{\mu G}} (G\ \mu G) \circ u \xrightarrow{in} \mu G \circ u$$

□

(8.91) 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.

(8.92) **Computational interpretation.** Folding the ornamental algebra, we obtain a map from the ornamented type μF to its unornamented version μG. 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 mapped to suc. Post-composing by *in*, we obtain a natural number. This is the algebra computing the length of a list.

8.3.2. Algebraic ornaments

(8.94) 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.

8.95 Definition (Refinement functor [Atkey et al., 2012, §4.3])**.** Let F be an endofunctor on $\textsc{Set}^I$. Let $(X : \textsc{Set}^I, \alpha : F\ X \to X)$ be an algebra over F.

The *refinement functor* is defined by

$$F^{\alpha} \triangleq \Sigma_{\alpha} \circ \hat{F} : Set^{(i:I) \times X\, i} \rightarrow Set^{(i:I) \times X\, i}$$

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

$$\begin{array}{l} \hat{F}\ (P : (i : I) \times X\, i \rightarrow \textsc{Set})\ (ixs : (i : I) \times F\, X\, i)\ :\ \textsc{Set} \\ \hat{F}\ P\ ixs \ \mapsto\ (ps : F\ (\lambda - .\ (ix : (i : I) \times X\, i) \times P\, ix)) \times F\ \pi_0\ ps = ixs \end{array}$$

(8.96) The idea, drawn from refinement types [Freeman and Pfenning, 1991], is that a function $(\!|\alpha|\!) : \mu F \rightarrow X$ can be thought of as a predicate over μF. By *integrating* the algebra α *into* the signature F, we obtain a signature F^{α} indexed by X that describes the F-objects satisfying, by construction, the predicate $(\!|\alpha|\!)$. Categorically, this translates to:

8.97 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 $\mu {P_F}^{\alpha} \cong \Sigma_{(\!|\alpha|\!)} \mathbf{1}\, \mu F$ where $\mathbf{1} : \textsc{Set}^{I} \rightarrow [\textsc{Set}^{I}, \textsc{Set}^{I}]$, the terminal object functor, maps objects X to id_X (Remark 4.61).

(8.98) Informally, using a set-theoretic notation, this isomorphisms reads as

$$\begin{aligned} \mu F^{\alpha}\ i\ x &\cong \Sigma_{(\!|\alpha|\!)} \mathbf{1}\, \mu F \\ &\cong \{t : \mu F\ i \mid (\!|\alpha|\!)\ t = x\} \end{aligned}$$

That is, the algebraic ornament μF^{α} 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 $(\!|\alpha|\!)\ t = x$.

(8.99) 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 α, we can *compute* an ornament code that describes the functor D^{α}. 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.

8.100 Proposition. Let $\sigma \triangleq \mathsf{Op} \lhd^{\mathsf{Sort}} \mathsf{Ar}$ be a container indexed by I. Let (X, α) be an algebra over F, *i.e.* $\alpha : [\![\sigma]\!]_{\mathsf{Cont}}\ X \dot{\rightarrow} X$.

The refinement functor σ^{α} is polynomial and ornaments σ.

Proof. We first show that $\hat{\sigma}$ ornaments σ, and then that Σ_{α} ornaments the identity container. By horizontal composition, we thus get that σ^{α} ornaments σ.

To show that $\hat{\sigma}$ ornaments σ, we remark that the canonical lifting of a container itself a container. Indeed, we define

$$\begin{array}{lcl} \mathsf{C}\Box\,(\sigma : \mathsf{ICont}\,I\,J) & : & \mathsf{ICont}\,((i : I) \times X\,i)\,((j : J) \times [\![\sigma]\!]_{\mathsf{Cont}}\,X\,j) \\ \mathsf{C}\Box\quad \mathsf{Op} \lhd^{\mathsf{Sort}} \mathsf{Ar} & \mapsto & \begin{cases} \mathsf{Op}^{l}\;(jxs : (j : J) \times [\![\sigma]\!]_{\mathsf{Cont}}\,X\,j) \;:\; \textsc{Set} \\ \mathsf{Op}^{l}\;jxs \mapsto \mathbb{1} \\ \mathsf{Ar}^{l}\;(op : \mathsf{Op}^{l}\,jxs) \;:\; \textsc{Set} \\ \mathsf{Ar}^{l}\quad * \quad \mapsto \mathsf{Ar}\,(\pi_0\,(\pi_1\,jxs)) \\ \mathsf{Sort}^{l}\;(ar : \mathsf{Ar}^{l}\,op) \;:\; (i : I) \times X\,i \\ \mathsf{Sort}^{l}\quad ar \quad \mapsto (\mathsf{Sort}\,ar, (\pi_1\,(\pi_1\,jxs))\,ar) \end{cases} \end{array}$$

We then verify that $[\![\mathsf{C}\Box\,\sigma]\!]_{\mathsf{Cont}} \cong \widehat{[\![\sigma]\!]_{\mathsf{Cont}}}$ and that there exists a container morphism from $\mathsf{C}\Box\,\sigma$ to σ, which is suggested by the fact that both containers have the same arities.

Similarly, we can construct a container representing Σ_α (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 Σ_α to the identity container indexed by I.

By horizontal composition of these two Cartesian natural transformations, we obtain a Cartesian natural transformation from $\sigma^\alpha \triangleq \Sigma_\alpha \circ \hat{\sigma}$ to $\mathrm{id} \circ \sigma \cong \sigma$.

□

(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^α as the composition of the canonical lifting, followed by the op-reindexing by α.

First, we explicitly appeal to the canonical lifting. Second, the op-reindexing is coded by inserting an $xs : [\![D]\!]\,X\,k$ satisfying the constraint $\alpha\,xs = x$. In the internal language, we have:

$$\begin{aligned} &\Sigma_\alpha : \textsc{Set} / [\![D]\!]\,X \to \textsc{Set} / X \\ &\Sigma_\alpha\;\{P_{xs} \mid xs \in [\![D]\!]\,X\} \triangleq \left\{ \sum_{xs \in \alpha^{-1} x} P_{xs} \mid x \in X \right\} \\ &\qquad\qquad\qquad\qquad\quad \triangleq \{(xs : [\![D]\!]\,X) \times \alpha\,xs = x \times P_{xs} \mid x \in X\} \end{aligned}$$

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 description as input and returning the identity ornament:

$$
\begin{array}{lcll}
\mathsf{idO}\ (D\!:\!\mathsf{IDesc}\ I) & & : & \mathsf{Orn}\ D\ \mathsf{id} \\
\mathsf{idO} & (\mathsf{var}\ i) & \mapsto & \mathsf{var}\ (\mathsf{inv}\ i) \\
\mathsf{idO} & 1 & \mapsto & 1 \\
\mathsf{idO} & A \times B & \mapsto & \mathsf{idO}\ A \times \mathsf{idO}\ B \\
\mathsf{idO} & (\sigma\ E\ T) & \mapsto & \sigma\ \lambda e.\ \mathsf{idO}\ (\mathsf{switch}\ T\ e) \\
\mathsf{idO} & (\Pi\ S\ T) & \mapsto & \Pi\ \lambda s.\ \mathsf{idO}\ T\ s \\
\mathsf{idO} & (\Sigma\ S\ T) & \mapsto & \Sigma\ \lambda s.\ \mathsf{idO}\ T\ s
\end{array}
$$

This definition then lifts pointwise to ornaments:

$$\mathsf{idO} : (D\!:\!\mathsf{func}\ I\ J) \to \mathsf{orn}\ D\ \mathsf{id}\ \mathsf{id}$$

(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:

$$\begin{tikzcd}
\textsc{Set}/I \arrow[rr, bend left=60, "F"] \arrow[rr, "G" description] \arrow[rr, bend right=60, "H"'] \arrow[rr, phantom, shift left=4, "\Downarrow o_1"] \arrow[rr, phantom, shift right=4, "\Downarrow o_2"] & & \textsc{Set}/J & = & \textsc{Set}/I \arrow[rr, bend left=60, "F"] \arrow[rr, bend right=60, "H"'] \arrow[rr, phantom, "{\substack{o_2 \bullet o_1 \\ \Downarrow}}"] & & \textsc{Set}/J
\end{tikzcd}$$

8.105 Example (Vertical composition of ornaments)**.** We have seen that List ornaments Nat. We also know that Vec ornaments List. By vertical composition, we thus obtain that Vec ornaments Nat.

(8.106) **Horizontal composition.** Turning to horizontal composition, we have the following identity:

$$\begin{tikzcd} \textsc{Set}/I \arrow[r, bend left, "F_1"] \arrow[r, bend right, "G_1"'] \arrow[r, phantom, "\Downarrow o_1"] & \textsc{Set}/J \arrow[r, bend left, "F_2"] \arrow[r, bend right, "G_2"'] \arrow[r, phantom, "\Downarrow o_2"] & \textsc{Set}/K \end{tikzcd}$$

$$= \begin{tikzcd}[column sep=huge] \textsc{Set}/I \arrow[r, bend left, "F_2 \circ F_1"] \arrow[r, bend right, "G_2 \circ G_1"'] \arrow[r, phantom, "\substack{o_2 \circ o_1 \\ \Downarrow}"] & \textsc{Set}/K \end{tikzcd}$$

8.107 Example (Horizontal composition of ornaments)**.** Let us consider the following containers:

$$\begin{aligned} &\mathsf{Square}\; X \mapsto X \times X : \textsc{Set}/\mathbb{1} \to \textsc{Set}/\mathbb{1} \\ &\mathsf{Height}\; \{X_n \mid n \in \mathsf{Nat}\} \mapsto \{X_n \times X_{n+1} \mid n \in \mathsf{Nat}\} \\ &\qquad\qquad + \{X_n \times X_n \mid n \in \mathsf{Nat}\} : \textsc{Set}/\mathsf{Nat} \to \textsc{Set}/\mathsf{Nat} \end{aligned}$$

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 ∘ Height – describing a balanced binary tree – is an ornament of ListCont ∘ 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):

$$\begin{array}{l} \textbf{data}\ \mathsf{Expr}^{\mathsf{evalAlg}}\ (k : \mathsf{Nat}) : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Expr}^{\mathsf{evalAlg}}\ \ (k = n) \quad\ \ \ni \mathsf{const}\ (n : \mathsf{Nat}) \\ \quad \mathsf{Expr}^{\mathsf{evalAlg}}\ (k = m + n) \ni \mathsf{add}\ (m\ n : \mathsf{Nat})(d : \mathsf{Expr}^{\mathsf{evalAlg}}\ m)(e : \mathsf{Expr}^{\mathsf{evalAlg}}\ n) \end{array}$$

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

$$\begin{array}{l} \textbf{data}\ \mathsf{Nat}^* : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Nat}^* \ni \mathsf{1} \\ \qquad\quad \mid \mathsf{suc}\,(n : \mathsf{Nat}^*) \end{array}$$

and trivially embeds into natural numbers:

$$\begin{array}{lcl} \mathsf{toNat}\,(n : \mathsf{Nat}^*) & : & \mathsf{Nat} \\ \mathsf{toNat}\quad \mathsf{1} & \mapsto & \mathsf{suc}\,\mathsf{0} \\ \mathsf{toNat}\ (\mathsf{suc}\,n) & \mapsto & \mathsf{suc}\,(\mathsf{toNat}\,n) \end{array}$$

The container describing $\mathsf{Expr}^{\mathsf{evalAlg}}$ is indexed by Nat: we can therefore reindex it by toNat. We obtain a container that describes the expressions denoting strictly-positive natural numbers. Put explicitly, the object thus computed is the following datatype:

$$\begin{array}{l} \textbf{data}\ \mathsf{Expr}^{\mathsf{evalAlg}+}\,(k : \mathsf{Nat}^*) : \textsc{Set}\ \textbf{where} \\ \quad \mathsf{Expr}^{\mathsf{evalAlg}+}\,k \ni \mathsf{const}\,(n : \mathsf{Nat})(q : \mathsf{toNat}\,k = n) \\ \qquad\qquad\qquad \mid \mathsf{add}\ (m\,n : \mathsf{Nat}^*)(d : \mathsf{Expr}^{\mathsf{evalAlg}+}\,m)(e : \mathsf{Expr}^{\mathsf{evalAlg}+}\,n) \\ \qquad\qquad\qquad\qquad (q : \mathsf{toNat}\,k = \mathsf{toNat}\,m + \mathsf{toNat}\,n) \end{array}$$

For both constructors, the strict-positivity of inhabitants of $\mathsf{Expr}^{\mathsf{evalAlg}+}$ is enforced by the side-conditions q: the (strictly-positive) index k must be equal to the denotation of the expression.

