Chapter 2

From intuitionistic type theory to dependently typed programming

This preliminary chapter serves three purposes:

- The general theme of the dissertation is set up by describing the propositions-as-types-principle (Section 2.1) and its influence on program construction (Section 2.5).
- It is explained that, while we adopt AGDA as the expository language, we make sure that every AGDA program in this dissertation can at least in principle be translated down to well-studied aspects of type theory — no mysterious AGDA-specific feature is used. Specifically, we briefly discuss foundational aspects of pattern matching (Section 2.2) and equality (Section 2.3).
- Most AGDA-specific syntax, notational conventions, and basic constructions used throughout the dissertation are also introduced. In particular, a universe for "index-first" inductive families is constructed (Section 2.4), which is the basis of essentially all later constructions.

2.1 Propositions as types

Mathematics is all about mental constructions, that is, the intuitive grasp and manipulation of mental objects, the intuitionists say [Heyting, 1971; Dummett, 2000]. Take the natural numbers as an example. We have a distinct idea of how natural numbers are built: start from an origin 0, and form its successor 1, and then the successor of 1, which is 2, and so on. In other words, it is in our nature to be able to count, and counting is just the way the natural numbers are constructed. This construction then gives a specification of when we can immediately (i.e., directly intuitively) recognise a natural number, namely when it is 0 or a successor of some other natural number, and this specification of immediately recognisable forms is one of the conditions of forming the **set** of natural numbers in Martin-Löf Type Theory. In symbols, we are justified by our intuition to have the **formation rule**

$$\frac{}{\mathsf{Nat} \ : \ \mathsf{Set}}$$

saying that we can conclude (below the line) that Nat is a set from no assumptions (above the line), and the two **introduction rules**

$$\frac{}{\mathsf{zero} \ : \ \mathsf{Nat}} \qquad \frac{n \ : \ \mathsf{Nat}}{\mathsf{suc}\ n \ : \ \mathsf{Nat}}$$

specifying the **canonical inhabitants** of Nat, i.e., those inhabitants that are immediately recognisable as belonging to Nat, namely zero and suc n whenever n is an inhabitant of Nat. There are natural numbers which are not in canonical form (like 10^{10}) but instead encode an effective method for computing a canonical inhabitant. We accept them as **non-canonical inhabitants** of Nat, as long as they compute to a canonical form so we can see that they are indeed natural numbers. Thus, to form a set, we should be able to recognise its inhabitants, either directly or indirectly, as bearing a certain form and thus belonging to the set, so the inhabitants of the set are intuitively clear to us as a certain kind of mental construction.

What is more characteristic of intuitionism is that the intuitionistic interpretation of propositions — in particular the logical constants/connectives — follows the same line of thought as the specification of the set of natural numbers. A proposition is an expression of its truth condition, and since intuitionistic truth follows from proofs, a proposition is clearly specified exactly when what constitutes a proof of it is determined [Martin-Löf, 1987]. What is a proof of a proposition, then? It is a piece of mental construction such that, upon inspection, the truth of the proposition is immediately recognised. For a simple example, in type theory we can formulate the formation rule for conjunctions

$$\frac{A \,:\, \mathsf{Set} \qquad B \,:\, \mathsf{Set}}{A \wedge B \,:\, \mathsf{Set}}$$

and the introduction rule

$$\frac{a \,:\, A \qquad b \,:\, B}{(a\,,b) \,:\, A \wedge B}$$

saying that an immediately acceptable proof (canonical inhabitant) of $A \wedge B$ is a pair consisting of a proof (inhabitant) of A and a proof (inhabitant) of B. Any other (non-canonical) way of proving a conjunction must effectively yield a proof in the form of a pair. The relationship between a proposition and its proofs is thus exactly the same as the one between a set and its inhabitants — the proofs must be effectively recognisable as proving the proposition. Hence, in type theory, the notion of propositions and proofs is subsumed by the notion of sets and inhabitants. This is called the **propositions-as-types principle**, which reflects the observation that proofs are nothing but a certain kind of mental construction.

Notice that the notion of "effective methods" — or computation — was presumed when the notion of sets was introduced, and at some point we need to concretely specify an effective method. Since the description of every set includes an effective way to construct its canonical inhabitants, it is possible to express an effective method that mimics the construction of an inhabitant by saying that the computation has the same structure as how the inhabitant is constructed, and the computation is guaranteed to terminate since the construction of the inhabitant is finitary. For a typical example, let us look again at the natural numbers. Suppose that we have a **family of sets** $P \,:\, \mathsf{Nat} \to \mathsf{Set}$ indexed by inhabitants of Nat. (Since we only aim to present a casual sketch of type theory, we take the liberty of using AGDA functions (including Set-computing ones like P above) in places where terms under contexts should have been used.) If we have an inhabitant z of P zero and a method s that, for any n : Nat, transforms an inhabitant of $P\ n$ to an inhabitant of P (suc n), then we can compute an inhabitant of $P\ n$ for any given n by essentially the same counting process with which we construct n, but the counting now starts from z instead of zero and proceeds with s instead of suc. For instance, if a proof of P 2 is required, we can simply apply s to z twice, just like we apply suc to zero twice to form 2, so the computation was guided by the structure of 2. This explanation justifies the following **elimination rule**

$$\frac{P : \mathsf{Nat} \to \mathsf{Set} \qquad z : P\ \mathsf{zero} \qquad s : (n : \mathsf{Nat}) \to P\ n \to P\ (\mathsf{suc}\ n) \qquad n : \mathsf{Nat}}{\mathsf{Nat\text{-}elim}\ P\ z\ s\ n : P\ n}$$

(The type of s illustrates AGDA's syntax for dependent function types — the value n of the first argument is referred to in the types of the second argument and the result.) The symbol Nat-elim symbolises the method described above, which, given P, z, and s, transforms any natural number n into an inhabitant of the set $P\ n$. The actual computation performed by Nat-elim is stated as two **computation rules** in the form of equality judgements (see Section 2.3):

$$\frac{P : \mathsf{Nat} \to \mathsf{Set} \qquad z : P\ \mathsf{zero} \qquad s : (n : \mathsf{Nat}) \to P\ n \to P\ (\mathsf{suc}\ n)}{\mathsf{Nat\text{-}elim}\ P\ z\ s\ \mathsf{zero} = z \in P\ \mathsf{zero}}$$

$$\frac{P : \mathsf{Nat} \to \mathsf{Set} \qquad z : P\ \mathsf{zero} \qquad s : (n : \mathsf{Nat}) \to P\ n \to P\ (\mathsf{suc}\ n) \qquad n : \mathsf{Nat}}{\mathsf{Nat\text{-}elim}\ P\ z\ s\ (\mathsf{suc}\ n) = s\ n\ (\mathsf{Nat\text{-}elim}\ P\ z\ s\ n) \in P\ (\mathsf{suc}\ n)}$$

From the logic perspective, predicates on Nat are a special case of Nat-indexed families of sets like P; Nat-elim then delivers the induction principle for natural numbers, as it produces a proof of $P\ n$ for every n : Nat if the base case z and the inductive case s can be proved. In general, the propositions-as-types principle treats logical entities as ordinary mathematical objects; the logic hence inherits the computational meaning of intuitionistic mathematics and becomes constructive.

By enabling the interplay of various sets governed by rules like the above ones, type theory is capable of formalising various mental constructions we manipulate in mathematics in a fully computational way, making it a powerful programming language. As Martin-Löf [1984a] noted: "If programming is understood [...] as the design of the methods of computation [...], then it no longer seems possible to distinguish the discipline of programming from constructive mathematics". Indeed, sets are easily comparable with inductive datatypes in functional programming — a formation rule names a datatype, the associated introduction rules list the constructors of the datatype, and the associated elimination rule and computation rules define a precisely typed version of primitive recursion on the datatype. Consequently, we identify "sets" with "types", and regard them as interchangeable terms.

The uniform treatment of programs and proofs in type theory reveals new possibilities regarding proofs of program correctness. Traditional mathematical theories employ a standalone logic language for talking about some postulated objects. For example, Peano arithmetic is set up by postulating axioms about natural numbers in the language of first-order logic. Inside the postulated system of natural numbers, there is no knowledge of logic formulas or proofs (except via exotic encodings) — logic is at a higher level than the objects it is used for talking about. Programming systems based on such principle (e.g., Hoare logic) then need to have a meta-level logic language to reason about properties of programs. In **dependently typed** languages based on type theory, however, the two traditional levels are coherently integrated into one, so programs and their correctness proofs can be constructed together in the same language. For example, the proposition $\forall\, a : A.\ \exists\, b : B.\ R\ a\ b$ is interpreted as the type of a function taking $a : A$ to a pair consisting of $b : B$ and a proof of the proposition $R\ a\ b$, so the output of type B is guaranteed to be related by R to the input of type A. Checking of proof validity reduces to typechecking, and correctness proofs coexist with programs, as opposed to being separately presented at a meta-level.

The propositions-as-types principle, however, can lead to a more intimate form of program correctness by construction by blurring the distinction between programs and proofs even further; this form of program correctness — called **internalism** — is introduced in Section 2.5, which opens the central topic studied in this dissertation. Before that, we make a transition from type theory to practical programming in Agda, starting with its pattern matching notation.

