Chapter 2

From intuitionistic type theory to dependently typed programming

We start with an introduction to intuitionistic type theory [Martin-Löf, 1984] and dependently typed programming [Altenkirch et al., 2005; McBride, 2004] using the Agda language [Norell, 2007, 2009; Bove and Dybjer, 2009]. Intuitionistic type theory was developed by Martin-Löf to serve as a foundation of intuitionistic mathematics like Bishop's renowned work on constructive analysis [Bishop and Bridges, 1985]. While originated from intuitionistic type theory, dependently typed programming is more concerned with mechanisation and practicalities, and is influenced by the program construction movement. It has thus departed from the mathematical traditions considerably, and deviations can be found from syntactic presentations to the underlying philosophy.

2.1 Datatypes and universe construction

Central to *datatype-generic programming* is the idea that the definitional structure of datatypes can be coded as first-class entities and thus become ordinary parameters to programs. The same idea is also found in Martin-Löf's Type Theory [Martin-Löf, 1984], in which a set of codes for datatypes is called a *uni-*

verse (à la Tarski), and there is a decoding function translating codes to actual types. Type theory being the foundation of dependently typed languages, universe construction can be done directly in such languages, so datatype-generic programming becomes just ordinary programming in the dependently typed world [Altenkirch and McBride, 2003]. In this section we construct a universe of *index-first datatypes* [Chapman et al., 2010; Dagand and McBride, 2012], on which a second universe of *ornaments*, to be constructed in **??**, will depend.

present codes along with their interpretation; not induction-recursion [Dybjer, 1998] though

2.1.1 High-level introduction to index-first datatypes

In Agda, an inductive family is declared by listing all possible constructors and their types, all ending with one of the types in that inductive family. This conveys the idea that the index in the type of an inhabitant is synthesised in a *bottom-up* fashion following the construction of the inhabitant. Consider vectors, for example: the cons constructor takes a vector at some index n and constructs a vector at suc n — the final index is computed bottom-up from the index of the sub-vector. This approach can yield redundant representation, though — the cons constructor for vectors has to store the index of the sub-vector, so the representation of a vector would be cluttered with all the intermediate lengths. If we switch to the opposite perspective, determining *top-down* from the targeted index what constructors should be supplied, then the representation can usually be significantly cleaned up — for a vector, if the index of its type is known to be suc n for some n, then we know that its top-level constructor can only be cons and the index of the sub-vector must be n. To reflect this important reversal of logical order, Dagand and McBride [2012] proposed a new notation for index-first datatype declarations, in which we first list all possible patterns of (the indices of) the types in the inductive family, and then specify for each pattern which constructors it offers. Below we follow Ko and Gibbons's slightly more Agda-like adaptation of the notation [2013].

Index-first declarations of simple datatypes look almost like Haskell data declarations. For example, natural numbers are declared by

```
indexfirst data Nat : Set where
  Nat ∋ zero
      | suc (n : Nat)
```

We use the keyword **indexfirst** to explicitly mark the declaration as an index-first one. The only possible pattern of the datatype is Nat, which offers two constructors zero and suc, the latter taking a recursive argument named n. We declare lists similarly, this time with a uniform parameter A : Set:

```
indexfirst data List (A : Set) : Set where
  List A ∋ []
         | _::_ (a : A) (as : List A)
```

The declaration of vectors is more interesting, fully exploiting the power of index-first datatypes:

```
indexfirst data Vec (A : Set) : Nat → Set where
  Vec A zero    ∋ []
  Vec A (suc n) ∋ _::_ (a : A) (as : Vec A n)
```

Vec A is a family of types indexed by Nat, and we do pattern matching on the index, splitting the datatype into two cases Vec A zero and Vec A (suc n) for some n : Nat. The first case only offers the nil constructor $[]$, and the second case only offers the cons constructor $\_::\_$. Because the form of the index restricts constructor choice, the recursive structure of a vector as : Vec A n must follow that of n, i.e., the number of cons nodes in as must match the number of successor nodes in n. We can also declare the bottom-up vector datatype in index-first style:

```
indexfirst data Vec (A : Set) : Nat → Set where
  Vec A n ∋ nil (neq : n ≡ zero)
          | cons (a : A) {m : Nat}
                 (as : Vec A m) (meq : n ≡ suc m)
```

Besides the field m storing the length of the tail, two more fields neq and meq are inserted, demanding explicit equality proofs about the indices. When a

vector of type Vec A n is demanded, we are "free" to choose between nil or cons regardless of the index n; however, because of the equality constraints, we are indirectly forced into a particular choice.