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

(8.111) 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 $n : \mathsf{Nat}$. Put explicitly, the object thus computed is the following datatype:

$$\begin{array}{l} \textbf{data}\ \mathsf{BoundedList}\,[A\!:\!\textsc{Set}](n\!:\!\mathsf{Nat})\!:\!\textsc{Set}\ \textbf{where} \\ \quad \mathsf{BoundedList}_A\,(n = \mathsf{suc}\,n') \ni \mathsf{nil}\,(n'\!:\!\mathsf{Nat}) \\ \qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{cons}\,(n'\!:\!\mathsf{Nat})(a\!:\!A)(as\!:\!\mathsf{BoundedList}_A\,n') \end{array}$$

(8.113) 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

(8.116) 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 $I\!:\!\textsc{Set}$.

F is differentiable in $i : I$ if and only if, for any container G, we have the following bijection of morphisms

$$\frac{\mathsf{ICont}(G \times \pi_i, F)}{\mathsf{ICont}(G, \partial_i F)}$$

We denote $\mathsf{ICont}^{\partial_i}$ 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.

(8.120) 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 $\mathsf{ICont}^{\partial_i}$.
If F ornaments G, then $\partial_i F$ ornaments $\partial_i G$.

Proof. The proof simply follows from the functoriality of ∂_i over $\mathsf{ICont}^{\partial_i}$ [Abbott, 2003, Section 6.4]. Intuitively, we have the following Cartesian morphism

$$\partial_i F \times \pi_i \to^c F \to^c G$$

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:

$$\partial_i F \to^c \partial_i G$$

□

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.

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.

(8.128) 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 Nat comes with operations such as addition and subtraction. When defining List A as an ornament of Nat, it seems natural to transport the structure-preserving functions of Nat to List A, such as moving from addition of natural numbers to concatenation of lists:

$$
\begin{array}{lll}
(m : \mathsf{Nat}) + (n : \mathsf{Nat}) & : & \mathsf{Nat} \\
\mathsf{0} \qquad + n & \mapsto & n \\
(\mathsf{suc}\, m) + n & \mapsto & \mathsf{suc}\,(m + n)
\end{array}
\quad \Rightarrow \quad
\begin{array}{lll}
(xs : \mathsf{List}\, A) \mathbin{+\!\!+} (ys : \mathsf{List}\, A) & : & \mathsf{List}\, A \\
\mathsf{nil} \qquad \mathbin{+\!\!+} ys & \mapsto & ys \\
(\mathsf{cons}\, a\, xs) \mathbin{+\!\!+} ys & \mapsto & \mathsf{cons}\, a\, (xs \mathbin{+\!\!+} ys)
\end{array}
$$

Or, from subtraction of natural numbers to dropping a prefix:

$$
\begin{array}{lllll}
(m : \mathsf{Nat}) & - & (n : \mathsf{Nat}) & : & \mathsf{Nat} \\
\mathsf{0} & - & n & \mapsto & \mathsf{0} \\
m & - & \mathsf{0} & \mapsto & m \\
(\mathsf{suc}\, m) & - & (\mathsf{suc}\, n) & \mapsto & m - n
\end{array}
\quad \Rightarrow \quad
\begin{array}{lllll}
\mathsf{drop} & (xs : \mathsf{List}\, A) & (n : \mathsf{Nat}) & : & \mathsf{List}\, A \\
\mathsf{drop} & \mathsf{nil} & n & \mapsto & \mathsf{nil} \\
\mathsf{drop} & xs & \mathsf{0} & \mapsto & xs \\
\mathsf{drop} & (\mathsf{cons}\, a\, xs) & (\mathsf{suc}\, n) & \mapsto & \mathsf{drop}\, xs\, n
\end{array}
$$

(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.

$$\begin{array}{llllll} (m\!:\!\mathsf{Nat}) & < & (n\!:\!\mathsf{Nat}) & : & \mathsf{Bool} \\ m & < & 0 & \mapsto & \mathsf{false} \\ 0 & < & \mathsf{suc}\, n & \mapsto & \mathsf{true} \\ \mathsf{suc}\, m & < & \mathsf{suc}\, n & \mapsto & m < n \end{array} \Longrightarrow \begin{array}{llllll} \mathsf{lookup} & (m\!:\!\mathsf{Nat}) & (xs\!:\!\mathsf{List}\, A) & : & \mathsf{Maybe}\, A \\ \mathsf{lookup} & m & \mathsf{nil} & \mapsto & \mathsf{nothing} \\ \mathsf{lookup} & 0 & (\mathsf{cons}\, a\, xs) & \mapsto & \mathsf{just}\, a \\ \mathsf{lookup} & (\mathsf{suc}\, n) & (\mathsf{cons}\, a\, xs) & \mapsto & \mathsf{lookup}\, n\, xs \end{array}$$

Figure 9.1.: Implementation of $- < -$ and lookup

(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 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, false and nothing) in the first case analysis and a success value (respectively, true and 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:

$$\begin{tikzcd}
- < - & : & \mathsf{Nat} \arrow[d, "\mathsf{idO}_{\mathsf{Nat}}"'] & \to & \mathsf{Nat} \arrow[d, "\mathsf{ListO}\ A"'] & \to & \mathsf{Bool} \arrow[d, "\mathsf{MaybeO}\ A"'] \\
\mathsf{lookup} & : & \mathsf{Nat} & \to & \mathsf{List}\ A & \to & \mathsf{Maybe}\ A
\end{tikzcd}$$

Note that the first argument is ornamented to itself, or put differently, it has been ornamented by the identity ornament.

(9.10) **Ornamental algebras.** Each of these ornaments come with a forgetful map, computed from the ornamental algebra. These forgetful maps correspond to the following functions:

$$\begin{array}{lll}
\mathsf{length}\ (as : \mathsf{List}\ A) & : & \mathsf{Nat} \\
\mathsf{length}\quad \mathsf{nil} & \mapsto & 0 \\
\mathsf{length}\ (\mathsf{cons}\ a\ as) & \mapsto & \mathsf{suc}\ (\mathsf{length}\ as)
\end{array}
\qquad
\begin{array}{lll}
\mathsf{isJust}\ (m : \mathsf{Maybe}\ A) & : & \mathsf{Bool} \\
\mathsf{isJust}\quad \mathsf{nothing} & \mapsto & \mathsf{false} \\
\mathsf{isJust}\quad (\mathsf{just}\ a) & \mapsto & \mathsf{true}
\end{array}$$

(9.11) Using these forgetful map we deduce a relation, at the operational level, between $- < -$ and lookup. This relation is uniquely determined by the ornamentation of the individual datatypes. This *coherence property* is expressed as follows:

$$\forall n : \mathsf{Nat}.\ \forall xs : \mathsf{List}\ A.\ \ \mathsf{isJust}\ (\mathsf{lookup}\ n\ xs) = n < \mathsf{length}\ xs$$

Or, equivalently, using a commuting diagram:

$$\begin{tikzcd}
\mathsf{Nat} \times \mathsf{List}\ A \arrow[r, "\mathsf{lookup}"] \arrow[d, "\mathsf{id} \times \mathsf{length}"'] & \mathsf{Maybe}\ A \arrow[d, "\mathsf{isJust}"] \\
\mathsf{Nat} \times \mathsf{Nat} \arrow[r, "- < -"'] & \mathsf{Bool}
\end{tikzcd}$$

9.12 Remark (Vocabulary)**.** We call the function we start with the *base function* (here, $- < -$), its type being the *base type* (here, $\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool}$). The richer function type built by ornamenting the individual pieces is called the *functional ornament* (here, $\mathsf{Nat} \to$

$\mathsf{List}\ A \to \mathsf{Maybe}\ A$). A function inhabiting this type is called a *lifting* (here, 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, lookup remains to be implemented while making sure that it satisfies the coherence property. Traditionally, one would stop here: we would implement 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 lookup, the user can drive the type checker into verifying 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 $\mathsf{ilookup}$

$$\begin{array}{llllll}
\mathsf{ilookup} & (m : \mathsf{Nat}) & (vs : \mathsf{Vec}\ A\ n) & : & \mathsf{IMaybe}\ A\ (m < n) \\
\mathsf{ilookup} & m & \mathsf{nil} & \mapsto & \mathsf{nothing} \\
\mathsf{ilookup} & 0 & (\mathsf{cons}\ a\ vs) & \mapsto & \mathsf{just}\ a \\
\mathsf{ilookup} & (\mathsf{suc}\ m) & (\mathsf{cons}\ a\ vs) & \mapsto & \mathsf{ilookup}\ m\ vs
\end{array}$$

where $\mathsf{Vec}\ A$ is the reornament of $\mathsf{List}\ A$ by its length (Example 8.55), and $\mathsf{IMaybe}\ A$ is the reornament of $\mathsf{Maybe}\ A$ by its truth as computed by isJust (Example 8.64).

Indeed, the accuracy afforded by $\mathsf{ilookup}$'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 $\mathsf{nothing}$. In fact, in an interactive system such as Agda extended with Agsy [Lindblad and Benke, 2004], the user can drive the system to implement $\mathsf{ilookup}$ 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 $\mathsf{ilookup}$ is to *index* the types of lookup by their unornamented version, *i.e.* the arguments and result of $- < -$. Hence, we can make sure that the result computed by $\mathsf{ilookup}$ respects the output of $- < -$ on the unornamented indices: the result is correct *by indexing*! The type of $\mathsf{ilookup}$ is naturally derived from the ornamentation of $- < -$ into 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 $\mathsf{ilookup}$, we can extract both lookup and its proof of correctness *without having written any proof term ourselves*:

$$\begin{array}{llll}
\mathsf{lookup}\ (m\!:\!\mathsf{Nat})\ (xs\!:\!\mathsf{List}\,A) & : & \mathsf{Maybe}\,A \\
\mathsf{lookup}\quad m \qquad xs & \mapsto & \pi_0(\mathsf{forgetIMaybe}\,(\mathsf{ilookup}\,m\ (\mathsf{makeVec}\,xs)))
\end{array}$$

$$\begin{array}{lll}
\mathsf{cohLookup}\ (n\!:\!\mathsf{Nat})\ (xs\!:\!\mathsf{List}\,A) & : & \mathsf{isJust}\,(\mathsf{lookup}\,n\ xs) = n < \mathsf{length}\,xs \\
\mathsf{cohLookup}\quad m \qquad xs & \mapsto & \pi_1(\mathsf{forgetIMaybe}\,(\mathsf{ilookup}\,m\ (\mathsf{makeVec}\,xs)))
\end{array}$$

We rely on the function $\mathsf{makeVec} : (xs : \mathsf{List}\ A) \to \mathsf{Vec}\ A\ (\mathsf{length}\,xs)$, 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 ($\mathsf{ilookup}$ *vs.* $\mathsf{vlookup}$)**.** The function $\mathsf{ilookup}$ is very similar to the more familiar $\mathsf{vlookup}$ function:

$$\begin{array}{lllll}
\mathsf{vlookup} & (m\!:\!\mathsf{Fin}\,n) & (vs\!:\!\mathsf{Vec}\,A\,n) & : & A \\
\mathsf{vlookup} & \mathsf{f0} & (\mathsf{cons}\,a\,xs) & \mapsto & a \\
\mathsf{vlookup} & (\mathsf{fsuc}\,n) & (\mathsf{cons}\,a\,xs) & \mapsto & \mathsf{vlookup}\,n\,xs
\end{array}$$

These two definitions are actually equivalent, thanks to the isomorphism:

$$(m\!:\!\mathsf{Nat}) \to \mathsf{IMaybe}\,A(m < n) \cong \mathsf{Fin}\,n \to A$$

Intuitively, we can move the constraint that $m < n$ from the result – where we return an object of type $\mathsf{IMaybe}\,A(m < n)$ – to the premise – where we expect an object of type $\mathsf{Fin}\,n$. Indeed, we can think of the type $\mathsf{Fin}\,n$ as the combination of a number $m : \mathsf{Nat}$ together with a proof that $m < n$.