2.2 Elimination and pattern matching

The formation rules and introduction rules for sets in type theory directly translate into inductive datatype declarations in functional programming. For example, the set of natural numbers is translated into Agda as an inductive datatype with two constructors, with their full types displayed:

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

In type theory, computations on inductive datatypes are specified using eliminators like Nat-elim, whose style corresponds to **recursion schemes** [Meijer et al., 1991] in functional programming. (In particular, Nat-elim is a more informatively typed version of paramorphism on natural numbers [Meertens, 1992].) One reason for making elimination as the only option is that programs in type theory are required to terminate — which is a consequence of the requirement that an inhabitant should be effectively recognisable as belonging to a set, and results in decidable typechecking — and using eliminators throughout is a straightforward way of enforcing termination. On the other hand, in functional programming, the **pattern matching** notation is widely used for defining programs on (inductive) datatypes (see, e.g., Hudak et al. [2007, Section 5]) in addition to recursion schemes. Pattern matching is vital to the clarity of functional programs because it not only allows a function to be intuitively defined by equations suggesting how the function computes, but also clearly conveys the programming strategy of splitting a problem into sub-problems by case analysis.

When it comes to dependently typed programming, the situation becomes more complicated due to the presence of **inductive families** [Dybjer, 1994], i.e., simultaneously inductively defined families of sets, like the following:

```
data _≤N_ (m : Nat) : Nat → Set where
  refl : m ≤N m
```

$$\mathsf{step} : (n : \mathsf{Nat}) \to m \leqslant_{\mathsf{N}} n \to m \leqslant_{\mathsf{N}} \mathsf{suc}\, n$$

(An Agda name with underscores (like $\_\leqslant_{\mathsf{N}}\_$) can be applied to arguments either by prefixing (like "$\_\leqslant_{\mathsf{N}}\_\ m\ n$") or by substituting the arguments for the underscores with proper spacing (like "$m \leqslant_{\mathsf{N}} n$").) Reading the declaration logically, the types of the two constructors refl and step give the two proof rules for establishing that one natural number is less than or equal to another. More generally, we read the declaration as a datatype parametrised by $m : \mathsf{Nat}$ (as signified by its appearance right next to **data** $\_\leqslant_{\mathsf{N}}\_$) and indexed by Nat. For any $m : \mathsf{Nat}$, the type family $\_\leqslant_{\mathsf{N}}\_\ m : \mathsf{Nat} \to \mathsf{Set}$ as a whole is inductively populated: we have an inhabitant refl in the set $(\_\leqslant_{\mathsf{N}}\_\ m)\ m$, and whenever we have an inhabitant $p : (\_\leqslant_{\mathsf{N}}\_\ m)\ n$ for some $n : \mathsf{Nat}$, we can make a larger inhabitant $\mathsf{step}\ n\ p$ in another set $(\_\leqslant_{\mathsf{N}}\_\ m)\ (\mathsf{suc}\ n)$ in the family. (From now on we usually refer to inductive families simply as datatypes, especially when emphasising their use in programming and de-emphasising the distinction with non-indexed datatypes like Nat.)

With inductive families, splitting a problem into sub-problems by case analysis in dependently typed programming often leads to nontrivial refinement of the goal type and the context, and such refinement can be tricky to handle with eliminators. Admittedly, in terms of expressive power, pattern matching and elimination are basically equivalent, as eliminators can be easily defined by dependent pattern matching, and conversely, it has been shown that dependent pattern matching can be reduced to elimination if **uniqueness of identity proofs** — or, equivalently, the **K axiom** [Streicher, 1993] — is assumed [McBride, 1999; Goguen et al., 2006]. (See Section 2.3 for more on uniqueness of identity proofs.) Nevertheless, there is a significant notational advantage of recovering pattern matching in dependently typed programming, especially with the support of an interactive development environment for managing detail about datatype indices. Below we look at an example of interactively constructing a program with pattern matching in Agda, whose design was inspired by Epigram [McBride, 2004; McBride and McKinna, 2004].

2.2.1 Pattern matching and interactive development

Suppose that we are asked to prove that $\_\leqslant_{\mathsf{N}}\_$ is transitive, i.e., to construct the program

$$\mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant_{\mathsf{N}} y \to y \leqslant_{\mathsf{N}} z \to x \leqslant_{\mathsf{N}} z$$

(The "telescopic" quantification "$(x\ y\ z : \mathsf{Nat}) \to$" is a shorthand for "$(x : \mathsf{Nat})\ (y : \mathsf{Nat})\ (z : \mathsf{Nat}) \to$", which, in turn, is a shorthand for "$(x : \mathsf{Nat}) \to (y : \mathsf{Nat}) \to (z : \mathsf{Nat}) \to$".) We define *trans* interactively by first putting pattern variables for the arguments on the left of the defining equation and then leaving an "interaction point" — also called a "goal" — on the right, which is numbered 0. Agda then tells us that a term of type $x \leqslant_{\mathsf{N}} z$ is expected (shown in the goal).

$$\begin{array}{l}
\mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant_{\mathsf{N}} y \to y \leqslant_{\mathsf{N}} z \to x \leqslant_{\mathsf{N}} z \\
\mathit{trans}\ x\ y\ z\ p\ q \ = \ \{x \leqslant_{\mathsf{N}} z\}_0
\end{array}$$

We instruct Agda to perform case analysis on q, and there are two cases: refl and $\mathsf{step}\ w\ r$ where r has type $y \leqslant_{\mathsf{N}} w$. The original Goal 0 is split into two sub-goals, and unification is triggered for each sub-goal.

$$\begin{array}{llllll}
\multicolumn{6}{l}{\mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant_{\mathsf{N}} y \to y \leqslant_{\mathsf{N}} z \to x \leqslant_{\mathsf{N}} z} \\
\mathit{trans}\ x\ .z & z & p^{x \leqslant_{\mathsf{N}} z} & \mathsf{refl} & = & \{x \leqslant_{\mathsf{N}} z\}_1 \\
\mathit{trans}\ x & y & .(\mathsf{suc}\ w)\ p^{x \leqslant_{\mathsf{N}} y} & (\mathsf{step}\ w\ r^{y \leqslant_{\mathsf{N}} w}) & = & \{x \leqslant_{\mathsf{N}} \mathsf{suc}\ w\}_2
\end{array}$$

In Goal 1, the type of refl demands that y be unified with z, and hence the pattern variable y is replaced with a "dot pattern" $.z$ indicating that the value of y is determined by unification to be z. Therefore, upon enquiry, Agda tells us that the type of p in the context — which was originally $x \leqslant_{\mathsf{N}} y$ — is now $x \leqslant_{\mathsf{N}} z$ (shown in superscript next to p, which is not part of the Agda program but only appears when interacting with Agda). Similarly for Goal 2, z is unified with $\mathsf{suc}\ w$ and the goal type is rewritten accordingly. We see that the case analysis has led to two sub-problems with different goal types and contexts, where Goal 1 is easily solvable as there is a term in the context with the right type, namely p.

$$\begin{array}{llllll}
\multicolumn{6}{l}{\mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant_{\mathsf{N}} y \to y \leqslant_{\mathsf{N}} z \to x \leqslant_{\mathsf{N}} z} \\
\mathit{trans}\ x\ .z & z & p & \mathsf{refl} & = & p
\end{array}$$

$$\mathit{trans}\ x\ \ y\ .(\mathsf{suc}\ w)\ p^{x \leqslant_{\mathsf{N}} y}\ (\mathsf{step}\ w\ r^{y \leqslant_{\mathsf{N}} w})\ =\ \{x \leqslant_{\mathsf{N}} \mathsf{suc}\ w\}_2$$

The second goal type $x \leqslant_{\mathsf{N}} \mathsf{suc}\ w$ looks like the conclusion in the type of the term $\mathsf{step}\ w : x \leqslant_{\mathsf{N}} w \to x \leqslant_{\mathsf{N}} \mathsf{suc}\ w$, so we use this term to reduce Goal 2 to Goal 3, which now requires a term of type $x \leqslant_{\mathsf{N}} w$.

$$\begin{array}{lcl} \mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant_{\mathsf{N}} y \to y \leqslant_{\mathsf{N}} z \to x \leqslant_{\mathsf{N}} z \\ \mathit{trans}\ x\ .z\ \ z\qquad p\qquad \mathsf{refl} & = & p \\ \mathit{trans}\ x\ \ y\ .(\mathsf{suc}\ w)\ p^{x \leqslant_{\mathsf{N}} y}\ (\mathsf{step}\ w\ r^{y \leqslant_{\mathsf{N}} w}) & = & \mathsf{step}\ w\ \{x \leqslant_{\mathsf{N}} w\}_3 \end{array}$$

Now we see that the induction hypothesis term $\mathit{trans}\ x\ y\ w\ p\ r : x \leqslant_{\mathsf{N}} w$ has the right type. Solving Goal 3 with this term completes the program.

$$\begin{array}{lcl} \mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant_{\mathsf{N}} y \to y \leqslant_{\mathsf{N}} z \to x \leqslant_{\mathsf{N}} z \\ \mathit{trans}\ x\ .z\ \ z\qquad p\ \mathsf{refl} & = & p \\ \mathit{trans}\ x\ \ y\ .(\mathsf{suc}\ w)\ p\ (\mathsf{step}\ w\ r) & = & \mathsf{step}\ w\ (\mathit{trans}\ x\ y\ w\ p\ r) \end{array}$$