Remark (*detagging*)**.** The transformation from bottom-up vectors to top-down vectors is exactly what Brady et al.'s *detagging* optimisation [2004] does. With index-first datatypes, however, detagged representations are available directly, rather than arising from a compiler optimisation. (*End of remark.*)

Remark (*bidirectional typechecking*)**.**

TBC

(*End of remark.*)

2.1.2 Universe construction

Now we proceed to construct a universe for index-first datatypes. An inductive family of type $I \to \mathsf{Set}$ is constructed by taking the least fixed point of a base endofunctor on $I \to \mathsf{Set}$. For example, to get index-first vectors, we would define a base functor (parametrised by $A : \mathsf{Set}$)

$$\begin{aligned} &\mathit{VecF}\ A : (\mathsf{Nat} \to \mathsf{Set}) \to (\mathsf{Nat} \to \mathsf{Set}) \\ &\mathit{VecF}\ A\ X\ \mathsf{zero} \quad = \top \\ &\mathit{VecF}\ A\ X\ (\mathsf{suc}\ n) = A \times X\ n \end{aligned}$$

and take its least fixed point. If we flip the order of arguments of $\mathit{VecF}\ A$:

$$\begin{aligned} &\mathit{VecF}'\ A : \mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Set}) \to \mathsf{Set} \\ &\mathit{VecF}'\ A\ \mathsf{zero} \quad = \lambda\ X \to \top \\ &\mathit{VecF}'\ A\ (\mathsf{suc}\ n) = \lambda\ X \to A \times X\ n \end{aligned}$$

we see that $\mathit{VecF}'\ A$ consists of two different "responses" to the index request, each of type $(\mathsf{Nat} \to \mathsf{Set}) \to \mathsf{Set}$. It suffices to construct for such responses a universe

$$\textbf{data}\ \mathsf{RDesc}\ (I : \mathsf{Set}) : \mathsf{Set}_1$$

with a decoding function specifying its semantics:

$$[\![\_]\!] \;:\; \{I \;:\; \mathsf{Set}\} \to \mathsf{RDesc}\; I \to (I \to \mathsf{Set}) \to \mathsf{Set}$$

Inhabitants of $\mathsf{RDesc}\; I$ will be called *response descriptions*. A function of type $I \to \mathsf{RDesc}\; I$, then, can be decoded to an endofunctor on $I \to \mathsf{Set}$, so the type $I \to \mathsf{RDesc}\; I$ acts as a universe for index-first datatypes. We hence define

$$\begin{aligned}&\mathsf{Desc} \;:\; \mathsf{Set} \to \mathsf{Set}_1\\ &\mathsf{Desc}\; I \;=\; I \to \mathsf{RDesc}\; I\end{aligned}$$

with decoding function

$$\begin{aligned}&\mathbb{F} \;:\; \{I \;:\; \mathsf{Set}\} \to \mathsf{Desc}\; I \to (I \to \mathsf{Set}) \to (I \to \mathsf{Set})\\ &\mathbb{F}\; D\; X\; i \;=\; [\![\; D\; i \;]\!]\; X\end{aligned}$$

Inhabitants of type $\mathsf{Desc}\; I$ will be called *datatype descriptions*, or *descriptions* for short. Actual datatypes are manufactured from descriptions by the least fixed point operator:

$$\begin{aligned}&\mathbf{data}\; \mu\; \{I \;:\; \mathsf{Set}\}\; (D \;:\; \mathsf{Desc}\; I) \;:\; I \to \mathsf{Set}\; \mathbf{where}\\ &\quad \mathsf{con} \;:\; \mathbb{F}\; D\; (\mu\; D) \rightrightarrows \mu\; D\end{aligned}$$