(9.21) 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 $- < - : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool}$
- Ornament its inductive components as desired, here Nat to $\mathsf{List}\ A$ and Bool to $\mathsf{Maybe}\ A$ in order to describe the lifting of interest, here

$$\mathsf{lookup} : \mathsf{Nat} \to \mathsf{List}\ A \to \mathsf{Maybe}\ A$$

satisfying

$$\forall n : \mathsf{Nat}.\ \forall xs : \mathsf{List}\ A.\ \ \mathsf{isJust}\ (\mathsf{lookup}\ n\ xs) = n < \mathsf{length}\ xs$$

- Implement a carefully indexed version of the lifting, here

$$\mathsf{ilookup} : (m : \mathsf{Nat})(vs : \mathsf{Vec}\ A\ n) \to \mathsf{IMaybe}\ A\ (m < n)$$

- Derive the lifting, here lookup, and its coherence proof, without *writing* a proof!

Besides, the implementation of ilookup is useful on its own: this function corresponds exactly to vector lookup, a function that one would have implemented anyway.

(9.22) This manual unfolding of the lifting is instructive: it involves many constructions on datatypes (here, the datatypes $\mathsf{List}\ A$ and $\mathsf{Maybe}\ A$) as well as on functions (here, the type of ilookup, the definition of 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

(9.23) 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`

9.24 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:

$$\begin{array}{l} \textbf{data}\ \mathsf{Type} : \mathrm{SET}_1\ \textbf{where} \\ \quad \mathsf{Type}\ \ni\ \mu\{(D : \mathsf{func}\,K\,K)\ (k : K)\} \rightarrow (T : \mathsf{Type}) \\ \qquad\quad\ \ \mid\ \mu\{(D : \mathsf{func}\,K\,K)\ (k : K)\} \times (T : \mathsf{Type}) \\ \qquad\quad\ \ \mid\ 1 \end{array}$$

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:

$$\begin{array}{lcl} [\![(T : \mathsf{Type})]\!]_{\mathsf{Type}} & : & \mathrm{SET} \\ [\![\mu\{D\,k\} \rightarrow T]\!]_{\mathsf{Type}} & \mapsto & \mu\,D\,k \rightarrow [\![T]\!]_{\mathsf{Type}} \\ [\![\mu\{D\,k\} \times T]\!]_{\mathsf{Type}} & \mapsto & \mu\,D\,k \times [\![T]\!]_{\mathsf{Type}} \\ [\![1]\!]_{\mathsf{Type}} & \mapsto & \mathbb{1} \end{array}$$

(9.25) 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.

9.26 Example (Coding $- < -$)**.** Written in the universe of function types, the type of the comparison function is:

$$\begin{array}{lcl} \mathsf{type}{<} & : & \mathsf{Type} \\ \mathsf{type}{<} & \mapsto & \mu\{\mathsf{NatD} *\} \rightarrow \mu\{\mathsf{NatD} *\} \rightarrow \mu\{\mathsf{BoolD} *\} \times 1 \end{array}$$

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* ($\Leftarrow$) gadget:

$$\begin{array}{lcl}
- < - & : & [\![\mathsf{type}{<}]\!]_{\mathsf{Type}} \\
m < n & \Leftarrow & \mathsf{Nat\text{-}case}\, n \\
\quad m < 0 & \mapsto & (\mathsf{false}, *) \\
\quad m < \mathsf{suc}\, n & \Leftarrow & \mathsf{Nat\text{-}elim}\, m \\
\qquad 0 < \mathsf{suc}\, n & \mapsto & (\mathsf{true}, *) \\
\qquad \mathsf{suc}\, m < \mathsf{suc}\, n & \mapsto & m < n
\end{array}$$

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:

$$\begin{array}{lcl}
\mathsf{type}{+} & : & \mathsf{Type} \\
\mathsf{type}{+} & \mapsto & \mu\{\mathsf{NatD}\,*\} \!\to \mu\{\mathsf{NatD}\,*\} \!\to \mu\{\mathsf{NatD}\,*\} \!\times 1
\end{array}$$

Again, up to a trivial multiplication by $\mathbb{1}$, the implementation of $- + -$ is left unchanged:

$$\begin{array}{lcl}
- + - & : & [\![\mathsf{type}{+}]\!]_{\mathsf{Type}} \\
m + n & \Leftarrow & \mathsf{Nat\text{-}elim}\, m \\
\quad 0 + n & \mapsto & (n, *) \\
\quad \mathsf{suc}\, m + n & \mapsto & (\mathsf{suc}\, m + n, *)
\end{array}$$

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:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{FunOrn}\ (T : \mathsf{Type}) : \textsc{Set}_1\ \textbf{where} \\
\quad \mathsf{FunOrn}\ (\mu\{D\,k\} \mapsto T)\ \ni\ \mu^{+}\{(o : \mathsf{orn}\ D\ u\ u)\ (i : u^{-1}\ k)\} \mapsto (T^{+} : \mathsf{FunOrn}\ T) \\
\quad \mathsf{FunOrn}\ (\mu\{D\,k\} \times T)\ \ni\ \mu^{+}\{(o : \mathsf{orn}\ D\ u\ u)\ (i : u^{-1}\ k)\} \times (T^{+} : \mathsf{FunOrn}\ T) \\
\quad \mathsf{FunOrn}\qquad 1 \qquad\quad \ni\ 1
\end{array}
$$

9.30 Definition (Lifting type)**.** We get the type of the liftings by interpreting the ornaments as we traverse the functional ornament:

$$
\begin{array}{lcl}
[\![(T^{+} : \mathsf{FunOrn}\ T)]\!]_{\mathsf{FunOrn}} & : & \textsc{Set} \\
[\![\mu^{+}\{o\ (\mathsf{inv}\,i)\} \mapsto T^{+}]\!]_{\mathsf{FunOrn}} & \mapsto & \mu\ [\![o]\!]_{\mathsf{orn}}\ k \to [\![T^{+}]\!]_{\mathsf{FunOrn}} \\
[\![\mu^{+}\{o\ (\mathsf{inv}\,i)\} \times T^{+}]\!]_{\mathsf{FunOrn}} & \mapsto & \mu\ [\![o]\!]_{\mathsf{orn}}\ k \times [\![T^{+}]\!]_{\mathsf{FunOrn}} \\
[\![1]\!]_{\mathsf{FunOrn}} & \mapsto & \mathbb{1}
\end{array}
$$

(9.31) We want our ornamented function to be *coherent* with respect to the base function we started from: for a function $f : \mu\, D \to \mu\, E$, the ornamented function

$$f^{+} : \mu\ [\![o_D]\!]_{\mathsf{orn}} \to \mu\ [\![o_E]\!]_{\mathsf{orn}}$$

is said to be coherent with f if the following diagram commutes:

$$\begin{tikzcd}
\mu\ [\![o_D]\!]_{\mathsf{orn}} \arrow[r, "f^{+}"] \arrow[d, "\mathsf{forget}\ o_D"'] & \mu\ [\![o_E]\!]_{\mathsf{orn}} \arrow[d, "\mathsf{forget}\ o_E"] \\
\mu\, D \arrow[r, "f"'] & \mu\, E
\end{tikzcd}$$

Or, equivalently in type theory:

$$\forall x^{+} : \mu\ [\![o_D]\!]_{\mathsf{orn}}\ i.\ f\ (\mathsf{forget}\ o_D\ x^{+}) = \mathsf{forget}\ o_E\ (f^{+}\ x^{+})$$

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:

$$\begin{array}{llll}
\mathsf{Coherence}(T^{+} : \mathsf{FunOrn}\, T)(f : [\![T]\!]_{\mathsf{Type}})(f^{+} : [\![T^{+}]\!]_{\mathsf{FunOrn}}) : \textsc{Set} \\
\mathsf{Coherence}\ (\mu^{+}\{o\ (\mathsf{inv}\, i)\}\!\!\rightarrow T^{+}) & f & f^{+} & \mapsto \\
\qquad\qquad \forall x^{+} : \mu\, [\![o]\!]_{\mathsf{orn}}\, k.\ \mathsf{Coherence}\, T^{+}\ (f\ (\mathsf{forgetOrn}\, x^{+}))\ (f^{+} x^{+}) \\
\mathsf{Coherence}\ (\mu^{+}\{o\ (\mathsf{inv}\, i)\}\!\times T^{+}) & (x, xs) & (x^{+}, xs^{+}) & \mapsto \\
\qquad\qquad x = \mathsf{forgetOrn}\, x^{+} \times \mathsf{Coherence}\, T^{+}\ xs\ xs^{+} \\
\mathsf{Coherence}\quad 1 & * & * & \mapsto \mathbb{1}
\end{array}$$

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 Nat to List A (Example 8.25), and Bool to Maybe A (Example 8.22). From there, the functional ornament describing the type of the lookup function is as follows:

$$\begin{array}{lll}
\mathsf{typeLookup} & : & \mathsf{FunOrn}\,\mathsf{type}{<} \\
\mathsf{typeLookup} & \mapsto & \mu^{+}\{\mathsf{idO}_{\mathsf{Nat}} *\}\!\!\rightarrow \mu^{+}\{\mathsf{ListO}\, A *\}\!\!\rightarrow \mu^{+}\{\mathsf{MaybeO}\, A *\}\!\times 1
\end{array}$$

The reader checks that $[\![\mathsf{typeLookup}]\!]_{\mathsf{FunOrn}}$ gives us the type of the lookup function, up to a multiplication by $\mathbb{1}$. Also, unfolding the coherence condition gives the desired property:

$$\begin{array}{c}
\mathsf{Coherence}\,\mathsf{typeLookup}\ (- < -) \rightsquigarrow \lambda f^{+} : [\![\mathsf{typeLookup}]\!]_{\mathsf{FunOrn}}. \\
\forall n : \mathsf{Nat}.\ \forall xs : \mathsf{List}\, A.\ \mathsf{isJust}\ (f^{+}\ n\ xs) = n < \mathsf{length}\, xs
\end{array}$$

9.34 Remark. This equation is not *specifying* the lookup function: it is only establishing a computational relation between $- < -$ and a candidate lifting f^{+}, for which 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.

9.35 Example (Ornamenting type+ to describe $- +\!\!+ -$)**.** The functional ornament of type+ relies solely on the ornamentation of Nat into List A:

$$\begin{array}{lll}
\mathsf{type}{+\!\!+} & : & \mathsf{FunOrn}\,\mathsf{type}{+} \\
\mathsf{type}{+\!\!+} & \mapsto & \mu^{+}\{\mathsf{ListO}\, A *\}\!\!\rightarrow \mu^{+}\{\mathsf{ListO}\, A *\}\!\!\rightarrow \mu^{+}\{\mathsf{ListO}\, A *\}\!\times 1
\end{array}$$

Again, we check that $[\![\mathsf{type}{+\!\!+}]\!]_{\mathsf{FunOrn}}$ is indeed the type of $- +\!\!+ -$ while the coherence condition $\mathsf{Coherence}\,\mathsf{type}{+\!\!+}\ (- + -)$ 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.