In contrast, if we stick to the default elimination approach in type theory, we would use the eliminator

$$\begin{array}{ll} \leqslant_{\mathsf{N}}\text{-}\mathsf{elim} : & (m : \mathsf{Nat})\ (P : (n : \mathsf{Nat}) \to m \leqslant_{\mathsf{N}} n \to \mathsf{Set}) \to \\ & ((t : m \leqslant_{\mathsf{N}} m) \to P\ m\ t) \to \\ & ((n : \mathsf{Nat})\ (t : m \leqslant_{\mathsf{N}} n) \to P\ n\ t \to P\ (\mathsf{suc}\ n)\ (\mathsf{step}\ n\ t)) \to \\ & (n : \mathsf{Nat})\ (t : m \leqslant_{\mathsf{N}} n) \to P\ n\ t \end{array}$$

and write

$$\begin{array}{l} \mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant_{\mathsf{N}} y \to y \leqslant_{\mathsf{N}} z \to x \leqslant_{\mathsf{N}} z \\ \mathit{trans}\ x\ y\ z\ p\ q\ =\ \leqslant_{\mathsf{N}}\text{-}\mathsf{elim}\ y\ (\lambda\, y'\ \_ \mapsto x \leqslant_{\mathsf{N}} y \to x \leqslant_{\mathsf{N}} y') \\ \qquad\qquad\qquad\qquad (\lambda\, \_\ p' \mapsto p')\ (\lambda\, w\ r\ \mathit{ih}\ p' \mapsto \mathsf{step}\ w\ (\mathit{ih}\ p'))\ z\ q\ p \end{array}$$

We are forced to write the program in continuation passing style, where the two continuations correspond to the two clauses in the pattern matching version and likewise have more specific goal types, and the relevant context (p in this case) must be explicitly passed into the continuations in order to be refined to a more specific type. Even with interactive support, the eliminator version is inherently harder to write and understand, especially when complicated dependent types are involved. If a function definition requires more than one level of elimination, then the advantage of using pattern matching over using eliminators becomes even more apparent.

2.2.2 Pattern matching on intermediate computation

It is often the case that we need to perform pattern matching not only on an argument but also on some intermediate computation. In simply typed languages, this is usually achieved by "case expressions", a special case being if-then-else expressions for booleans. But again, pattern matching on intermediate computation can make refinements to the goal type and the context in dependently typed languages, so case expressions — being more like eliminators — become less convenient. McBride and McKinna [2004] thus proposed **with-matching**, which generalises pattern guards [Peyton Jones, 1997] and shifts pattern matching on intermediate computations from the right of an equation to the left, thereby granting them equal status with pattern matching on arguments, in particular the power to refine contexts and goal types.

Example (*insertion into a list*). To demonstrate the syntax of **with**-matching, we give a simple example of writing the function inserting an element into a list as used in, e.g., insertion sort. (More precisely, the element is inserted at the rightmost position to the left of which all elements are strictly smaller.) First we define the usual list datatype:

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

and (throughout this dissertation) let Val : Set be equipped with a decidable total ordering, i.e., there is a relation

```
_≤_ : Val → Val → Set
```

with the following operations:

```
≤-refl    : {x : Val} → x ≤ x
≤-trans   : {x y z : Val} → x ≤ y → y ≤ z → x ≤ z
_≤?_      : (x y : Val) → Dec (x ≤ y)
≰-invert  : {x y : Val} → ¬ (x ≤ y) → y ≤ x
```

where Dec is the following datatype witnessing whether a set is inhabited or not:

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

```
  yes :     A → Dec A
  no  : ¬ A → Dec A   -- ¬ A = A → ⊥, where ⊥ is the empty set
```

(Quantifications like $\{x : \mathit{Val}\}$ are implicit arguments to a function. They can be omitted when applying the function, and AGDA will try to infer them.) The insertion function is then written as

```
insert : Val → List Val → List Val
insert y [] = y :: []
insert y (x :: xs) with y ≤? x
insert y (x :: xs) | yes _ = y :: x :: xs
insert y (x :: xs) | no _  = x :: insert y xs
```

The result of the intermediate computation $y \leqslant_? x$: Dec $(y \leqslant x)$ is matched against yes and no on the left-hand side of the last two equations, just like we are performing pattern matching on a new argument. (In fact, AGDA implements **with**-matching exactly by synthesising an auxiliary function with an additional argument [Norell, 2007, Section 2.3].) The witnesses carried by yes and no are ignored in this case (their names are suppressed by underscores), but in general these can be proofs that are further matched and change the context and the goal type. (Admittedly, this is a trivial example with regard to the full power of **with**-matching, but *insert* will be used in later examples in this dissertation.) □

An important application of **with**-matching is McBride and McKinna's adaptation [2004] of Wadler's **views** [1987] — or "customised pattern matching" — for dependently typed programming. Suppose that we wish to implement a snoc-list view for cons-lists, i.e., to say that a list is either empty or has the form $ys +\!\!+ (y :: [])$ (where $\_+\!\!+\_$ is list append, a definition of which is shown in Section 2.4.2). We would define the following view type

```
data SnocView {A : Set} : List A → Set where
  nil  : SnocView []
  snoc : (ys : List A) (y : A) → SnocView (ys ++ (y :: []))
```

and write a **covering function** for the view:

```
snocView : {A : Set} (xs : List A) → SnocView xs
snocView [] = nil
snocView (x :: xs)                 with snocView xs
```

$$\begin{array}{lll} \mathit{snocView}\ (x :: .[\,]) & \mid \mathsf{nil} & = \mathsf{snoc}\ [\,]\ x \\ \mathit{snocView}\ (x :: .(\mathit{ys} +\!\!+ (y :: [\,]))) & \mid \mathsf{snoc}\ \mathit{ys}\ y & = \mathsf{snoc}\ (x :: \mathit{ys})\ y \end{array}$$

Note that the type of *snocView* ensures that every list is covered under one constructor of SnocView. Also, this is a nontrivial example of **with**-matching, because performing pattern matching on the result of *snocView xs* refines *xs* in the context to either [] or $\mathit{ys} +\!\!+ (y :: [\,])$, and the refinement is propagated to the goal type. Now, for example, the function *init* which removes the last element (if any) in a list can be implemented simply as

$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{init}\ :\ \{A\ :\ \mathsf{Set}\} \to \mathsf{List}\ A \to \mathsf{List}\ A} \\ \mathit{init}\ \mathit{xs} & \multicolumn{2}{l}{\textbf{with}\ \mathit{snocView}\ \mathit{xs}} \\ \mathit{init}\ .[\,] & \mid \mathsf{nil} & = [\,] \\ \mathit{init}\ .(\mathit{ys} +\!\!+ (y :: [\,])) & \mid \mathsf{snoc}\ \mathit{ys}\ y & = \mathit{ys} \end{array}$$

Views are not enough for dependently typed programming, though; they offer customised case analyses but not terminating recursion. McBride and McKinna [2004] proposed a general mechanism for invoking any programmer-defined eliminator using the pattern matching syntax, so the programmer can choose whichever recursive problem-splitting strategy they needs and express it conveniently with pattern matching. This mechanism is not implemented in AGDA, but it is implemented in EPIGRAM, and greatly improves readability of dependently typed programs. Specifically, EPIGRAM syntax makes it clear which and in what order eliminators are invoked, so programs are easily guaranteed to be based on elimination — and thus be terminating — while remaining readable. AGDA's design does not emphasise reducibility to elimination, but almost all the recursive programs in this dissertation are written with this in mind, so termination is evident even without understanding AGDA's termination checker in detail. Also, although AGDA provides pattern matching only for datatype constructors, all but one of the recursive programs in this dissertation use default structural induction (without need of programmer-defined elimination), so AGDA syntax suffices. (The exception is the final program in **??**, where we use an explicit eliminator.)

2.3 Equality

In logic, the **intension** of a concept is its defining description, while the **extension** of the concept is the range of objects it refers to. Two concepts can differ intensionally yet agree extensionally when they use different ways to describe the same range of objects. Classical mathematics cares about extensions only (e.g., the axiom of extensionality in set theory defines that two sets are equal exactly when they have the same inhabitants, regardless of how they are described), whereas in intuitionistic mathematics, objects are given to us as mental constructions, which are inherently intensional descriptions. As a consequence, the fundamental equality for intuitionistic mathematics is intensional [Dummett, 2000, Section 1.2], since we can only compare the intensional descriptions given to us, and furthermore it might be impossible to effectively recognise whether two intensionally different constructions are extensionally the same. For example, functions describing different computational procedures are intensionally distinguished even when they always map the same input to the same output, as it is well known that pointwise equality of functions is undecidable. We can, of course, still talk about extensional equalities in intuitionistic mathematics, but they are just treated as ordinary propositions.

The fundamental equality is formulated in type theory as **judgemental equality**, a meta-level notion for determining whether two types match in typechecking, which also involves determining whether two terms match because types can contain terms. (The computation rules for Nat-elim in Section 2.1 are examples of judgemental equalities between terms.) If we take the position of intuitionistic mathematics seriously, judgemental equality would be chosen to be the intensional, syntactic equality — also called **definitional equality** — which can be implemented by reducing two types/terms to normal forms and checking whether the normal forms match. The resulting type theory is called an **intensional type theory**; its characteristic feature is decidable typechecking (enabling effective recognition of set membership) due to decidability of judgemental equality. Agda, in particular, is intensional in this sense.