We now define the datatype of response descriptions — which determines the syntax available for defining base functors — and its decoding function:

$$\begin{aligned}&\mathbf{data}\; \mathsf{RDesc}\; (I \;:\; \mathsf{Set}) \;:\; \mathsf{Set}_1\; \mathbf{where}\\ &\quad \mathsf{v} \;:\; (\mathit{is} \;:\; \mathsf{List}\; I) \to \mathsf{RDesc}\; I\\ &\quad \sigma \;:\; (S \;:\; \mathsf{Set})\; (D \;:\; S \to \mathsf{RDesc}\; I) \to \mathsf{RDesc}\; I\\ &[\![\_]\!] \;:\; \{I \;:\; \mathsf{Set}\} \to \mathsf{RDesc}\; I \to (I \to \mathsf{Set}) \to \mathsf{Set}\\ &[\![\; \mathsf{v}\; \mathit{is} \quad]\!]\; X \;=\; \mathbb{P}\; \mathit{is}\; X \quad \text{-- see below}\\ &[\![\; \sigma\; S\; D \;]\!]\; X \;=\; \Sigma\langle\, s : S \,\rangle\; [\![\; D\; s \;]\!]\; X\end{aligned}$$

The operator $\mathbb{P}$ computes the product of a finite number of types in a type family, whose indices are given in a list:

$$\begin{aligned}&\mathbb{P} \;:\; \{I \;:\; \mathsf{Set}\} \to \mathsf{List}\; I \to (I \to \mathsf{Set}) \to \mathsf{Set}\\ &\mathbb{P}\; [\,] \qquad X \;=\; \top\\ &\mathbb{P}\; (i :: \mathit{is})\; X \;=\; X\; i \times \mathbb{P}\; \mathit{is}\; X\end{aligned}$$

Thus, in a response, given $X \;:\; I \to \mathsf{Set}$, we are allowed to form dependent sums (by σ) and the product of a finite number of types in X (via v, suggesting

variable positions in the base functor).

Convention. We will informally refer to the index part of a σ as a *field*. Like Σ, we regard σ as a binder and write $\sigma\langle s : S \rangle\ D\ s$ for $\sigma\ S\ (\lambda s \mapsto D\ s)$. (*End of convention.*)

Example (*natural numbers*)**.** The datatype of natural numbers is considered to be an inductive family trivially indexed by ⊤, so the declaration of Nat corresponds to an inhabitant of Desc ⊤.

```
data ListTag : Set where ‘nil ‘cons : ListTag

NatD : Desc ⊤
NatD ▪ = σ ListTag λ { ‘nil  ↦ v []
                     ; ‘cons ↦ v (▪ :: []) }
```

The index request is necessarily ▪, and we respond with a field of type ListTag representing the constructor choices. If the field receives ‘nil, then we are constructing zero, which takes no recursive values, so we write v [] to end this branch; if the ListTag field receives ‘cons, then we are constructing a successor, which takes a recursive value at index ▪, so we write v (▪ :: []). (*End of example.*)

Example (*lists*)**.** The datatype of lists is parametrised by the element type. We represent parametrised descriptions simply as functions producing descriptions, so the declaration of lists corresponds to a function taking element types to descriptions.

```
ListD : Set → Desc ⊤
ListD A ▪ = σ ListTag λ { ‘nil  ↦ v []
                        ; ‘cons ↦ σ⟨ _ : A ⟩ v (▪ :: []) }
```

ListD A is the same as *NatD* except that, in the ‘cons case, we use σ to insert a field of type *A* for storing an element. (*End of example.*)