9.2.3. Patches

MODEL: `Chapter9.Patch`

(9.36) By definition of a functional ornament, the lifting of a base function $f : [\![T]\!]_{\mathsf{Type}}$ is a function f^+ of type $[\![T^+]\!]_{\mathsf{FunOrn}}$ satisfying the coherence property $\mathsf{Coherence}\, T^+\, f$. 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 Patch as the type of *all* the functions that are coherent by construction.

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:

$$\mathsf{Patch}\, T\, T^+\, f \cong (f^+ : [\![T^+]\!]_{\mathsf{FunOrn}}) \times \mathsf{Coherence}\, T^+\, f\, f^+ \tag{9.1}$$

(9.38) In this chapter, we constructively use this bijection in the left-to-right direction: having implemented a patch f^{++} of type $\mathsf{Patch}\, T\ \ T^+\, f$, we show how we can extract a lifting together with its coherence proof.

9.39 Example (Patching $- < -$)**.** Before giving the general construction of a Patch, let us first work through our running example. After having functionally ornamented $- < -$ with typeLookup, the lifting function f^+ and coherence property can be represented by the following commuting diagram:

$$\begin{tikzcd}
\mathsf{Nat} \arrow[d, equal, "\mathsf{id}"'] & \times & \mathsf{List}\, A \arrow[r, "f^+"] \arrow[d, "\mathsf{length}"'] & \mathsf{Maybe}\, A \arrow[d, "\mathsf{isJust}"] \\
\mathsf{Nat} & \times & \mathsf{Nat} \arrow[r, "- < -"'] & \mathsf{Bool}
\end{tikzcd} \tag{9.2}$$

In type theory, this is written as:

$$\begin{aligned}&(f^{+} : \mathsf{Nat} \times \mathsf{List}\, A \to \mathsf{Maybe}\, A) \times \\ &\qquad \forall m : \mathsf{Nat}.\ \forall as : \mathsf{List}\, A.\ m < \mathsf{length}\, as = \mathsf{isJust}\, (f^{+}\ m\ as)\end{aligned}$$

Applying intensional choice, this is equivalent to:

$$\begin{aligned}&\cong (m : \mathsf{Nat}) \times (n : \mathsf{Nat}) \times (as : \mathsf{List}\, A) \times \mathsf{length}\, as = n \to \\ &\qquad (ma : \mathsf{Maybe}\, A) \times \mathsf{isJust}\, ma = m < n\end{aligned}$$

Now, by definition of reornaments, we have that:

$$\begin{aligned}(as : \mathsf{List}\, A) \times \mathsf{length}\, as = n &\cong \mathsf{Vec}\, A\, n \qquad\qquad \text{and} \\ (ma : \mathsf{Maybe}\, A) \times \mathsf{isJust}\, ma = b &\cong \mathsf{IMaybe}\, A\, b\end{aligned}$$

Applying these isomorphisms, we obtain the following type, which we call the Patch of the functional ornament typeLookup:

$$\cong (m : \mathsf{Nat}) \times (n : \mathsf{Nat}) \times (vs : \mathsf{Vec}\, A\, n) \to \mathsf{IMaybe}\, A\, (m < n)$$

This last type is thus equivalent to a pair of a lifting and its coherence proof.

(9.40) Intuitively, the 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 ornamented 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 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 $\mu^{+}\{o\ \}\!\!\rightarrowtail$), we introduce a fresh index and the reornament of o. Upon a result (*i.e.* a code $\mu^{+}\{o\ \}\times$), we ask for a reornament of o indexed by the result of the base function.

$$
\begin{array}{llll}
\mathsf{Patch}\ \ (T : \mathsf{Type}) & (T^{+} : \mathsf{FunOrn}\ T) & (f : [\![T]\!]_{\mathsf{Type}}) & : \ \textsc{Set} \\
\mathsf{Patch}\ (\mu\{D\ (u\ i)\ \}\!\!\to T) & (\mu^{+}\{o\ (\mathsf{inv}\ i)\ \}\!\!\to T^{+}) & f & \mapsto \\
\quad (x : \mu\, D\ (u\ i)) \to \mu\ \lceil o \rceil\ (i, x) \to \mathsf{Patch}\ T\ T^{+}\ (f\ x) & & & \\
\mathsf{Patch}\ (\mu\{D\ (u\ i)\ \}\!\times T) & (\mu^{+}\{o\ (\mathsf{inv}\ i)\ \}\!\times T^{+}) & (x, xs) & \mapsto \\
\quad \mu\ \lceil o \rceil\ (i, x) \times \mathsf{Patch}\ T\ T^{+}\ xs & & & \\
\mathsf{Patch}\qquad 1 & \qquad 1 & \ast & \mapsto \mathbb{1}
\end{array}
$$

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:

$$(m : \mathsf{Nat})(m^{+} : \mu\ \lceil \mathsf{idO}_{\mathsf{Nat}} \rceil\ m) \to (n : \mathsf{Nat})(vs : \mu\ \lceil \mathsf{List}\ A \rceil\ n) \to \mu\ \lceil \mathsf{Maybe}\ A \rceil\ (m < n) \times \mathbb{1}$$

9.43 Remark. $\mu\ \lceil \mathsf{idO}_{\mathsf{Nat}} \rceil\ n$ 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, $\lceil \mathsf{List}\ A \rceil$ and $\lceil \mathsf{Maybe}\ A \rceil$ 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

$$(m : \mathsf{Nat})(xs : \lceil \mathsf{List}\ A \rceil\ m) \to (n : \mathsf{Nat})(ys : \lceil \mathsf{List}\ A \rceil\ m) \to \lceil \mathsf{List}\ A \rceil\ (m + n) \times \mathbb{1}$$

where, again, the datatype $\lceil \mathsf{List}\ A \rceil$ 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:

$$
\begin{array}{l}
\forall T : \mathsf{Type}.\ \forall T^{+} : \mathsf{FunOrn}\ T.\ \forall f : [\![T]\!]_{\mathsf{Type}}. \\
\qquad\qquad \mathsf{Patch}\ T\ T^{+}\ f \cong (f^{+} : [\![T^{+}]\!]_{\mathsf{FunOrn}}) \times \mathsf{Coherence}\ T^{+}\ f\ f^{+}
\end{array}
$$

That is, our definition of the Patch type enforces that its inhabitants are exactly those liftings that are coherent by construction.

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:

$$\begin{array}{l}(f^{+} : \mu\, [\![o_A]\!]_{\mathsf{orn}} \to \mu\, [\![o_B]\!]_{\mathsf{orn}}) \times \\ \qquad \forall a^{+} : \mu\, [\![o_A]\!]_{\mathsf{orn}}.\, \mathsf{forget}\, o_B\, (f^{+}\, a^{+}) = f(\mathsf{forget}\, o_A\, a^{+})\end{array}$$

Applying intensional choice, we obtain the following equivalent type:

$$\cong (a : \mu\, A) \times (a^{+} : \mu\, [\![o_A]\!]_{\mathsf{orn}}) \times \mathsf{forget}\, o_A\, a^{+} = a \to (b^{+} : \mu\, [\![o_B]\!]_{\mathsf{orn}}) \times \mathsf{forget}\, o_B\, b^{+} = f a$$

Then, we can simply use the characterisation of a reornament to turn every pair $(x^{+} : \mu\, [\![o_X]\!]_{\mathsf{orn}}) \times \mathsf{forget}\, o_X\, x^{+} = t$ into the equivalent inductive type $\mu\, \lceil o_X \rceil\, t$

$$\cong (a : \mu\, A) \times \mu\, \lceil o_A \rceil\, a \to \mu\, \lceil o_B \rceil (f\, a)$$

which corresponds to the Patch type of this functional ornament.

□

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:

$$\begin{array}{lcl}\mathsf{vrev}\ (xs : \mathsf{Vec}\ A\ n) & : & \mathsf{Vec}\ A\ n \\ \mathsf{vrev}\quad \mathsf{nil} & \mapsto & \mathsf{nil} \\ \mathsf{vrev}\quad (\mathsf{cons}\, a\, vs) & \mapsto & \{(\mathsf{vrev}\, vs) +\!\!+ (\mathsf{cons}\, a\, \mathsf{nil}) : \mathsf{Vec}\ A\ (1 + n)\}\end{array}$$

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 Vec A $(n+1)$ while the expected type is 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.

9.2.4. Patching and coherence

MODEL: `Chapter9.Patch.Apply, Chapter9.Patch.Coherence`

(9.47) 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 $\mu^{+}\{o\ \}\!\mapsto$), we simply patch the body of the function. This is possible because from $x^{+} : \mu\ [\![o_D]\!]_{\mathsf{orn}}$, 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 $\mu^{+}\{o\ \}\times$), we simply forget the reornamentation computed by the coherent lifting:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathsf{patch}\ (T^{+} : \mathsf{FunOrn}\ T)\ (f : [\![T]\!]_{\mathsf{Type}})\ (p : \mathsf{Patch}\ T\ T^{+}\ f)\ :\ [\![T^{+}]\!]_{\mathsf{FunOrn}}} \\
\mathsf{patch}\ (\mu^{+}\{o\ (\mathsf{inv}\ i)\ \}\!\mapsto T^{+}) & f & f^{++} & \mapsto \\
\multicolumn{4}{l}{\quad \lambda x^{+}.\ \mathsf{patch}\ (f\ (\mathsf{forgetOrn}\ x^{+}))} \\
\multicolumn{4}{l}{\qquad\qquad (f^{++}\ (\mathsf{forgetOrn}\ x^{+})\ (\mathsf{makeAlgOrn}\ x^{+}))} \\
\mathsf{patch}\ (\mu^{+}\{o\ (\mathsf{inv}\ i)\ \}\times T^{+}) & (x, xs) & (x^{++}, xs^{++}) & \mapsto \\
\multicolumn{4}{l}{\quad (\mathsf{forgetOrn}\ x^{++}, \mathsf{patch}\ T^{+}\ xs\ xs^{++})} \\
\mathsf{patch}\quad 1 & * & * & \mapsto *
\end{array}
$$

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:

$$
\begin{array}{llll}
\mathsf{coherence}\,(T^{+} : \mathsf{FunOrn}\,T)\,(f : [\![T]\!]_{\mathsf{Type}})\,(p : \mathsf{Patch}\,T\,T^{+}\,f) \;:\; \mathsf{Coherence}\,T^{+}\,f\,(\mathsf{patch}\,T^{+}\,f\,p) \\
\mathsf{coherence}\,(\mu^{+}\{o\,(\mathsf{inv}\,i)\,\}\!\!\to T^{+}) \quad f \quad p \quad \mapsto \\
\qquad \lambda x^{+}.\,\mathsf{coherence}T^{+}\,(f\,(\mathsf{forgetOrn}\,x^{+})) \\
\qquad\qquad\qquad (p\,(\mathsf{forgetOrn}\,x^{+})\,(\mathsf{makeAlgOrn}\,x^{+})) \\
\mathsf{coherence}\,(\mu^{+}\{o\,(\mathsf{inv}\,i)\,\}\!\times T^{+})\,(x, xs)\,(x^{+}, p) \mapsto \\
\qquad (\mathsf{coherentOrn}\,x^{+}, \mathsf{coherence}\,T^{+}\,xs\,p) \\
\mathsf{coherence} \quad 1 \quad * \quad * \quad \mapsto *
\end{array}
$$

9.50 Example (Obtaining lookup and its coherence, for free)**.** This last step is a mere application of the patch and coherence functions. Hence, we define lookup as follows:

$$
\begin{array}{lcl}
\mathsf{lookup} & : & [\![\mathsf{typeLookup}]\!]_{\mathsf{FunOrn}} \\
\mathsf{lookup} & \mapsto & \mathsf{patch}\,\mathsf{typeLookup}\,(- < -)\,\mathsf{ilookup}
\end{array}
$$

And we get its coherence proof, here spelled in full:

$$
\begin{array}{llll}
\mathsf{cohLookup}\,(n : \mathsf{Nat})\,(xs : \mathsf{List}\,A) & : & \mathsf{length}\,(\mathsf{lookup}\,n\,xs) = n < \mathsf{length}\,xs \\
\mathsf{cohLookup}\quad n \quad xs & \mapsto & \mathsf{coherence}\,\mathsf{typeLookup}\,(- < -)\,\mathsf{ilookup}\,n\,xs
\end{array}
$$

9.51 Remark (Code readability)**.** The lookup function thus defined is rather daunting, especially for a potential user of that piece of code. However, we must bear in mind that lookup is in fact entirely specified by ilookup: there is no point in inspecting the definition of lookup. In a programming environment, we could imagine a syntactic device akin to our informal syntax for ornaments. For instance, we would state that lookup is a functional ornamentation of $- < -$. This would lead us to – transparently – implement ilookup in lieu of lookup.