Judgemental equality — being a meta-level notion — is not an entity inside the theory. To state equality between two terms as a proposition and have proof for that proposition inside the theory, we need **propositional equality**, which can be defined in Agda by the following inductive family:

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

The canonical way to prove an equality proposition $x \equiv y$ is refl, which is permitted when x and y are judgementally equal. Under contexts, however, it is possible to prove that two judgementally different terms are propositionally equal. For example, the following "catamorphic" identity function on natural numbers

```
id′ : Nat → Nat
id′ zero      = zero
id′ (suc n)  = suc (id′ n)
```

can be shown to be pointwise equal to the polymorphic identity function

```
id : {A : Set} → A → A
id x = x
```

That is, given n : Nat in the context, even though the two open terms *id n* (which is definitionally just n) and *id′ n* are judgementally different, we can still prove *id n* $\equiv$ *id′ n* by induction (elimination) on n, whose two cases instantiate n to a more specific form and make computation on *id′ n* happen. One might say that propositional equality — in this most basic, inductive form — is "delayed" judgemental equality as a proposition: the judgementally different terms *id n* and *id′ n* would compute to the same canonical term — and hence become judgementally equal — after substituting a canonical natural number for n, turning them into closed terms and allowing the computation to complete. Formally, this is stated as the "reflection principle": two propositionally equal closed terms are judgementally equal (see, e.g., Luo [1994, Section 5.1.3] and Streicher [1993, Section 1.1]).

A propositional equality satisfying the reflection principle — e.g., the Agda one — can sometimes be too discriminating. For example, in category theory (which we use in Chapters **??** and **??**), a universal function (i.e., a universal morphism in the category of sets and total functions) is unique up to extensional (i.e., pointwise) equality, but pointwise propositionally equal functions are usually not propositionally equal themselves. (If, under the empty context, two pointwise propositionally equal functions are propositionally equal themselves, then by the reflection principle they are also judgementally equal, but the two functions can well have different intensions.) Of course, we can explicitly work with pointwise equality on functions, i.e., establishing and using properties formulated in terms of the relation

$$
\begin{aligned}
&\_\doteq\_ \;:\; \{A\; B \;:\; \mathsf{Set}\} \to (A \to B) \to (A \to B) \to \mathsf{Set}\\
&\_\doteq\_ \;\{A\}\, f\, g \;=\; (x \;:\; A) \to f\; x \equiv g\; x
\end{aligned}
$$

(The implicit argument A is explicitly displayed in curly braces on the left-hand side of the equation since we need to refer to it on the right-hand side.) Not being able to identify extensionally equal functions propositionally, however, means that we do not automatically get basic properties like

$$\mathit{cong} \;:\; \{A\; B \;:\; \mathsf{Set}\}\; (f \;:\; A \to B)\; \{x\, y \;:\; A\} \to x \equiv y \to f\; x \equiv f\; y$$

i.e., a function maps equal arguments to equal results, and

$$\mathit{subst} \;:\; \{A \;:\; \mathsf{Set}\}\; (P \;:\; A \to \mathsf{Set})\; \{x\, y \;:\; A\} \to x \equiv y \to P\; x \to P\; y$$

i.e., inhabitants in an indexed set can be transported to another set with an equal index, for extensionally equal functions. We would need to prove such properties for every entity like f and P on a case-by-case basis, which quickly becomes tedious.

Several foundational modifications to intensional type theory have been proposed to obtain a more liberal notion of equality. While this dissertation sticks to AGDA's intensional approach and does not adopt any of the alternatives, it is interesting to reflect on the relationship between these alternatives and our development.

- A simple yet radical approach is to add the **equality reflection rule** to the theory, injecting propositional equality back into judgemental equality.

  $$\frac{x \;:\; A \qquad y \;:\; A \qquad \mathit{eq} \;:\; x \equiv y}{x \;=\; y \;\in\; A}$$

  Extensionally equal functions become judgementally equal (and thus propositionally equal) in such a theory: Suppose that f and g are functions of type $A \to B$ and we have a proof

$fgeq : (x : A) \to f\ x \equiv g\ x$

Then, judgementally,

$$\begin{array}{ll} & f \\ = & \{\ \eta\text{-expansion}\ \} \\ & \lambda\, x \mapsto f\ x \\ = & \{\ \text{equality reflection} — f\ x = g\ x \in B \text{ since } fgeq\ x : f\ x \equiv g\ x\ \} \\ & \lambda\, x \mapsto g\ x \\ = & \{\ \eta\text{-contraction}\ \} \\ & g \qquad \in A \to B \end{array}$$

The judgemental equality is thus able to identify pointwise equal functions and becomes extensional, and such a theory is called an **extensional type theory** (see, e.g., Nordström et al. [1990, Section 8.2]). In extensional type theory, combinators like *subst* become unnecessary, since having a proof of $x \equiv y$ means that x and y are identified judgementally and hence are regarded as the same during typechecking, so $P\ x$ and $P\ y$ are simply the same type — no explicit transportation is needed. Consequently, programs become very lightweight, with all the equality justifications moved to typing derivations at the meta-level. The downside is that typechecking in extensional type theory is undecidable, because whenever there is possibility that the equality reflection rule is needed, the typechecker would have to somehow determine whether there is a suitable equality proof, for which there is no effective procedure. This is not a big problem for proof assistants like Nuprl [Constable et al., 1985], in which the programmer instructs the proof assistant to construct typing derivations and can supply the right proof when using the equality reflection rule. (Nuprl, in fact, simply identifies judgemental equality and propositional equality and does not have the equality reflection rule explicitly.) But for programming languages like Ωmega [Sheard and Linger, 2007], equality reflection does present a problem, since the programmer constructs a term only, and the typing derivation has to be constructed by the typechecker, which then has to search for proofs. Ωmega can take hints from the programmer so the proof search is more likely to succeed, but the fundamental problem is that justification of program correctness now relies on the proof searching algorithm and is tied to the implementation detail of a specific programming system. Since the focus of this dissertation is on dependently typed programming, extensional type theory is not a satisfactory foundation.

- Altenkirch et al. [2007] proposed a variant of intensional type theory called **observational type theory**, which defines propositional equality to be an extensional one — in particular, propositional equality on functions is pointwise equality — but retains computational behaviour (strong normalisation and canonicity) and decidable typechecking of intensional type theory. We step away from observational type theory merely for a practical reason: the theory is not implemented natively in any programming system yet. While it might be possible to model observational equality in Agda to some extent and then construct the universes of descriptions (Section 2.4) and ornaments (**??**) inside the observational model, developing and programming within the nested model would be too complex to be worthwhile.
- A new direction is being pursued by **homotopy type theory** [The Univalent Foundations Program, 2013], which gives propositional equality a higher-dimensional homotopic interpretation and broadens its scope with the univalence axiom and higher inductive types, but the computational meaning of the theory remains an open problem. Its investigations into the higher dimensional structure of propositional equality might eventually lead to a systematic treatment of equality in dependently typed programming. For this dissertation, however, we confine ourselves to the zero-dimensional setting, in which we freely invoke **uniqueness of identity proofs**, which is definable in Agda by pattern matching:

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

  Our identification of types and sets is thus consistent with the terminology of homotopy type theory: types on which identity proofs are unique (and hence lack higher dimensional structure) are indeed termed “sets” by homotopy type theorists [The Univalent Foundations Program, 2013, Section 3.1].

With only Agda’s intensional equality, we have to explicitly work with equality-like propositions (like pointwise equality on functions) and manage them with the help of **setoids** [Barthe et al., 2003] in this dissertation — Chapters **??** and **??**, specifically. We will discuss to what extent the intensional approach works at the end of **??**.

2.4 Universes and datatype-generic programming

Martin-Löf [1984b] introduced the notion of **universes** to support "large elimination", i.e., arbitrary computation of sets, which is necessary for proving the fourth Peano axiom that zero is not the successor of any natural number [Smith, 1988]. A universe (à la Tarski) is a set of codes for sets, which is equipped with a decoding function translating codes to sets. Large elimination is then computing an inhabitant in the universe (via ordinary elimination like Nat-elim) and decoding the result to a set. Another important purpose of universes is to support quantification over sets while precluding paradoxical formation of self-referential sets — AGDA, for example, has a universe hierarchy (Set, Set_1, Set_2, ...) for this purpose. From the programming perspective, however, the most interesting case is when the universe is supplied with an elimination rule [Nordström et al., 1990, Section 14.2]: allowing computation on universes turns out to roughly correspond to **datatype-generic programming** [Gibbons, 2007], whose idea is that the "shapes" of datatypes can be encoded so programs can be defined in terms of these shapes. Universes can encode such shapes, and since universes are just ordinary sets, datatype-generic programming becomes just ordinary programming in dependently typed languages [Altenkirch and McBride, 2003].

In this dissertation we not only use but also need to compute a class of inductive families which we call **index-first datatypes** [Chapman et al., 2010; Dagand and McBride, 2012b], and hence need to construct a universe for them. Before that, we give a high-level introduction to these datatypes first.

2.4.1 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. For example, consider the following datatype of **vectors**, i.e., length-indexed lists:

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

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 [2012b] 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 use a slightly more Agda-like adaptation of the notation.