Example (*vectors*)**.** The datatype of vectors is parametrised by the element type and (non-trivially) indexed by Nat, so the declaration of vectors corresponds to

```
VecD : Set → Desc Nat
```

$$\begin{aligned}
&\mathit{VecD}\ A\ \mathsf{zero} &&= \mathsf{v}\ [\,]\\
&\mathit{VecD}\ A\ (\mathsf{suc}\ n) &&= \sigma\langle\ _ : A\ \rangle\ \mathsf{v}\ (n :: [\,])
\end{aligned}$$

which is directly comparable to the index-first base functor VecF' at the beginning of Section 2.1.2. (*End of example.*)

There is no problem defining functions on the encoded datatypes except that it has to be done with the raw representation. For example, list append is defined by

$$\begin{aligned}
&_{+\!\!+}_ : \mu\ (\mathit{ListD}\ A)\ \blacksquare \to \mu\ (\mathit{ListD}\ A)\ \blacksquare \to \mu\ (\mathit{ListD}\ A)\ \blacksquare\\
&\mathsf{con}\ (\text{`}\mathsf{nil}\quad ,\qquad\quad \blacksquare) {+\!\!+}\ \mathit{bs} = \mathit{bs}\\
&\mathsf{con}\ (\text{`}\mathsf{cons}\ ,\ a\ ,\ \mathit{as}\ ,\ \blacksquare) {+\!\!+}\ \mathit{bs} = \mathsf{con}\ (\text{`}\mathsf{cons}\ ,\ a\ ,\ \mathit{as} {+\!\!+}\ \mathit{bs}\ ,\ \blacksquare)
\end{aligned}$$

To improve readability, we define the following higher-level terms:

$$\begin{aligned}
&\mathsf{List} : \mathsf{Set} \to \mathsf{Set}\\
&\mathsf{List}\ A = \mu\ (\mathit{ListD}\ A)\ \blacksquare\\
&[\,] : \{A : \mathsf{Set}\} \to \mathsf{List}\ A\\
&[\,] = \mathsf{con}\ (\text{`}\mathsf{nil}\ ,\ \blacksquare)\\
&_::_ : \{A : \mathsf{Set}\} \to A \to \mathsf{List}\ A \to \mathsf{List}\ A\\
&a :: \mathit{as} = \mathsf{con}\ (\text{`}\mathsf{cons}\ ,\ a\ ,\ \mathit{as}\ ,\ \blacksquare)
\end{aligned}$$

List append can then be rewritten in the usual form (assuming that the terms $[\,]$ and $\_::\_$ can be used in pattern matching):

$$\begin{aligned}
&_{+\!\!+}_ : \mathsf{List}\ A \to \mathsf{List}\ A \to \mathsf{List}\ A\\
&[\,] \qquad {+\!\!+}\ \mathit{bs} = \mathit{bs}\\
&(a :: \mathit{as}) {+\!\!+}\ \mathit{bs} = a :: (\mathit{as} {+\!\!+}\ \mathit{bs})
\end{aligned}$$

Later on, when an encoded datatype is defined, we almost always supply a corresponding index-first datatype declaration immediately afterwards, which is thought of as giving definitions of higher-level terms for type and data constructors — the terms List, $[\,]$, and $\_::\_$ above, for example, can be considered to be defined by the index-first declaration of lists given in Section 2.1.1. Index-first declarations will only be regarded in this thesis as informal hints at how encoded datatypes are presented at a higher level; we do not give a formal treatment of the elaboration process from index-first declarations to

mutual

$$
\begin{aligned}
&\mathit{fold} : \{I : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\} \to \\
&\qquad \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \rightrightarrows X) \to (\mu\ D \rightrightarrows X) \\
&\mathit{fold}\ \{I\}\ \{D\}\ f\ \{i\}\ (\mathsf{con}\ \mathit{ds}) \ = \ f\ (\mathit{mapFold}\ D\ (D\ i)\ f\ \mathit{ds}) \\
&\mathit{mapFold} : \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I)\ (D' : \mathsf{RDesc}\ I) \to \\
&\qquad \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \rightrightarrows X) \to [\![\ D'\ ]\!]\ (\mu\ D) \to [\![\ D'\ ]\!]\ X \\
&\mathit{mapFold}\ D\ (\mathsf{v}\ [\,])\quad f\ \blacksquare \quad = \ \blacksquare \\
&\mathit{mapFold}\ D\ (\mathsf{v}\ (i :: \mathit{is}))\ f\ (d\ ,\ \mathit{ds}) \ = \ \mathit{fold}\ f\ d\ ,\ \mathit{mapFold}\ D\ (\mathsf{v}\ \mathit{is})\ f\ \mathit{ds} \\
&\mathit{mapFold}\ D\ (\sigma\ S\ D')\quad f\ (s\ ,\ \mathit{ds}) \ = \ s\ ,\ \mathit{mapFold}\ D\ (D'\ s)\ f\ \mathit{ds}
\end{aligned}
$$