9.52 Example (Obtaining $- +\!\!+ -$ and its coherence, for free)**.** Assuming that we have implemented the coherent lifting vappend, we obtain concatenation of lists and its coherence proof by simply running our generic machinery:

$$\begin{array}{lll}
{+\!\!+} & : & [\![\mathsf{type}{+\!\!+}]\!]_{\mathsf{FunOrn}} \\
{+\!\!+} & \mapsto & \mathsf{patch}\,\mathsf{type}{+\!\!+}\ (- + -)\ \mathsf{vappend}
\end{array}$$

$$\begin{array}{llll}
\mathsf{coh}{+\!\!+}\ (xs : \mathsf{List}\ A) & (ys : \mathsf{List}\ A) & : & \mathsf{length}\,(xs \mathbin{+\!\!+} ys) = (\mathsf{length}\,xs) + (\mathsf{length}\,ys) \\
\mathsf{coh}{+\!\!+}\qquad xs & \qquad ys & \mapsto & \mathsf{coherence}\,\mathsf{type}{+\!\!+}\ (- + -)\ \mathsf{vappend}\ xs\ ys
\end{array}$$

(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 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 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 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 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.

```
−         < −    : ⟦type<⟧Type   ilookup      (m : Nat) (vs : Vec A n)  :  IMaybe A (m < n)
m         < n    ⇐ Nat-case n    ilookup       m         vs             ⇐  Vector-case vs
  m       < 0    ↦ (false, ∗)      ilookup     m         nil            ↦  nothing
  m       < suc n ⇐ Nat-elim m      ilookup     m        (cons a vs)     ⇐  Nat-elim m
    0     < suc n ↦ (true, ∗)         ilookup   0        (cons a vs)     ↦  just a
    suc m < suc n ↦ m < n             ilookup  (suc m)   (cons a vs)     ↦  ilookup m vs
```

Figure 9.2.: Implementations of − < − and ilookup

9.3. Lazy Programmers, Smart Constructors

MODEL: `Chapter9.Lift`

(9.56) In our journey from − < − to lookup, we had to implement the ilookup function. It is instructive to put − < − and ilookup side-by-side (Figure 9.2). First, both functions follow the same recursion pattern: case analysis over n/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 − < −, 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 − < − – to guide the implementation of the coherent lifting – such as ilookup: instead of re-implementing ilookup by duplicating most of the code of − < −, 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 $(\!|\alpha|\!)$ and the user provides us with a *coherent algebra* $\hat{\alpha}$ of α, we construct the coherent lifting $(\!|\hat{\alpha}|\!)$ of $(\!|\alpha|\!)$;

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.

(9.58) 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

(9.59) 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.

9.60 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:

$$\begin{tikzcd}
\mu \llbracket o_D \rrbracket_{\mathsf{orn}} \arrow[r, "(\!|\beta|\!)"] \arrow[d, "\mathsf{forget}\, o_D"'] & \mu \llbracket o_E \rrbracket_{\mathsf{orn}} \arrow[d, "\mathsf{forget}\, o_E"] \\
\mu\, D \arrow[r, "(\!|\alpha|\!)"'] & \mu\, E
\end{tikzcd}$$

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:

$$\begin{tikzcd}
\llbracket o_D \rrbracket_{\mathsf{orn}}\, \mu \llbracket o_E \rrbracket_{\mathsf{orn}} \arrow[rr, "\beta"] \arrow[d, "\llbracket o_D \rrbracket\ (\mathsf{forget}\, o_E)"] & & \mu \llbracket o_E \rrbracket_{\mathsf{orn}} \arrow[d, "\mathsf{forget}\, o_E"'] \\
\llbracket o_D \rrbracket_{\mathsf{orn}}\, \mu\, E \arrow[r, "\mathsf{forgetNT}\, o_D"'] & \llbracket D \rrbracket\, \mu\, E \arrow[r, "\alpha"'] & \mu\, E
\end{tikzcd}$$

Now, we would like to find an algebra $\hat{\alpha}$ such that its catamorphism gives us a function of the Patch type.

9.61 Example (Lifting isSuc)**.** To illustrate our approach, let us work through a concrete example: we derive hd : List A → Maybe A from isSuc : Nat → Bool by transporting the algebra. For the sake of argument, we artificially define isSuc by a catamorphism:

$$\begin{array}{llll}
\mathsf{isSuc}\,(n\!:\!\mathsf{Nat}) & : & \mathsf{Bool} & \\
\mathsf{isSuc}\quad n & \mapsto & (\!|\mathsf{isSucAlg}|\!)\,n & \textbf{where} \\
\quad \mathsf{isSucAlg}\,(xs\!:\![\![\mathsf{NatD}]\!]\,\mathsf{Bool}) & : & \mathsf{Bool} & \\
\quad \mathsf{isSucAlg}\quad ['0] & \mapsto & \mathsf{false} & \\
\quad \mathsf{isSucAlg}\quad ['\mathsf{suc}\,xs] & \mapsto & \mathsf{true} &
\end{array}$$

Our objective is thus to define the algebra for hd, which has the following type

$$\mathsf{hdAlg}\!:\![\![\mathsf{ListD}\ A]\!]\ \mathsf{Maybe}\ A \to \mathsf{Maybe}\ A$$

such that its catamorphism is coherent. By the fold-fusion theorem, it is sufficient for hdAlg to satisfy the following condition:

$$\begin{array}{l}
\forall ms\!:\![\![\mathsf{ListD}\ A]\!]\ \mathsf{Maybe}\ A. \\
\quad \mathsf{isJust}\,(\mathsf{hdAlg}\,ms) = \mathsf{isSucAlg}\,(\mathsf{forgetNT}\ (\mathsf{ListO}\ A)\ ([\![\mathsf{ListD}\ A]\!]\ \mathsf{isJust}\ ms))
\end{array}$$

Following the same methodology we applied to define the Patch type, we can massage the type of hdAlg and its coherence condition to obtain an equivalent definition enforcing the coherence by indexing. In this case, we obtain the type:

$$\mathsf{lift}^{\mathsf{hdAlg}}_{\mathsf{Alg}} \triangleq [\![\mathsf{VecD}\ A]\!]\ (\lambda n'.\,\mathsf{IMaybe}\ A\ (\mathsf{isSuc}\,n'))\ n \to \mathsf{IMaybe}\ A\ (\mathsf{isSuc}\,n)$$

This construction generalises to any functional ornament.

9.62 Definition (Coherent algebra)**.** Let α be an algebra

$$\alpha\!:\!(k\!:\!K) \to [\![D]\!]\ (\lambda - .\,[\![T]\!]_{\mathsf{Type}})\ k \to [\![T]\!]_{\mathsf{Type}}$$

together with an ornament $o_D\!:\!\mathsf{orn}\ D\ u\ u$ and a functional ornament $T^{+}\!:\!\mathsf{FunOrn}\ T$.

We define the *coherent algebras* for α to be the inhabitants of the type:

$$\begin{array}{l}
\mathsf{lift}^{\alpha}_{\mathsf{Alg}} \triangleq (i\!:\!I)(t\!:\!\mu\,D\ (u\ i)) \to \\
\qquad\qquad [\![\lceil o \rceil]\!]\ (\lambda(i,t).\,\mathsf{Patch}\ T\ ((\!|\alpha|\!)\ t)\ T^{+})\ (i,t) \to \mathsf{Patch}\ T\ ((\!|\alpha|\!)\ t)\ T^{+}
\end{array}$$

9.63 Definition (Lifting of coherent algebra)**.** Constructively, we get that coherent algebras induce coherent liftings, by the catamorphism of the coherent algebra:

$$\begin{array}{l}
\mathsf{lift\text{-}fold}\ (\alpha : (k : K) \to [\![D]\!]\ (\lambda - .\ [\![T]\!]_{\mathsf{Type}})\ k \to [\![T]\!]_{\mathsf{Type}}) \\
\qquad\qquad (\hat{\alpha} : \mathsf{lift}^{\alpha}_{\mathsf{Alg}})\ :\ \mathsf{Patch}\ (\mu\{D\ (u\ i)\ \}\!\to T)\ (\!|\alpha|\!)\ (\mu^{+}\{o\ i\ \}\!\to T^{+}) \\
\mathsf{lift\text{-}fold}\ \alpha\ \hat{\alpha}\ \mapsto\ \lambda x.\ \lambda x^{++}.\ (\!|\hat{\alpha}|\!)\ x^{++}
\end{array}$$

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 262). 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:

$$\begin{array}{l}
\begin{array}{lcl}
\mathsf{ihd}\ (vs : \mathsf{Vec}\ A\ n) & : & \mathsf{IMaybe}\ A\ (\mathsf{isSuc}\ n) \\
\mathsf{ihd}\quad vs & \mapsto & \mathsf{lift\text{-}fold}\ \mathsf{isSucAlg}\ \mathsf{ihdAlg}
\end{array}\quad \textbf{where} \\
\quad \mathsf{ihdAlg}\ (vs : [\![\mathsf{VecD}\ A]\!]\ (\lambda n'.\ \mathsf{IMaybe}\ A\ (\mathsf{isSuc}\ n'))\ n)\ :\ \mathsf{IMaybe}\ A\ (\mathsf{isSuc}\ n) \\
\quad \mathsf{ihdAlg}\quad [\text{'}\mathsf{nil}]\quad \mapsto\ \{?\} \\
\quad \mathsf{ihdAlg}\ [\text{'}\mathsf{cons}\ a\ xs]\ \mapsto\ \{?\}
\end{array}$$

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 α be an inductive step

$$\alpha : (k : K)(xs : [\![D]\!]\ (\mu\ D)\ k) \to \Box_{D}\ (\lambda - .\ [\![T]\!]_{\mathsf{Type}})\ xs \to [\![T]\!]_{\mathsf{Type}}$$

together with an ornament $o_D : \mathsf{orn}\ D\ u\ u$ and a functional ornament $T^{+} : \mathsf{FunOrn}\ T$.