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, distinguishing it from an Agda datatype declaration. 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 $A : \mathsf{Set}$:

$$\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{List}\ (A : \mathsf{Set}) : \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{List}\ A \ni [\,] \\
\qquad\qquad \mid\ \_{::}\_\ (a : A)\ (as : \mathsf{List}\ A)
\end{array}$$

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

$$\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{Vec}\ (A : \mathsf{Set}) : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{Vec}\ A\ \mathsf{zero} \qquad \ni [\,] \\
\quad \mathsf{Vec}\ A\ (\mathsf{suc}\ n) \ni \_{::}\_\ (a : A)\ (as : \mathsf{Vec}\ A\ n)
\end{array}$$

$\mathsf{Vec}\ A$ is a family of types indexed by Nat, and we do pattern matching on the index, splitting the datatype into two cases $\mathsf{Vec}\ A\ \mathsf{zero}$ and $\mathsf{Vec}\ A\ (\mathsf{suc}\ n)$ for some $n : \mathsf{Nat}$. 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 $as : \mathsf{Vec}\ A\ n$ 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:

$$\begin{array}{l}
\textbf{indexfirst data}\ \mathsf{Vec}'\ (A : \mathsf{Set}) : \mathsf{Nat} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{Vec}'\ A\ n \ni \mathsf{nil}\ (neq : n \equiv \mathsf{zero}) \\
\qquad\qquad\quad \mid\ \mathsf{cons}\ (a : A)\ \{m : \mathsf{Nat}\} \\
\qquad\qquad\qquad\qquad (as : \mathsf{Vec}'\ A\ m)\ (meq : n \equiv \mathsf{suc}\ m)
\end{array}$$

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 $\mathsf{Vec}'\ A\ n$ 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. □

2.4.2 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 $A : \mathsf{Set}$)

```
VecF A : (Nat → Set) → (Nat → Set)
VecF A X zero      = ⊤
VecF A X (suc n)  = A × X n   -- _×_ is cartesian product
```

and take its least fixed point. (⊤ is a singleton set whose only inhabitant is "▪".) If we flip the order of the arguments of *VecF A*:

```
VecF′ A : Nat → (Nat → Set) → Set
VecF′ A zero      = λ X → ⊤
VecF′ A (suc n)  = λ X → A × X n
```

we see that *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

```
data RDesc (I : Set) : Set₁
```

with a decoding function specifying its semantics:

```
⟦_⟧ : {I : Set} → RDesc I → (I → Set) → Set
```

Inhabitants of 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

```
Desc : Set → Set₁
Desc I = I → RDesc I
```

with decoding function

```
𝔽 : {I : Set} → Desc I → (I → Set) → (I → Set)
𝔽 D X i = ⟦ D i ⟧ X
```

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

$$
\begin{array}{l}
\textbf{data}\ \mu\ \{I : \mathsf{Set}\}\ (D : \mathsf{Desc}\ I) : I \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{con} : \mathbb{F}\ D\ (\mu\ D) \rightrightarrows \mu\ D
\end{array}
$$

where $\_{\rightrightarrows}\_$ is defined by

$$
\begin{array}{l}
\_{\rightrightarrows}\_ : \{I : \mathsf{Set}\} \to (I \to \mathsf{Set}) \to (I \to \mathsf{Set}) \to \mathsf{Set} \\
\_{\rightrightarrows}\_\ \{I\}\ X\ Y \ = \ \{i : I\} \to X\ i \to Y\ i
\end{array}
$$

Remark (*presentation of universes and their decoding*). We always present universes (e.g., RDesc) along with their decoding (e.g., $[\![\_]\!]$ for RDesc) to emphasise the meaning of the codes, even when the decoding is not logically tied to the codes (cf. Martin-Löf's universe [1984b], which is inductive-recursive [Dybjer, 1998] and must present the universe and its decoding simultaneously). □

Notation (*dependent pairs and* AGDA *records*). Cartesian product is a special case of **Σ-types**, also known as **dependent pairs**, which are defined in AGDA as a record:

$$
\begin{array}{l}
\textbf{record}\ \Sigma\ (A : \mathsf{Set})\ (X : A \to \mathsf{Set}) : \mathsf{Set}\ \textbf{where} \\
\quad \textbf{constructor}\ \_,\_ \\
\quad \textbf{field} \\
\qquad \mathit{outl} : A \\
\qquad \mathit{outr} : X\ \mathit{outl} \\
\textbf{infixr}\ 4\ \_,\_ \\
\textbf{open}\ \Sigma \\
\textbf{syntax}\ \Sigma\ A\ (\lambda\, a \mapsto T) \ = \ \Sigma[\, a : A\,]\ T
\end{array}
$$

An inhabitant of $\Sigma\ A\ X$ is a pair where the type of the second component depends on the first component; it is written by listing values for the fields like

$$
\begin{array}{l}
\textbf{record}\ \{\ \mathit{outl} \ = \ a \\
\qquad\quad ;\ \mathit{outr} \ = \ x\ \}
\end{array}
$$

(where $a : A$ and $x : X\ a$) — a cartesian product $A \times B$ is thus a special case, which is defined as $\Sigma\ A\ (\lambda\ \_ \mapsto B)$. The **constructor** declaration gives rise to a constructor function

$$
\_,\_ : \{A : \mathsf{Set}\}\ \{X : A \to \mathsf{Set}\} \to (a : A) \to X\ a \to \Sigma\ A\ X
$$

which associates to the right when used as an infix operator because of the

infixr statement below, and can be used in pattern matching. The two field declarations give rise to two projection functions, qualified by "$\Sigma.$":

$$\begin{aligned}
&\Sigma.\mathit{outl} \;:\; \{A \;:\; \mathsf{Set}\}\; \{X \;:\; A \to \mathsf{Set}\} \to \Sigma\; A\; X \to A \\
&\Sigma.\mathit{outr} \;:\; \{A \;:\; \mathsf{Set}\}\; \{X \;:\; A \to \mathsf{Set}\} \to (p \;:\; \Sigma\; A\; X) \to X\; (\Sigma.\mathit{outl}\; p)
\end{aligned}$$

We can drop the qualifications and refer to them simply as *outl* and *outr* due to the **open** statement. Finally, we can treat Σ as a binder and write, e.g., $\Sigma\; A\; X$ as $\Sigma[\, a : A\,]\;\; X\; a$, due to the **syntax** statement. □

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

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

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

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

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 informally refer to the index part of a σ as a **field** of the datatype. Like Σ, we sometimes regard σ as a binder and write $\sigma[\, s : S\,]\;\; D\; s$ for $\sigma\; S\; (\lambda\, s \mapsto D\; s)$. □

Example (*natural numbers*)**.** The datatype of natural numbers is considered to be an inductive family trivially indexed by $\top$, so the declaration of Nat corresponds to an inhabitant of $\mathsf{Desc}\; \top$.

$$\begin{aligned}
&\textbf{data}\; \mathsf{ListTag} \;:\; \mathsf{Set}\; \textbf{where} \\
&\quad \text{'}\mathsf{nil} \quad\; :\; \mathsf{ListTag}
\end{aligned}$$

```
  ‘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. A pattern-matching lambda function (which is syntactically distinguished by enclosing its body in curly braces) follows, which computes the trailing responses to the two possible values ‘nil and ‘cons for the field: 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 (▪ :: []). □

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. □

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

```
VecD : Set → Desc Nat
VecD A zero    = v []
VecD A (suc n) = σ[ _ : A ] v (n :: [])
```

which is directly comparable to the index-first base functor *VecF′* at the beginning of this section. □

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

```
_++_ : μ (ListD A) ■ → μ (ListD A) ■ → μ (ListD A) ■
con (‘nil   ,          ■) ++ bs = bs
con (‘cons , a , as , ■) ++ bs = con (‘cons , a , as ++ bs , ■)
```

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

```
List : Set → Set
List A = μ (ListD A) ■

[] : {A : Set} → List A
[] = con (‘nil , ■)

_::_ : {A : Set} → A → List A → List A
a :: as = con (‘cons , a , as , ■)
```

List append can then be rewritten in the usual form:

```
_++_ : List A → List A → List A
[]        ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

(Agda supports such definitions of higher-level terms by “pattern synonyms”, which we do not explicit use in this dissertation.) Later on, encoded datatypes are almost always accompanied by corresponding index-first datatype declarations, which are 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 2.4.1. Index-first declarations will only be regarded in this dissertation 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 corresponding descriptions and definitions of higher-level terms. (One such treatment was given by Dagand and McBride [2012a].)

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:

```
fold : {I : Set} {D : Desc I} {X : I → Set} → (𝔽 D X ⇉ X) → (μ D ⇉ X)
```

There is also a generic *induction* operator, which can be used to prove generic

```
mutual
  fold : {I : Set} {D : Desc I} {X : I → Set} → (𝔽 D X ⇉ X) → (μ D ⇉ X)
  fold {I} {D} f {i} (con ds) = f (mapFold D (D i) f ds)

  mapFold : {I : Set} (D : Desc I) (D′ : RDesc I) →
            {X : I → Set} → (𝔽 D X ⇉ X) → ⟦ D′ ⟧ (μ D) → ⟦ D′ ⟧ X
  mapFold D (v [])        f ▪          = ▪
  mapFold D (v (i :: is)) f (d , ds)   = fold f d , mapFold D (v is) f ds
  mapFold D (σ S D′)      f (s , ds)   = s , mapFold D (D′ s) f ds
```

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

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

```
fold : {I : Set} {D : Desc I} {X : I → Set} → (𝔽 D X ⇉ X) → (μ D ⇉ X)
fold {I} {D} f {i} (con ds) = f (mapRD (D i) (fold f) ds)
```

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

```
mapRD : {I : Set} (D : RDesc I) →
        {X Y : I → Set} (g : X ⇉ Y) → ⟦ D ⟧ X → ⟦ D ⟧ Y
mapRD (v [])        g ▪         = ▪
mapRD (v (i :: is)) g (x , xs)  = g x , mapRD (v is) g xs
mapRD (σ S D)       g (s , xs)  = s , mapRD (D s) g xs
```

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 to Agda, we instead define *fold* mutually recursively with *mapFold*, which is *mapRD* specialised by fixing its argument g to *fold f*.