Figure 2.1 Definition of the datatype-generic *fold* operator.

corresponding descriptions and definitions of higher-level terms. (One such treatment was given by Dagand and McBride [2013].)

Direct function definitions by pattern matching work fine for individual datatypes, but when we need to define operations and to state properties for all the datatypes encoded by the universe, it is necessary to have a generic *fold* operator parametrised by descriptions:

$$
\begin{aligned}
&\mathit{fold} : \{I : \mathsf{Set}\}\ \{D : \mathsf{Desc}\ I\} \to \\
&\qquad \{X : I \to \mathsf{Set}\} \to (\mathbb{F}\ D\ X \rightrightarrows X) \to (\mu\ D \rightrightarrows X)
\end{aligned}
$$

There is also a generic *induction* operator, which can be used to prove generic propositions about all encoded datatypes and subsumes *fold*, but *fold* is much easier to use when the full power of *induction* is not required. The implementations of both operators are adapted for our two-level universe from those in McBride's original work [2011]. We look at the implementation of the *fold* operator only, which is shown in Figure 2.1. As McBride, we would have wished to define *fold* by

$$
\mathit{fold}\ \{I\}\ \{D\}\ f\ \{i\}\ (\mathsf{con}\ \mathit{ds}) \ = \ f\ (\mathit{mapRD}\ (D\ i)\ (\mathit{fold}\ f)\ \mathit{ds})
$$

where the functorial mapping *mapRD* on response structures is defined by

$$
\mathit{mapRD} : \{I : \mathsf{Set}\}\ (D : \mathsf{RDesc}\ I) \to
$$

$$\{X\ Y : I \to \mathsf{Set}\}\ (g : X \rightrightarrows Y) \to [\![\ D\ ]\!]\ X \to [\![\ D\ ]\!]\ Y$$
$$\begin{array}{lll} \mathit{mapRD}\ (\mathsf{v}\ [\,])\ \ \ \ \ \ & g\ \blacksquare & = \blacksquare \\ \mathit{mapRD}\ (\mathsf{v}\ (i :: \mathit{is}))\ g\ (x\ ,\ \mathit{xs}) & & = g\ x\ ,\ \mathit{mapRD}\ (\mathsf{v}\ \mathit{is})\ g\ \mathit{xs} \\ \mathit{mapRD}\ (\sigma\ S\ D) & g\ (s\ ,\ \mathit{xs}) & = s\ ,\ \mathit{mapRD}\ (D\ s)\ g\ \mathit{xs} \end{array}$$

Agda does not see that this definition of *fold* is terminating, however, since the termination checker does not expand the definition of *mapRD* to see that *fold f* is applied to structurally smaller arguments. To make termination obvious, we instead define *fold* mutually recursively with *mapFold*, which is *mapRD* specialised by fixing its argument g to *fold f*.

It is helpful to form a two-dimensional image of our datatype manufacturing scheme: we manufacture a datatype by first defining a base functor, and then recursively duplicating the functorial structure by taking its least fixed point. The shape of the base functor can be imagined to stretch horizontally, whereas the recursive structure generated by the least fixed point grows vertically. This image works directly when the recursive structure is linear, like lists. (Otherwise one resorts to the abstraction of functor composition.) For example, we can typeset a list two-dimensionally like

```
con (‘cons , a ,
con (‘cons , b ,
con (‘nil  ,
  ▪) , ▪) , ▪)
```

Ignoring the last line of trailing ▪'s, things following con on each line — including constructor tags and list elements — are shaped by the base functor of lists, whereas the con nodes, aligned vertically, are generated by the least fixed point. This two-dimensional metaphor will be referred to in later explanations.