We define the *coherent inductive step* for α to be the inhabitants of the type:

$$\begin{array}{l}
\mathsf{lift}^{\alpha}_{\mathsf{IH}} \triangleq (i : I)(t : \mu\ D\ (u\ i))(xs : [\![\lceil o \rceil]\!]\ (\mu\ \lceil o \rceil)\ (i, t)) \to \\
\qquad\qquad \Box_{\lceil o \rceil}\ (\lambda (i, t).\ \mathsf{Patch}\ T\ ((\mathsf{iinduction}\ \alpha)\ t)\ T^{+})\ xs \to \mathsf{Patch}\ T\ ((\mathsf{iinduction}\ \alpha)\ t)\ T^{+}
\end{array}$$

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:

$$\begin{array}{l}
\mathsf{lift\text{-}ind}\ (\alpha : (k : K)(xs : [\![D]\!]\ (\mu\,D)\ k) \to \Box_D\ (\lambda - .\ [\![T]\!]_{\mathsf{Type}})\ xs \to [\![T]\!]_{\mathsf{Type}}) \\
\qquad (\hat{\alpha} : \mathsf{lift}^{\alpha}_{\mathsf{IH}})\ :\ \mathsf{Patch}\ (\mu\{D\ (u\ i)\} \!\to T)\ (\mathsf{iinduction}\ \alpha)\ (\mu^{+}\{o\ i\} \!\to T^{+}) \\
\mathsf{lift\text{-}ind}\ \alpha\ \hat{\alpha}\ \mapsto\ \lambda x.\, \lambda x^{++}.\ \mathsf{iinduction}\ \hat{\alpha}\ x^{++}
\end{array}$$

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).

$$\begin{array}{l}
\mathsf{lift\text{-}case}\ (\alpha : (k : K)(xs : [\![D]\!]\ (\mu\,D)\ k) \to [\![T]\!]_{\mathsf{Type}}) \\
\qquad (\hat{\alpha} : \mathsf{lift}^{(\lambda xs\ -.\ \alpha\ xs)}_{\mathsf{IH}})\ :\ \mathsf{Patch}\ (\mu\{D\ (u\ i)\} \!\to T) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{iinduction}\ (\lambda xs\ -.\ \alpha\ xs)) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mu^{+}\{o\ i\} \!\to T^{+}) \\
\mathsf{lift\text{-}case}\ \alpha\ \hat{\alpha}\ \mapsto\ \mathsf{lift\text{-}ind}\ (\lambda xs\ -.\ \alpha\ xs)\ (\lambda xs\ -.\ \hat{\alpha}\ xs)
\end{array}$$

9.69 Example (Transporting the recursion pattern of $- < -$)**.** To implement ilookup, we use lift-case to transport the case analysis on n:

$$\begin{array}{lllll}
\multicolumn{5}{l}{\mathsf{ilookup}\ :\ \mathsf{Patch}\ \mathsf{type}{<}\ \mathsf{typeLookup}\ - < -} \\
\mathsf{ilookup}\ \ m\ im & n & vs & \overset{\mathsf{lift}}{\Leftarrow} & \mathsf{lift\text{-}case} \\
\quad \mathsf{ilookup}\ m\ im & 0 & \mathsf{nil} & \{?\} & \\
\quad \mathsf{ilookup}\ m\ im & (\mathsf{suc}\ n) & (\mathsf{cons}\ a\ vs) & \{?\} &
\end{array}$$

Followed by lift-ind to transport the induction over m:

$$\begin{array}{lllllll}
\multicolumn{7}{l}{\mathsf{ilookup}\ :\ \mathsf{Patch}\ \mathsf{type}{<}\ \mathsf{typeLookup}\ - < -} \\
\mathsf{ilookup} & m & im & n & vs & \overset{\mathsf{lift}}{\Leftarrow} & \mathsf{lift\text{-}case} \\
\quad \mathsf{ilookup} & m & im & 0 & \mathsf{nil} & \{?\} & \\
\quad \mathsf{ilookup} & m & im & (\mathsf{suc}\ n) & (\mathsf{cons}\ a\ vs) & \overset{\mathsf{lift}}{\Leftarrow} & \mathsf{lift\text{-}ind} \\
\qquad \mathsf{ilookup} & 0 & 0 & 0 & \mathsf{nil} & \{?\} & \\
\qquad \mathsf{ilookup} & (\mathsf{suc}\ m) & (\mathsf{suc}\ im) & 0 & \mathsf{nil} & \{?\} &
\end{array}$$

The interactive nature of this construction is crucial: the user types in the $\overset{\mathsf{lift}}{\Leftarrow}$ 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:

$$\begin{array}{llllll}
\mathsf{vappend} & : \ \mathsf{Patch}\,\mathsf{type+}\ \mathsf{type{+}{+}} & - + - & & & \\
\mathsf{vappend} & m \qquad\qquad xs & n & ys & \overset{\mathsf{lift}}{\Leftarrow} & \mathsf{lift\text{-}case} \\
\quad \mathsf{vappend} & 0 \qquad\qquad\ \mathsf{nil} & n & ys & \{?\} & \\
\quad \mathsf{vappend} & (\mathsf{suc}\, m)\ (\mathsf{cons}\, a\ xs) & n & ys & \{?\} &
\end{array}$$

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.

(9.72) Recall that, when implementing the coherent lifting, we are working on the reornaments 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 arguments.

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 arguments of the datatype, nothing more. In terms of implementation, this is as simple as:

$$\begin{array}{lr}
\mathsf{lift\text{-}constructor}\ (e : \mathsf{Extension}\ (o\ i)\ xs) & \textit{– coherent extension}\\
\qquad\qquad (a : [\![\mathsf{Structure}\ (o\ i)\ xs\ e]\!]_{\mathsf{orn}}\ (\mu\ \lceil o \rceil)) & \textit{– arguments}\\
\qquad\qquad (t^{++} : \mathsf{Patch}\ T\ t\ T^{+}) & \\
\qquad\qquad : \mathsf{Patch}\ (\mu\{D\ (u\ i)\}\times T)\ (\mathsf{in}\ xs, t)\ (\mu^{+}\{o\ i\}\times T^{+}) & \\
\mathsf{lift\text{-}constructor}\ e\ a\ t^{++}\ \mapsto\ (\mathsf{in}\ (e, a), t^{++}) &
\end{array}$$

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 ($\mathbb{1}$) 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 would be represented as follows:

$$\begin{array}{llllll}
\multicolumn{6}{l}{\mathsf{ilookup}\ :\ \mathsf{Patch}\ \mathsf{type}{<}\ \mathsf{typeLookup}\ - < -}\\
\mathsf{ilookup} & m & im & n & vs & \overset{\mathsf{lift}}{\Leftarrow}\ \mathsf{lift\text{-}case}\\
\quad\mathsf{ilookup} & m & im & 0 & \mathsf{nil} & \overset{\mathsf{lift}}{\mapsto}\ \mathsf{nothing} * [*]\\
\quad\mathsf{ilookup} & m & im & (\mathsf{suc}\ n) & (\mathsf{cons}\ a\ vs) & \overset{\mathsf{lift}}{\Leftarrow}\ \mathsf{lift\text{-}ind}\\
\qquad\mathsf{ilookup} & 0 & 0 & (\mathsf{suc}\ n) & (\mathsf{cons}\ a\ vs) & \overset{\mathsf{lift}}{\mapsto}\ \mathsf{just}\ \{?:a\}\ A[*]\\
\qquad\mathsf{ilookup} & (\mathsf{suc}\ m) & (\mathsf{suc}\ im) & (\mathsf{suc}\ n) & (\mathsf{cons}\ a\ vs) & \{?\}
\end{array}$$

As before, in an interactive setting, the user would instruct the machine to execute the command $\overset{\mathsf{lift}}{\mapsto}$ and the computer would come back with the skeleton of the expected inputs. Finishing the implementation of ilookup is now one baby step away:

$$\begin{array}{llllll}
\multicolumn{6}{l}{\mathsf{ilookup}\ :\ \mathsf{Patch}\ \mathsf{type}\!<\ \mathsf{typeLookup}\ \ - < -} \\
\mathsf{ilookup} & m & im & n & vs & \overset{\mathsf{lift}}{\Leftarrow}\ \mathsf{lift\text{-}case} \\
\quad\mathsf{ilookup} & m & im & 0 & \mathsf{nil} & \overset{\mathsf{lift}}{\mapsto}\ \mathsf{nothing}\ {*}[{*}] \\
\quad\mathsf{ilookup} & m & im & (\mathsf{suc}\ n) & (\mathsf{cons}\ a\ vs) & \overset{\mathsf{lift}}{\Leftarrow}\ \mathsf{lift\text{-}ind} \\
\qquad\mathsf{ilookup} & 0 & 0 & (\mathsf{suc}\ n) & (\mathsf{cons}\ a\ vs) & \overset{\mathsf{lift}}{\mapsto}\ \mathsf{just}\ a[{*}] \\
\qquad\mathsf{ilookup} & (\mathsf{suc}\ m) & (\mathsf{suc}\ im) & (\mathsf{suc}\ n) & (\mathsf{cons}\ a\ vs) & \mapsto\ \mathsf{ilookup}\ m\ im\ n\ vs
\end{array}$$

9.77 Example (Transporting the constructors of $- + -$)**.** We can also benefit from the automatic lifting of constructors to fill out the cons case of vector append. We instruct the system that we want to lift the suc constructor, which results in the following goals:

$$\begin{array}{llllll}
\multicolumn{6}{l}{\mathsf{vappend}\ :\ \mathsf{Patch}\ \mathsf{type}\!+\ \mathsf{type}\!+\!+\ \ - + -} \\
\mathsf{vappend} & m & xs & n & ys & \overset{\mathsf{lift}}{\Leftarrow}\ \mathsf{lift\text{-}case} \\
\ \mathsf{vappend} & 0 & \mathsf{nil} & n & ys & \{?\} \\
\ \mathsf{vappend} & (\mathsf{suc}\ m) & (\mathsf{cons}\ a\ xs) & n & ys & \overset{\mathsf{lift}}{\mapsto}\ \mathsf{cons}\ \{?:A\}\,[\,\{?\}\,]
\end{array}$$

It is then straightforward to, manually this time, conclude the implementation of vappend:

$$\begin{array}{llllll}
\multicolumn{6}{l}{\mathsf{vappend}\ :\ \mathsf{Patch}\ \mathsf{type}\!+\ \mathsf{type}\!+\!+\ \ - + -} \\
\mathsf{vappend} & m & xs & n & ys & \overset{\mathsf{lift}}{\Leftarrow}\ \mathsf{lift\text{-}case} \\
\ \mathsf{vappend} & 0 & \mathsf{nil} & n & ys & \mapsto\ ys \\
\ \mathsf{vappend} & (\mathsf{suc}\ m) & (\mathsf{cons}\ a\ xs) & n & ys & \overset{\mathsf{lift}}{\mapsto}\ \mathsf{cons}\ a[\mathsf{vappend}\ m\ xs\ n\ ys]
\end{array}$$

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 interesting 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.

(a) Request lifting of algebra (user input):

```
ihd   (vs : Vec A n)  :  IMaybe A isSuc n
ihd                  ⇐lift  lift-fold
{?}
```

(b) Result of lifting the algebra (system output):

```
ihd (vs : Vec A n)  :  IMaybe A (isSuc n)
ihd                ⇐lift  lift-fold    where
    ihdAlg (vs : ⟦VecD⟧ (λn′. IMaybe A (isSuc n′)) n)   :   IMaybe A (isSuc n)
    ihdAlg                    ′nil                     {?}
    ihdAlg                ( ′cons a xs)                {?}
```

(c) Request lifting of constructors (user input):

```
ihd (vs : Vec A n)  :  IMaybe A (isSuc n)
ihd                ⇐lift  lift-fold    where
    ihdAlg (vs : ⟦VecD⟧ (λn′. IMaybe A (isSuc n′)) n)  :  IMaybe A (isSuc n)
    ihdAlg                    ′nil                     ↦lift  {?}
    ihdAlg                ( ′cons a xs)                ↦lift  {?}
```

(d) Result of lifting constructors (system output)

```
ihd (vs : Vec A n)  :  IMaybe A (isSuc n)
ihd                ⇐lift  lift-fold    where
    ihdAlg (vs : ⟦Vec A⟧ (λn′. IMaybe A (isSuc n′)) n)  :  IMaybe A (isSuc n)
    ihdAlg                    ′nil                      ↦lift  nothing {?: 𝟙} [ {?: 𝟙} ]
    ihdAlg                ( ′cons a xs)                 ↦lift  just {?: A} [ {?: 𝟙} ]
```

(e) type checked term (automatically generated from (d)):

```
ihd (vs : Vec A n) : IMaybe A (isSuc n)
ihd vs ↦ lift-fold isSucAlg ihdAlg     where
    ihdAlg (vs : ⟦Vec A⟧ (λn′. IMaybe A (isSuc n′)) n) : IMaybe A (isSuc n)
    ihdAlg       ′nil       ↦ lift-constructor ′nil {?: 𝟙}  {?: 𝟙} ∗
    ihdAlg ( ′cons a xs) ↦ lift-constructor ( ′suc n) {?: A}  {?: 𝟙} ∗
```

Figure 9.3.: Guided implementation of ihd

Conclusion

(9.79) 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.