Example (*list length*). The function *length* : {*A* : Set} → List *A* → Nat can be defined in terms of the datatype-generic *fold*:

```
length : {A : Set} → List A → Nat
length = fold length-alg
```

where the algebra *length-alg* is defined by

```
length-alg : {A : Set} → 𝔽 (ListD A) (const Nat) ⇉ const Nat
length-alg (‘nil  ,         ■) = zero
length-alg (‘cons , a , n , ■) = suc n
```

□

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.

Remark (*first-order vs higher-order representation*). The functorial structures generated by descriptions are strongly reminiscent of **indexed containers** [Morris, 2007, Chapter 8]; 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 representations is merely to obtain a simpler equality for the manufactured datatypes (AGDA's default equality would suffice); the examples of manufactured datatypes in this dissertation 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 in **??**). 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. □

2.5 Externalism and internalism

The use of "such that" to describe objects that have certain properties is universal in mathematics. If the objects in question have type A, then objects with certain properties form a subset of A, and using "such that" to describe such objects means that the subset is formed by specifying a suitable predicate on A. In type theory, this can be modelled by Σ-types of dependent pairs. In a type $\Sigma\ A\ P$, when A is interpreted as a ground set and P as a predicate on A, an inhabitant of $\Sigma\ A\ P$ is an inhabitant a of A paired with a proof that $P\ a$ holds. When programs are the objects we reason about, this style naturally suggests a distinction between programs and proofs: programs are written in the first place, and proofs are conducted afterwards with reference to existing programs and do not interfere with their execution. This conception underlies many developments in type theory and theorem proving. For example: Luo [1994] consistently argued that proofs should not be identified with programs, one of the reasons being that logic should be regarded as independent from the objects being reasoned about. A type theory of subsets was given by Nordström et al. [1990] to suppress the second component — i.e., the proof part — of Σ-types. The proof assistant COQ [Bertot and Castéran, 2004] uses a type-theoretic foundation [Coquand and Huet, 1988; Coquand and Paulin-Mohring, 1990] which distinguishes programs and proofs, and proofs are written in a tactic-based language, differently from programs; it is also famous for the ability to extract executable portions of proof scripts into programs [Paulin-Mohring, 1989; Letouzey, 2003], as used in the CompCert project developing a verified C compiler [Leroy, 2009], for example.

On the other hand, having unified programs and proofs in type theory (Section 2.1), it seems a pity if the unification is not exploited to a deeper level. In Dijkstra's proposal for program correctness by construction, proofs and programs should be conceived "hand in hand" [Dijkstra, 1976], and the unification of programs and proofs brings us unprecedentedly closer to this ideal, since we can start thinking about programs that also serve as correctness proofs themselves. The dependently typed programming community has been exploring the use of inductive families not only for defining predicates on data (like $\_\leqslant_{\mathsf{N}}\_$) but also for representing data with embedded constraints (like Vec). Programs manipulating such datatypes would also deal with the embedded constraints and are thus correct by construction. Vector append is a classic (albeit somewhat trivial) example: Defining addition on natural numbers as

$$
\begin{array}{l}
\_+\_ \;:\; \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{zero} \quad\;\; + n \;=\; n \\
(\mathsf{suc}\; m) + n \;=\; \mathsf{suc}\; (m + n)
\end{array}
$$

vector append is then defined by

$$
\begin{array}{l}
\_{+\!\!+}\_ \;:\; \{A \,:\, \mathsf{Set}\}\; \{m\; n \,:\, \mathsf{Nat}\} \to \mathsf{Vec}\; A\; m \to \mathsf{Vec}\; A\; n \to \mathsf{Vec}\; A\; (m + n) \\
[\,] \qquad\quad {+\!\!+}\; ys \;=\; ys \\
(x :: xs) \;{+\!\!+}\; ys \;=\; x :: (xs \;{+\!\!+}\; ys)
\end{array}
$$

The program for vector append looks exactly like the one for list append except for the more informative type, which makes it possible for the program to meet its specification — the length of the result of append should be the sum of the lengths of the two input lists — by construction. For list append, whose type uses the plain list datatype, we need to separately produce the following proof:

$$
\begin{array}{l}
\mathit{append\text{-}length} \;: \\
\quad \{A \,:\, \mathsf{Set}\}\; (xs\; ys \,:\, \mathsf{List}\; A) \to \mathit{length}\; (xs \;{+\!\!+}\; ys) \;\equiv\; \mathit{length}\; xs + \mathit{length}\; ys \\
\mathit{append\text{-}length}\; [\,] \qquad\; ys \;=\; \mathsf{refl}
\end{array}
$$

$\mathit{append\text{-}length}\ (x :: xs)\ ys\ =\ \mathit{cong}\ \mathsf{suc}\ (\mathit{append\text{-}length}\ xs\ ys)$

But by switching to the vector datatype, the proof dissolves into the typing of the program and needs no separate handling. This is possible because list append and the proof *append-length* share the same structure; consequently, by careful type design, the vector append program alone is able to carry both of them simultaneously. We propose to call this programming style **internalism**, suggesting that proofs are internalised in programs, while the traditional proving-after-programming style is called **externalism**. Externalism is necessary when we wish to show that an existing program satisfies additional properties, especially when the proofs are complicated and do not follow the structure of the program. On the other hand, writing a (simply typed) program and an externalist proof following the structure of the program is like stating the same thing twice: even though the programmer knows the meaning of the program, they has to first state the meaningless symbol manipulation aspect and then explain its meaning via a separate proof, doubling the effort. In contrast, internalism is programming with informative datatypes so as to give precise description of meaningful computations, so explanations via separate proofs become unnecessary. As McBride [2004] aptly put it, internalism makes programs and their explanations via proofs "not merely coexist but coincide". In addition, by encoding meanings in datatypes, semantic considerations are (at least partially) reduced to syntax and can be aided mechanically — AGDA's interactive development environment (Section 2.2.1) is one form of such aid. With interactive development, internalist types not only passively rule out nonsensical programs, but can actively provide helpful information to guide program construction. We will see several examples of such "type-directed programming" in this dissertation.

Internalism comes with its own problems, however. For one: internalist type design is difficult, yet there is almost no effective guideline or discipline for such design. Carelessly designed internalist types can lead to less natural programs. A simple example is when we switch the order of the arguments of the addition in the type of vector append,

$$\begin{array}{l} \_\!+\!\!+\!\_\ :\ \{A\ :\ \mathsf{Set}\}\ \{m\ n\ :\ \mathsf{Nat}\} \to \mathsf{Vec}\ A\ m \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (n + m) \\ []\qquad +\!\!+\ ys\ =\ \mathit{subst}\ (\mathsf{Vec}\ A)\ \{\ n\ \equiv\ n + \mathsf{zero}\ \}_0\ ys \end{array}$$

$$(x :: xs) +\!\!+ ys \;=\; \mathit{subst}\ (\mathsf{Vec}\ A)\ \{\ \mathsf{suc}\ (n + m)\ \equiv\ n + \mathsf{suc}\ m\ \}_1\ (x :: (xs +\!\!+ ys))$$

we would be forced to perform two type-casts that could have been avoided. Not only does the program look ugly, but this can also cause a cascade effect when we need to write other programs whose types depend on this program (e.g., externalist proofs about it), which would have to deal with the type-casts. McBride [2012] gave a more interesting example: to prove the polynomial testing principle (two polynomial functions of degree n are pointwise equal if they agree at $n + 1$ different points), McBride started with a datatype of encodings of polynomial functions indexed by degree but, after trying to program with the datatype, quickly found out that the datatype should instead be indexed by an arbitrary upper bound of the degree so a relaxed form of the polynomial testing principle can be naturally programmed (two polynomial functions of degree <u>at most</u> n are pointwise equal if they agree at $n + 1$ different points). Scalability is another issue, especially when the only tool we have is the primitive language of datatype declarations: writing a datatype declaration with a sophisticated property internalised is comparable to programming a sophisticated algorithm in assembly language, and understanding the meaning of a complicated datatype declaration takes thorough reading and some inductive guessing and reasoning. In short, the complexity of internalist types has made type design a nontrivial programming problem, and we are in serious lack of type-level programming support.

Internalist library design also poses a problem: Since internalism requires differently indexed versions of the same data structure, an internalist library should provide more or less the same set of operations for all possible variants of the data structure. Without a way to manage these formally unrelated datatypes and operations modularly, an ad hoc library would need to duplicate the same structure and logic for all the variants and becomes hard to expand. For example, suppose that we have constructed a library for lists that include vectors, ordered lists, and ordered vectors, and now wish to add a new flavour of lists, say, association lists indexed with the list of keys. Operations need to be reimplemented not only for such key-indexed lists, but also for key-indexed vectors, ordered key-indexed lists, and ordered key-indexed vectors, even though key-indexing is the only new feature. An ideal structure for such a library would be having a separate module for each of the properties about length, ordering, and key-indexing. These modules can be developed independently, and there would be a way to assemble components in these modules at will — for example, ordered vectors and related operations would be synthesised from the components in the modules about length and ordering. This ideal library structure calls for some form of composability of internalist datatypes and operations.