Remark (*first-order vs higher-order representation*)**.** The functorial structures generated by descriptions are strongly reminiscent of *indexed containers* [Altenkirch and Morris, 2009]; this will be explored and exploited in **??**. For now, it is enough to mention that we choose to stick to a first-order datatype manufacturing scheme, i.e., the datatypes we manufacture with descriptions use finite product types rather than dependent function types for branching, but it is

easy to switch to a higher-order representation that is even closer to indexed containers (allowing infinite branching) by storing in v a collection of I-indices indexed by an arbitrary set S:

$$\mathsf{v} : (S : \mathsf{Set})\ (f : S \to I) \to \mathsf{RDesc}\ I$$

whose semantics is defined in terms of dependent functions:

$$[\![\ \mathsf{v}\ S\ f\ ]\!]\ X\ =\ (s : S) \to X\ (f\ s)$$

The reason for choosing to stick to first-order representation is simply to obtain a simpler equality for the manufactured datatypes (Agda's default equality would suffice); the examples of manufactured datatypes in this thesis are all finitely branching and do not require the power of higher-order representation anyway. This choice, however, does complicate some subsequent datatype-generic definitions (e.g., ornaments). It would probably be helpful to think of the parts involving v and $\mathbb{P}$ in these definitions as specialisations of higher-order representations to first-order ones. (*End of remark.*)

2.2 Internalism vs externalism

Bibliography

Thorsten Altenkirch and Conor McBride [2003]. Generic programming within dependently typed programming. In *IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20. Kluwer, B.V. ↰ page 2

Thorsten Altenkirch, Conor McBride, and James McKinna [2005]. Why dependent types matter. Available at `http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf`. ↰ page 1

Thorsten Altenkirch and Peter Morris [2009]. Indexed containers. In *Logic in Computer Science*, LICS'09, pages 277–285. IEEE. ↰ page 9

Errett Bishop and Douglas Bridges [1985]. *Constructive Analysis*. Springer-Verlag. ↰ page 1

Ana Bove and Peter Dybjer [2009]. Dependent types at work. In *Language Engineering and Rigorous Software Development*, volume 5520 of *Lecture Notes in Computer Science*, pages 57–99. Springer-Verlag. ↰ page 1

Edwin Brady, Conor McBride, and James McKinna [2004]. Inductive families need not store their indices. In *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer-Verlag. ↰ page 4

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris [2010]. The gentle art of levitation. In *International Conference on Functional Programming*, ICFP'10, pages 3–14. ACM. doi:10.1145/1863543.1863547. ↰ page 2

Pierre-Évariste Dagand and Conor McBride [2012]. Transporting functions across ornaments. In *International Conference on Functional Programming*, ICFP'12, pages 103–114. ACM. doi:10.1145/2364527.2364544. ↰ page 2

Pierre-Évariste Dagand and Conor McBride [2013]. Elaborating inductive definitions. In *Journées Francophones des Langages Applicatifs*, JFLA'13. INRIA. ↰ page 8

Peter Dybjer [1998]. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549. ↰ pages 2 and 13

Hsiang-Shang Ko and Jeremy Gibbons [2013]. Modularising inductive families. *Progress in Informatics*, 10:65–88. doi:10.2201/NiiPi.2013.10.5. ↰ page 2

Per Martin-Löf [1984]. *Intuitionistic Type Theory*. Bibliopolis, Napoli. ↰ page 1

Conor McBride [2004]. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. ↰ page 1

Conor McBride [2011]. Ornamental algebras, algebraic ornaments. To appear in *Journal of Functional Programming*. ↰ page 8

Ulf Norell [2007]. *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology. ↰ page 1

Ulf Norell [2009]. Dependently typed programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer-Verlag. ↰ page 1

Todo list