(9.80) 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

(9.81) 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.

(9.82) 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 Π-types, Σ-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 simply justified by type checking.

(10.5) 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

(10.6) 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 μ, we could introduce a greatest fixpoint ν. 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 SET and slices of 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.

(10.7) 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.

(10.8) 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 $- \mathbb{1}_{\mathbb{C}} : [\mathbb{C}, \mathbb{D}] \rightarrow \mathbb{D}$ is a fibration for $\mathbb{D}$ pullback-complete and $\mathbb{C}$ equipped with a terminal object $\mathbb{1}_{\mathbb{C}}$. Specialised to the slices of an LCCC, the Cartesian morphisms are exactly our ornaments. What about the general case?

(10.9) 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.

(10.10) 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 (Π-types and Σ-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 ilookup, we would mark the index A of Vec A and IMaybe A as parametric so that in the 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.

(10.11) 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

(10.12) 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.

(10.13) A first direction of development is the elaboration of inductive definitions. In Epigram2, 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.

(10.14) 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.

(10.15) 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

(10.16) 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.

(10.17) 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.* a programming language, that offers an interesting playground for categorical ideas.

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.

(10.18) 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*.

(10.19) 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

$\llbracket (D : \mathsf{Desc}) \rrbracket : (X : \textsc{Set}) \to \textsc{Set}$ $\llbracket (D : \mathsf{IDesc}\ I) \rrbracket : (I \to \textsc{Set}) \to \textsc{Set}$ $\llbracket (D : \mathsf{func}\ I\ J) \rrbracket : (X : I \to \textsc{Set}) \to J \to \textsc{Set}$	Interpretation
$\Box_{(D:\mathsf{Desc})} : (P : X \to \textsc{Set})(xs : \llbracket D \rrbracket\ X) \to \textsc{Set}$ $\Box_{(D:\mathsf{func}\ I\ I)} : (P : (i : I) \times \mu\ D\ i \to \textsc{Set})(xs : \llbracket D \rrbracket\ (\mu\ D)\ i) \to \textsc{Set}$	Canonical lifting
$\mathsf{D}\Box_{(D:\mathsf{IDesc}\ I)} : (xs : \llbracket D \rrbracket\ X) \to \mathsf{IDesc}\ ((i : I) \times X\ i)$ $\mathsf{D}\Box_{(D:\mathsf{func}\ I\ J)} : \mathsf{func}\ ((i : I) \times X\ i)\ ((j : J) \times \llbracket D \rrbracket\ X\ j)$	Canonical lifting
$\Box^{\to}_{(D:\mathsf{Desc})} : (p : (x : X) \to P\ x)(xs : \llbracket D \rrbracket\ X) \to \Box_D\ P\ xs$ $\Box^{\to}_{(D:\mathsf{IDesc} I)} : (p : (x : X\ i) \to P\ x)(xs : \llbracket D \rrbracket\ X) \to \Box_D\ P\ xs$	Lifting map
$\mu\ (D : \mathsf{Desc}) : \textsc{Set}$ $\mu\ (D : \mathsf{func}\ I\ I) : I \to \textsc{Set}$	Least fixpoint
$\llparenthesis (\alpha : \llbracket D \rrbracket\ T \to T) \rrparenthesis : (x : \mu\ D) \to T$ $\llparenthesis (\alpha : \llbracket D \rrbracket\ T \mathrel{\dot{\to}} T) \rrparenthesis : (x : \mu\ D\ i) \to T\ i$	Catamorphism

Bibliography

M. Abbott. *Categories of Containers*. PhD thesis, University of Leicester, 2003.

M. Abbott, T. Altenkirch, C. McBride, and N. Ghani. ∂ for data: Differentiating data structures. *Fundamenta Informaticae*, 65(1-2):1–28, 2005.

A. Abel, T. Coquand, and P. Dybjer. Verifying a semantic $\beta\eta$-conversion test for Martin-Löf type theory. In *Mathematics of Program Construction*, pages 29–56, 2008. doi:10.1007/978-3-540-70594-9_4.

A. Abel, T. Coquand, and M. Pagano. A modular type-checking algorithm for type theory with singleton types and proof irrelevance. In *Typed Lambda Calculi and Applications*, 2009. doi:10.2168/LMCS-7(2:4)2011.

P. Aczel. An introduction to inductive definitions. In *Handbook of Mathematical Logic*, volume 90, pages 739 – 782. 1977. doi:10.1016/S0049-237X(08)71120-0.

R. Adams. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16(2):219–246, 2006. doi:10.1017/S0956796805005770.

G. Adelson-Velskii and E. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk USSR*, 146(2):263–266, 1962.

T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, 2003. doi:10.1007/978-3-540-76786-2_4.

T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, 1999. doi:10.1007/3-540-48168-0_32.

T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *Programming Languages meets Program Verification*, 2007. doi:10.1145/1292597.1292608.

A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8 (1), 2012. doi:10.2168/LMCS-8(1:18)2012.

R. Atkey, P. Johann, and N. Ghani. Refining inductive types. *Logical Methods in Computer Science*, 8, 2012. doi:10.2168/LMCS-8(2:9)2012.

L. Augustsson. Cayenne - a language with dependent types. In *Advanced Functional Programming*, pages 240–267, 1998. doi:10.1007/10704973_6.

L. Augustsson and M. Carlsson. An exercise in dependent types: A well-typed interpreter. Unpublished, 1999. URL `http://www.cs.chalmers.se/~augustss/cayenne/interp.ps`.

S. Awodey. *Category Theory (Oxford Logic Guides)*. Oxford University Press, USA, 2006. ISBN 0199237182.

S. Awodey and M. A. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 146:45–55, 2009. doi:10.1017/s0305004108001783.

S. Awodey, N. Gambino, and K. Sojakova. Inductive types in homotopy type theory. In *Logic in Computer Science*, pages 95–104, 2012. doi:10.1109/LICS.2012.21.

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.

M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10, 2003.

N. Benton, C.-K. Hur, A. J. Kennedy, and C. Mcbride. Strongly typed term representations in coq. *Journal of Automated Reasoning*, 49(2):141–159, 2012. doi:10.1007/s10817-011-9219-0.

J.-P. Bernardy and M. Lasson. Realizability and parametricity in pure type systems. In *Foundations of Software Science and Computation Structures*, pages 108–122, 2011. doi:10.1007/978-3-642-19805-2_8.

R. Bird and L. Meertens. Nested datatypes. In *Mathematics of Program Construction*, volume 1422, pages 52–67. 1998. doi:10.1007/BFb0054285.

R. S. Bird and O. de Moor. *Algebra of programming*. Prentice Hall, 1997. ISBN 013507245X.

E. Brady, J. Chapman, P.-E. Dagand, A. Gundry, C. McBride, P. Morris, and U. Norell. An Epigram implementation. URL `http://www.e-pig.org/`.

E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs*, 2003. doi:10.1007/978-3-540-24849-1_8.

J. Chapman, T. Altenkirch, and C. McBride. Epigram reloaded: a standalone typechecker for ETT. In *Trends in Functional Programming*, volume 6, pages 79–94, 2005.

J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. In *International Conference on Functional Programming*, pages 3–14, 2010. doi:10.1145/1863543.1863547.

J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

R. L. Constable. *Implementing Mathematics With the Nuprl Proof Development System*. Prentice Hall, 1986. ISBN 0134518322.

C. Coquand and T. Coquand. Structured type theory. In *Logical Frameworks and Metalanguages*, 1999.

T. Coquand. Pattern matching with dependent types. In *Types for Proofs and Programs*, 1992.

T. Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26, 1996. doi:10.1016/0167-6423(95)00021-6.

T. Coquand. Equality and dependent type theory. Bologna lectures, 2011. URL `http://www.cse.chalmers.se/~coquand/bologna.pdf`.

T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, Feb. 1988. doi:10.1016/0890-5401(88)90005-3.

J. Courant. Explicit universes for the calculus of constructions. In *Theorem Proving in Higher Order Logics*, 2002. doi:10.1007/3-540-45685-6_9.

P.-L. Curien. Substitution up to isomorphism. *Fundamenta Informaticae*, 19(1-2):51–85, 1993.

P.-E. Dagand and C. McBride. Transporting functions across ornaments. In *International Conference on Functional Programming*, pages 103–114, 2012. doi:10.1145/2364527.2364544.

P.-E. Dagand and C. McBride. A categorical treatment of ornaments. In *Logics in Computer Science*, 2013a. doi:10.1109/LICS.2013.60.

P.-E. Dagand and C. McBride. Elaborating inductive definitions. In *Journées Francophones des Langages Applicatifs*, 2013b.

G. Deleuze and F. Guattari. *A Thousand Plateaus: Capitalism and Schizophrenia*. University of Minnesota Press, 1987. ISBN 0816614024.

D. Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, 2005.

P. Dybjer. Inductive sets and families in Martin-Löf's type theory. In *Logical Frameworks*, 1991.

P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994. doi:10.1007/BF01211308.

P. Dybjer. Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoretical Computer Science*, 176(1-2):329–335, 1997. doi:10.1016/S0304-3975(96)00145-4.

P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications*, 1999. doi:10.1007/3-540-48959-2_11.

P. Dybjer and A. Setzer. Indexed induction-recursion. In *Proof Theory in Computer Science*. 2001. doi:10.1016/j.jlap.2005.07.001.

M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Logic in Computer Science*, pages 193–202, 1999. doi:10.1109/LICS.1999.782615.

T. Freeman and F. Pfenning. Refinement types for ML. In *Programming Language Design and Implementation*, pages 268–277, 1991. doi:10.1145/113445.113468.

C. Fumex. *Induction and coinduction schemes in category theory*. PhD thesis, University of Strathclyde, 2012.

N. Gambino and M. Hyland. Wellfounded trees and dependent polynomial functors. In *Types for Proofs and Programs*, volume 3085, pages 210–225. 2004. doi:10.1007/978-3-540-24849-1_14.

N. Gambino and J. Kock. Polynomial functors and polynomial monads. *Mathematical Proceedings of the Cambridge Philosophical Society*, 154(1):153–192, 2013. doi:10.1017/S0305004112000394.

R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Object-Oriented Programming, Systems, Languages and Applications*, 2003. doi:10.1145/949305.949317.

R. Garner. On the strength of dependent products in the type theory of Martin-Löf. *Annals of Pure and Applied Logic*, 160(1):1–12, 2009. doi:10.1016/j.apal.2008.12.003.

H. Geuvers. Induction is not derivable in second order dependent type theory. In *Typed Lambda Calculi and Applications*, 2001. doi:10.1007/3-540-45413-6_16.

J. Gibbons. Datatype-generic programming. In *Spring School on Datatype-Generic Programming*, volume 4719, pages 1–71, 2007. doi:10.1007/978-3-540-76786-2.

E. Giménez. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs*, volume 996, pages 39–59. 1995. doi:10.1007/3-540-60579-7_3.