Composability has never been a problem for externalism, however. In an externalist list library, we would have only one basic list datatype and several predicates on lists about length, ordering, key-indexing, etc. Lists are "promoted" to vectors, ordered lists, or ordered vectors by simply pairing the list datatype with the length predicate, the ordering predicate, or the pointwise conjunction of the two predicates, respectively. Common operations are implemented for basic lists only, and their properties regarding length or ordering are proved independently and invoked when needed. Can we somehow introduce this beneficial composability to internalism as well? The answer is yes, because there are isomorphisms between externalist and internalist datatypes to be exploited.

To illustrate, let us go through a small case study about upgrading the *insert* function on lists (Section 2.2.2) for vectors, ordered lists, and ordered vectors. The externalist would define vectors as a Σ-type,

$$
\begin{aligned}
&\mathit{ExtVec} \;:\; \mathsf{Set} \to \mathsf{Nat} \to \mathsf{Set}\\
&\mathit{ExtVec}\;A\;n \;=\; \Sigma[\,\mathit{xs} : \mathsf{List}\;A\,]\;\;\mathit{length}\;\mathit{xs} \;\equiv\; n
\end{aligned}
$$

prove that *insert* increases the length of a list by one,

$$
\begin{aligned}
\mathit{insert\text{-}length} \;:\; &(y \;:\; \mathit{Val})\;\{n \;:\; \mathsf{Nat}\}\;(\mathit{xs} \;:\; \mathsf{List}\;\mathit{Val}) \to\\
&\mathit{length}\;\mathit{xs} \;\equiv\; n \to \mathit{length}\;(\mathit{insert}\;y\;\mathit{xs}) \;\equiv\; \mathsf{suc}\;n
\end{aligned}
$$

and define insertion on vectors as

$$
\begin{aligned}
&\mathit{insert}_{EV} \;:\; \mathit{Val} \to \{n \;:\; \mathsf{Nat}\} \to \mathit{ExtVec}\;\mathit{Val}\;n \to \mathit{ExtVec}\;\mathit{Val}\;(\mathsf{suc}\;n)\\
&\mathit{insert}_{EV}\;y\;(\mathit{xs}\;,\;\mathit{len}) \;=\; \mathit{insert}\;y\;\mathit{xs}\;,\;\mathit{insert\text{-}length}\;y\;\mathit{xs}\;\mathit{len}
\end{aligned}
$$

which processes the list and the length proof by *insert* and *insert-length* respectively. Similarly for ordered lists (indexed with a lower bound), the externalist would use the Σ-type

$$\begin{aligned}&\mathit{ExtOrdList} \;:\; \mathit{Val} \rightarrow \mathsf{Set}\\&\mathit{ExtOrdList}\; b \;=\; \Sigma[\,\mathit{xs} : \mathsf{List}\;\mathit{Val}\,]\;\mathsf{Ordered}\; b\; \mathit{xs}\end{aligned}$$

where the Ordered predicate is defined by

```
indexfirst data Ordered : Val → List Val → Set where
  Ordered b []        ∋ nil
  Ordered b (x :: xs) ∋ cons (leq : b ⩽ x) (ord : Ordered x xs)
```

Insertion on ordered lists is then

$$\begin{aligned}&\mathit{insert}_{\mathit{EO}} \;:\; (y : \mathit{Val})\;\{b : \mathit{Val}\} \rightarrow \mathit{ExtOrdList}\; b \rightarrow\\&\qquad\qquad \{b' : \mathit{Val}\} \rightarrow b' \leqslant y \rightarrow b' \leqslant b \rightarrow \mathit{ExtOrdList}\; b'\\&\mathit{insert}_{\mathit{EO}}\; y\; (\mathit{xs}\;,\mathit{ord})\; b'{\leqslant}y\; b'{\leqslant}b \;=\; \mathit{insert}\; y\; \mathit{xs}\;,\mathit{insert\text{-}ordered}\; y\; \mathit{xs}\; \mathit{ord}\; b'{\leqslant}y\; b'{\leqslant}b\end{aligned}$$

where *insert-ordered* proves that *insert* preserves ordering:

$$\begin{aligned}&\mathit{insert\text{-}ordered} \;:\; (y : \mathit{Val})\;\{b : \mathit{Val}\}\;(\mathit{xs} : \mathsf{List}\;\mathit{Val}) \rightarrow \mathsf{Ordered}\; b\; \mathit{xs} \rightarrow\\&\qquad\qquad \{b' : \mathit{Val}\} \rightarrow b' \leqslant y \rightarrow b' \leqslant b \rightarrow \mathsf{Ordered}\; b'\; (\mathit{insert}\; y\; \mathit{xs})\end{aligned}$$

Now the externalist has arrived at a modular list library (albeit a tiny one), which contains

- a basic module consisting of the basic list datatype and insertion on basic lists, and
- two independent upgrading modules about length and ordering, each consisting of a predicate on lists and a related proof about insertion.

It is easy to mix all three modules and get ordered vectors and insertion on them. The Σ-type uses the pointwise conjunction of the two predicates,

$$\begin{aligned}&\mathit{ExtOrdVec} \;:\; \mathit{Val} \rightarrow \mathsf{Nat} \rightarrow \mathsf{Set}\\&\mathit{ExtOrdVec}\; b\; n \;=\; \Sigma[\,\mathit{xs} : \mathsf{List}\;\mathit{Val}\,]\;\mathsf{Ordered}\; b\; \mathit{xs} \times \mathit{length}\; \mathit{xs} \equiv n\end{aligned}$$

and insertion simply uses *insert-ordered* and *insert-length* to process the two proofs bundled with a list:

$$\begin{aligned}&\mathit{insert}_{\mathit{EOV}} \;:\; (y : \mathit{Val})\;\{b : \mathit{Val}\}\;\{n : \mathsf{Nat}\} \rightarrow \mathit{ExtOrdVec}\; b\; n \rightarrow\\&\qquad\qquad \{b' : \mathit{Val}\} \rightarrow b' \leqslant y \rightarrow b' \leqslant b \rightarrow \mathit{ExtOrdVec}\; b'\; (\mathsf{suc}\; n)\\&\mathit{insert}_{\mathit{EOV}}\; y\; (\mathit{xs}\;,\mathit{ord}\;,\mathit{len})\; b'{\leqslant}y\; b'{\leqslant}b \;=\; \mathit{insert}\; y\; \mathit{xs}\;,\\&\qquad\qquad\qquad\qquad \mathit{insert\text{-}ordered}\; y\; \mathit{xs}\; \mathit{ord}\; b'{\leqslant}y\; b'{\leqslant}b\;,\\&\qquad\qquad\qquad\qquad \mathit{insert\text{-}length}\; y\; \mathit{xs}\; \mathit{len}\end{aligned}$$

This is the kind of library we are looking for, except that the types are all externalist. The externalist and internalist types are not unrelated, however. For example, internalist and externalist vectors are related by the indexed family of **conversion isomorphisms**:

$$\mathit{Vec\text{-}iso}\ A\ :\ (n\ :\ \mathsf{Nat}) \to \mathsf{Vec}\ A\ n \cong \mathit{ExtVec}\ A\ n$$

Fixing $n\ :\ \mathsf{Nat}$, the left-to-right direction of the isomorphism

$$\mathsf{Iso}.\mathit{to}\ (\mathit{Vec\text{-}iso}\ A\ n)\ :\ \mathsf{Vec}\ A\ n \to \Sigma[\,\mathit{xs}\ :\ \mathsf{List}\ A\,]\ \mathit{length}\ \mathit{xs} \equiv n$$

computes the underlying list of a vector and a proof that the list has length n, and the right-to-left direction

$$\mathsf{Iso}.\mathit{from}\ (\mathit{Vec\text{-}iso}\ A\ n)\ :\ (\Sigma[\,\mathit{xs}\ :\ \mathsf{List}\ A\,]\ \mathit{length}\ \mathit{xs} \equiv n) \to \mathsf{Vec}\ A\ n$$

promotes a list to a vector when there is a proof that the list has length n. (The definition of $\_\cong\_$, which is actually an instance of a record datatype Iso, appears in **??**.) To get insertion on internalist vectors, we convert the input vector to its externalist representation, make insert_{EV} do the work, and convert the result back to the internalist representation; more formally, the operation

$$\mathit{insert}_V\ :\ \mathit{Val} \to \{n\ :\ \mathsf{Nat}\} \to \mathsf{Vec}\ \mathit{Val}\ n \to \mathsf{Vec}\ \mathit{Val}\ (\mathsf{suc}\ n)$$

is defined by the commutative diagram:

$$\begin{tikzcd}
\mathsf{Vec}\ \mathit{Val}\ n \arrow[rr, "\mathit{insert}_V\ y"] \arrow[d, "\mathsf{Iso}.\mathit{to}\ (\mathit{Vec\text{-}iso}\ \mathit{Val}\ n)"'] & & \mathsf{Vec}\ \mathit{Val}\ (\mathsf{suc}\ n) \\
\begin{array}{c}\Sigma[\,\mathit{xs} : \mathsf{List}\ \mathit{Val}\,] \\ \mathit{length}\ \mathit{xs} \equiv n\end{array} \arrow[rr, "\mathit{insert}\ y\ *", "\mathit{insert\text{-}length}\ y\ \_"'] & & \begin{array}{c}\Sigma[\,\mathit{xs} : \mathsf{List}\ \mathit{Val}\,] \\ \mathit{length}\ \mathit{xs} \equiv \mathsf{suc}\ n\end{array} \arrow[u, "\mathsf{Iso}.\mathit{from}\ (\mathit{Vec\text{-}iso}\ \mathit{Val}\ (\mathsf{suc}\ n))"']
\end{tikzcd}$$

(The $\_*\_$ operator is defined by $(f\ *\ g)\ (x\ ,\ y)\ =\ (f\ x\ ,\ g\ y)$, and the underscore leaves out the term for AGDA to infer.) Similarly, we can get insertion for internalist ordered lists and ordered vectors (definitions to appear in **??**) from the externalist library by suitable conversion isomorphisms of the same form as *Vec-iso*. It is due to these conversion isomorphisms between internalist and externalist representations that we can **analyse** internalist datatypes into externalist components, which can then be modularly processed. This analysis of internalist datatypes and its application to modular library structuring is explored in **??** (in particular, the insertion example is resolved in **??**).