J.-Y. Girard. *Interprétation functionelle et Elimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, Université Paris VII, 1972.

H. Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994.

H. Goguen and Z. Luo. Inductive data types: well-ordering types revisited. In *Workshop on Logical Environments*, pages 198–218, 1993.

H. Goguen, C. McBride, and J. McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning and Computation*, volume 4060, pages 521–540. 2006. doi:10.1007/11780274_27.

T. Hallgren and A. Ranta. An extensible proof text editor. In *Logic for Programming and Automated Reasoning*, pages 70–84, 2000. doi:10.1007/3-540-44404-1_6.

R. Harper and R. Pollack. Type checking with universes. In *Theory and Practice of Software Development*, 1989. doi:10.1016/0304-3975(90)90108-T.

R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language, and Interaction: essays in honour of Robin Milner*, 2000.

M. Hedberg. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming*, 8:413–436, 1998. doi:10.1017/S0956796898003153.

C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Information and Computation*, 145(2):107–152, 1998. doi:10.1006/inco.1998.2725.

R. Hinze. Memo functions, polytypically! In *Workshop on Generic Programming*, pages 17–32, 2000a.

R. Hinze. Polytypic values possess polykinded types. In *Mathematics of Program Construction*, 2000b. doi:10.1016/S0167-6423(02)00025-4.

R. Hinze and S. P. Jones. Derivable type classes. *Electronic Notes in Theoretical Computer Science*, 41(1):5 – 35, 2001. doi:10.1016/S1571-0661(05)80542-0.

R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. In *Mathematics of Program Construction*, 2002. doi:10.1016/j.scico.2003.07.001.

R. Hinze, J. Jeuring, and A. Löh. Comparing approaches to generic programming in Haskell. In *Datatype-Generic Programming*, 2007. doi:10.1007/978-3-540-76786-2_2.

M. Hofmann. On the interpretation of type theory in locally cartesian closed categories. In *Computer Science Logic*, volume 933, pages 427–441, 1995. doi:10.1007/BFb0022273.

M. Hofmann and T. Streicher. The groupoid model refutes uniqueness of identity proofs. In *Logic in Computer Science*, pages 208–212, 1994. doi:10.1109/LICS.1994.316071.

S. Holdermans, J. Jeuring, A. Löh, and A. Rodriguez. Generic views on data types. In *Mathematics of Program Construction*. 2006. doi:10.1007/11783596_14.

G. Huet. The zipper. *Journal of Functional Programming*, 7(05):549–554, 1997. doi:10.1017/S0956796897002864.

B. Jacobs. *Categorical Logic and Type Theory*. Elsevier Science, 2001. doi:10.1016/S0049-237X(98)80028-1.

P. Jansson and J. Jeuring. PolyP—a polytypic programming language extension. In *Principles of Programming Languages*, 1997. doi:10.1145/263699.263763.

B. Jay and R. Cockett. Shapely types and shape polymorphism. In *European Symposium on Programming*, 1994. doi:10.1007/3-540-57880-3_20.

G. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. PhD thesis, Technische Universiteit Eindhoven, 2004.

H.-S. Ko and J. Gibbons. Modularising inductive families. In *Workshop on Generic Programming*, pages 13–24, 2011.

J. Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103 (2):151–161, 1968. doi:10.1007/BF01110627.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Types in Language Design and Implementation*, 2003. doi:10.1007/978-3-540-40018-9_23.

L. Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, 1995. doi:10.2307/2974556.

S. Leather, A. Löh, and J. Jeuring. Pull-ups, push-downs, and passing it around. In *Implementation and Application of Functional Languages*, volume 6041, pages 159–178. 2011. doi:10.1007/978-3-642-16478-1_10.

D. R. Licata and R. Harper. 2-dimensional directed type theory. *Electronic Notes Theoretical Computer Science*, 276(0):263–289, 2011. doi:10.1016/j.entcs.2011.09.026.

F. Lindblad and M. Benke. A tool for automated theorem proving in Agda. In *Types for Proofs and Programs*, pages 154–169, 2004. doi:10.1007/11617990_10.

Z. Luo. *Computation and Reasoning*. Oxford University Press, 1994. ISBN 978-0-19-853835-6.

S. Mac Lane. *Categories for the Working Mathematician*. Springer, 1998. ISBN 0387984038.

J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for haskell. In *Haskell Symposium*, pages 37–48, 2010. doi:10.1145/1863523.1863529.

L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, pages 213–237, 1993. doi:10.1007/3-540-58085-9_78.

P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis Napoli, 1984. ISBN 88-7088-105-9.

P. Martin-Löf. An intuitionistic theory of types: Predicative part. *The Journal of Symbolic Logic*, 49, 1984. doi:10.2307/2274116.

P. Martin-Löf. Constructive mathematics and computer programming. In *Mathematical logic and programming languages*, pages 167–184, 1985. doi:10.1098/rsta.1984.0073.

P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1:11–60, 1996.

R. Matthes. An induction principle for nested datatypes in intensional type theory. *Journal of Functional Programming*, 19:439–468, 2009. doi:10.1017/S095679680900731X.

C. McBride. Kleisli arrows of outrageous fortune. Unpublished. URL `http://personal.cis.strath.ac.uk/~conor/Kleisli.pdf`.

C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, LFCS, 1999.

C. McBride. Elimination with a motive. In *Types for Proofs and Programs*, page 727, 2002. doi:10.1007/3-540-45842-5_13.

C. McBride. Let's see how things unfold: reconciling the infinite with the intensional. In *Conference on Algebra and Coalgebra in Computer Science*, pages 113–126, 2009. doi:10.1007/978-3-642-03741-2_9.

C. McBride. Ornamental algebras, algebraic ornaments. *Journal of Functional Programming*, 2013. To appear.

C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004. doi:10.1007/11780274_27.

C. McBride, H. Goguen, and J. McKinna. A few constructions on constructors. In *Types for Proofs and Programs*, pages 186–200, 2004. doi:10.1007/11617990_12.

R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.

R. Milner, M. Tofte, and D. MacQueen. *The Definition of Standard ML*. MIT Press, 1997. ISBN 0262631814.

P. Morris. *Constructing Universes for Generic Programming*. PhD thesis, University of Nottingham, 2007.

P. Morris and T. Altenkirch. Indexed containers. In *Logics in Computer Science*, 2009. doi:10.1109/LICS.2009.33.

P. Morris, T. Altenkirch, and C. McBride. Exploring the regular tree types. In *Types for Proofs and Programs*, 2004. doi:10.1007/11617990_16.

P. Morris, T. Altenkirch, and N. Ghani. A universe of strictly positive families. *International Journal of Foundations of Computer Science*, 20(1):83–107, 2009. doi:10.1142/S0129054109006462.

B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press, 1990. ISBN 0198538146.

U. Norell. Functional generic programming and type theory. Master's thesis, Chalmers University of Technology, 2002.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1998. ISBN 978-0521663502.

N. Oury. *Égalités et filtrages avec types dépendants dans le Calcul des Constructions Inductives*. PhD thesis, Université Paris-Sud, 2006.

N. Oury and W. Swierstra. The power of Pi. In *International Conference on Functional Programming*, 2008. doi:10.1145/1411204.1411213.

E. Palmgren. On universes in type theory. In *Twenty-Five Years of Constructive Type Theory*, pages 191–204, 1995.

C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris VII, 1989.

C. Paulin-Mohring. *Définitions inductives en théorie des types d'ordre supérieur*. Habilitation à diriger les recherches, ENS Lyon, 1996.

K. Petersson and D. Synek. A set constructor for inductive sets in Martin-Löf's type theory. In *Category Theory and Computer Science*, pages 128–140, 1989. doi:10.1007/BFb0018349.

S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*, volume 13 of *Journal of Functional Programming*. 2003. doi:10.1017/S0956796803000315.

H. Pfeifer and H. Ruess. Polytypic abstraction in type theory. In *Workshop on Generic Programming*, 1998.

H. Pfeifer and H. Ruess. Polytypic proof construction. In *Conference on Theorem Proving in Higher Order Logics*, pages 55–72, 1999. doi:10.1007/3-540-48256-3_5.

B. C. Pierce and D. N. Turner. Local type inference. *Transactions on Programming Languages and Systems*, 22:1–44, 2000. doi:10.1145/345099.345100.

A. Pitts. Polymorphism is set theoretic, constructively. In *Category Theory and Computer Science*, volume 283, pages 12–39, 1987. doi:10.1007/3-540-18508-9_18.

G. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003. doi:10.1023/A:1023064908962.

M. Puech. Proofs, upside down: a functional correspondence between natural deduction and the sequent calculus. In *Asian Symposium on Programming Languages and Systems*, 2013. In press.

J. Reynolds. Polymorphism is not set-theoretic. In *Semantics of Data Types*, volume 173, pages 145–156, 1984. doi:10.1007/3-540-13346-1_7.

A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell Symposium*, 2008. doi:10.1145/1411286.1411301.

T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *International Conference on Functional Programming*, pages 341–352, 2009. doi:10.1145/1596550.1596599.

R. A. G. Seely. Locally cartesian closed categories and type theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 95, 1983. doi:10.1017/S0305004100061284.

M. Shulman. Framed bicategories and monoidal fibrations. *Theory and Applications of Categories*, 20(18):650–738, 2008.

V. Siles. *Investigation on the typing of equality in type systems*. PhD thesis, École Polytechnique, 2010.

M. Smyth and G. Plotkin. The category-theoretic solution of recursive domain equations. In *Foundations of Computer Science*, pages 13–17, 1977. doi:10.1109/SFCS.1977.30.

M. Sozeau. Equations: A dependent pattern-matching compiler. In *Interactive Theorem Proving*, pages 419–434, 2010. doi:10.1007/978-3-642-14052-5_29.

G. L. Steele. Growing a language. *Higher Order Symbolic Computation*, 12(3):221–236, 1999. doi:10.1023/A:1010085415024.

P.-Y. Strub. Coq modulo theory. In *Computer Science Logic*, pages 529–543, 2010. doi:10.1007/978-3-642-15205-4_40.

N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *International Conference on Functional Programming*, pages 266–278, 2011. doi:10.1145/2034773.2034811.

W. Swierstra. *A Functional Specification of Effects*. PhD thesis, University of Nottingham, 2008.

W. Tait. Intensional interpretations of functionals of finite type. *The Journal of Symbolic Logic*, 32(2), 1967. doi:10.2307/2271658.

The Coq Development Team. *The Coq Proof Assistant Reference Manual*. URL `http://coq.inria.fr/refman/`.

W. Verbruggen, E. de Vries, and A. Hughes. Polytypic programming in Coq. In *Workshop on Generic Programming*, 2008. doi:10.1145/1411318.1411326.

W. Verbruggen, E. de Vries, and A. Hughes. Formal polytypic programs and proofs. *Journal of Functional Programming*, 20(3-4):213–269, 2010. doi:10.1017/S0956796810000158.

K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. In *Types for Proofs and Programs*, pages 355–377, 2004. doi:10.1007/978-3-540-24849-1_23.

S. Weirich and C. Casinghino. Arity-generic datatype-generic programming. In *Programming Languages meets Program Verification*, 2010. doi:10.1145/1707790.1707799.

B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris-Diderot - Paris VII, 1994.

A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, 1910. doi:10.1093/mind/LVII.226.137.

E. P. Wigner. The unreasonable effectiveness of mathematics in the natural sciences. *Communications on Pure and Applied Mathematics*, 13(1):1–14, 1960. doi:10.1002/cpa.3160130102.

G. C. Wraith. Algebraic theories. *Aarhus Universitet. Lecture Note Series*, 22, 1975.

A. R. Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *International Conference in Functional Programming*, 2009. doi:10.1145/1596550.1596585.

Index

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 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, writing 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 *finally* 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 Boutillier 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 Silès.

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.