The interconnection between internalism and externalism (in the form of conversion isomorphisms) also shed some light on supporting internalist type design. The **synthetic** direction of the interconnection goes from basic types and predicates to internalist types. It is conceivable that, for externalist predicates of some particular form, we can manufacture corresponding internalist types on the other side of the interconnection. The externalist side of the interconnection is usually kept non-dependently typed, so it is possible to use existing non-dependently typed calculi to derive suitable externalist predicates from specifications, which are then used to manufacture datatypes on the internalist side for type-directed programming. **??** presents one such approach, using relational calculus [Bird and de Moor, 1997] as a design language for internalist datatypes. Rather than improvising internalist types and hoping that they will work, we write specifications in the form of relational programs, which are amenable to algebraic transformation and can be much more concise and readable than the language of datatype declarations, making it easier to arrive at helpful and comprehensible internalist types.

To sum up: While internalism offers type-directed program construction and reduces the burden of producing correctness proofs, additional or more complicated properties of existing programs can only be established by externalism, which naturally gives rise to a modular organisation. Both internalism and externalism are here to stay, since the two styles serve different purposes and have their own advantages and disadvantages. Exploiting their interconnection can lead to interesting programming patterns, which the rest of this dissertation explores.

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. doi: 10.1007/978-0-387-35672-3_1. ↰ page 18

Thorsten ALTENKIRCH, Conor MCBRIDE, and Wouter SWIERSTRA [2007]. Observational equality, now! In *Programming Languages meets Program Verification*, PLPV'07, pages 57–68. ACM. doi: 10.1145/1292597.1292608. ↰ page 17

Gilles BARTHE, Venanzio CAPRETTA, and Olivier PONS [2003]. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293. doi: 10.1017/S0956796802004501. ↰ page 18

Yves BERTOT and Pierre CASTÉRAN [2004]. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag. ISBN: 978-3540208549. ↰ page 28

Richard BIRD and Oege DE MOOR [1997]. *Algebra of Programming*. Prentice-Hall. ISBN: 978-0135072455. ↰ page 35

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. doi: 10.1007/978-3-540-24849-1_8. ↰ page 20

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 18

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith [1985]. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall. ISBN: 978-1468059106. ↰ page 16

Thierry Coquand and Gérard Huet [1988]. The Calculus of Constructions. *Information and Computation*, 76(2–3):95–120. doi: 10.1016/0890-5401(88)90005-3. ↰ page 28

Thierry Coquand and Christine Paulin-Mohring [1990]. Inductively defined types. In *International Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag. doi: 10.1007/3-540-52335-9_47. ↰ page 28

Pierre-Évariste Dagand and Conor McBride [2012a]. Elaborating inductive definitions. arXiv:1210.6390. ↰ page 25

Pierre-Évariste Dagand and Conor McBride [2012b]. Transporting functions across ornaments. In *International Conference on Functional Programming*, ICFP'12, pages 103–114. ACM. doi: 10.1145/2364527.2364544. ↰ pages 18 and 19

Edsger W. Dijkstra [1976]. *A Discipline of Programming*. Prentice-Hall. ISBN: 978-0132158718. ↰ page 29

Michael Dummett [2000]. *Elements of Intuitionism*. Oxford University Press, second edition. ISBN: 978-0198505242. ↰ pages 2 and 13

Peter Dybjer [1994]. Inductive families. *Formal Aspects of Computing*, 6(4):440–465. doi: 10.1007/BF01211308. ↰ page 6

Peter Dybjer [1998]. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549. doi: 10.2307/2586554. ↰ page 22

Jeremy Gibbons [2007]. Datatype-generic programming. In *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 1–71. Springer-Verlag. doi: `10.1007/978-3-540-76786-2_1`. ↰ page 18

Healfdene Goguen, Conor McBride, and James McKinna [2006]. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*, volume 4060 of *Lecture Notes in Computer Science*, pages 521–540. Springer-Verlag. doi: `10.1007/11780274_27`. ↰ page 7

Arend Heyting [1971]. *Intuitionism: An Introduction*. Amsterdam: North-Holland Publishing, third revised edition. ISBN: `978-0720422399`. ↰ page 2

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler [2007]. A history of Haskell: Being lazy with class. In *History of Programming Languages*, HOPL-III, pages 1–55. ACM. doi: `10.1145/1238844.1238856`. ↰ page 6

Xavier Leroy [2009]. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446. doi: `10.1007/s10817-009-9155-4`. ↰ page 29

Pierre Letouzey [2003]. A new extraction for Coq. In *Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer-Verlag. doi: `10.1007/3-540-39185-1_12`. ↰ page 29

Zhaohui Luo [1994]. *Computation and Reasoning: A Type Theory for Computer Science*. Clarendon Press. ISBN: `978-0198538356`. ↰ pages 14 and 28

Per Martin-Löf [1984a]. Constructive mathematics and computer programming. *Philosophical Transactions of the Royal Society of London*, 312(1522):501–518. doi: `10.1098/rsta.1984.0073`. ↰ page 5

Per Martin-Löf [1984b]. *Intuitionistic Type Theory*. Bibliopolis, Napoli. ↰ pages 18 and 22

Per Martin-Löf [1987]. Truth of a proposition, evidence of a judgement, validity of a proof. *Synthese*, 73(3):407–420. doi: `10.1007/BF00484985`. ↰ page 3

Conor McBride [1999]. *Dependently Typed Functional Programs and their Proofs*. Ph.D. thesis, University of Edinburgh. ↰ page 7

Conor McBride [2004]. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer-Verlag. doi: `10.1007/11546382_3`. ↰ pages 7 and 30

Conor McBride [2011]. Ornamental algebras, algebraic ornaments. URL: `https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/LitOrn.pdf`. ↰ page 26

Conor McBride [2012]. A polynomial testing principle. URL: `https://personal.cis.strath.ac.uk/conor.mcbride/PolyTest.pdf`. ↰ page 31

Conor McBride and James McKinna [2004]. The view from the left. *Journal of Functional Programming*, 14(1):69–111. doi: `10.1017/S0956796803004829`. ↰ pages 7, 10, 11, and 12

Lambert Meertens [1992]. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424. doi: `10.1007/BF01211391`. ↰ page 6

Erik Meijer, Maarten Fokkinga, and Ross Paterson [1991]. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 124–144. Springer-Verlag. doi: `10.1007/3540543961_7`. ↰ page 6

Peter Morris [2007]. *Constructing Universes for Generic Programming*. Ph.D. thesis, University of Nottingham. ↰ page 27

Bengt Nordström, Kent Peterson, and Jan M. Smith [1990]. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press. ISBN: `978-0198538141`. ↰ pages 16, 18, and 28

Ulf Norell [2007]. *Towards a Practical Programming Language based on Dependent Type Theory*. Ph.D. thesis, Chalmers University of Technology. ↰ page 11

Christine Paulin-Mohring [1989]. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Principles of Programming Languages*, pages 89–104. ACM. doi: `10.1145/75277.75285`. ↰ page 29

Simon Peyton Jones [1997]. A new view of guards. URL: `http://research.microsoft.com/en-us/um/people/simonpj/Haskell/guards.html`. ↰ page 10

Tim Sheard and Nathan Linger [2007]. Programming in Ωmega. In *Central-European Functional Programming School*, volume 5161 of *Lecture Notes in Computer Science*, pages 158–227. Springer-Verlag. doi: `10.1007/978-3-540-88059-2_5`. ↰ page 16

Jan M. Smith [1988]. The independence of Peano's fourth axiom from Martin-Löf's type theory without universes. *Journal of Symbolic Logic*, 53(3):840–845. doi: `10.2307/2274575`. ↰ page 18

Thomas Streicher [1993]. Investigations into intensional type theory. Habilitation thesis, Ludwig Maximilian Universität. ↰ pages 7 and 14

The Univalent Foundations Program [2013]. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, Princeton. URL: `http://homotopytypetheory.org/book/`. ↰ page 17

Philip Wadler [1987]. Views: A way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages*, pages 307–313. ACM. doi: `10.1145/41625.41653`. ↰ page 11

Todo list