Chapter 2

From intuitionistic type theory to dependently typed programming

specific issues regarding practical programming with type theory (including introduction to Agda); lead into internalism vs externalism

We start with an introduction to Martin-Löf's intuitionistic type theory [Martin-Löf, 1975, 1984b; Nordström et al., 1990] 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-correctness-by-construction movement. It has thus departed from the mathematical traditions considerably, and deviations can be found from syntactic presentations to the underlying philosophy.

In principle, everything can be translated down to type theory.

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 for forming the **set** of natural numbers in Martin-Löf Type Theory. In symbols, we are justified by our intuition to have the **formation rule**

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

which says 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 elements** of Nat, i.e., those elements that are immediately recognisable as belonging to Nat, namely zero and suc n whenever n is an element 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 element. We accept them as **non-canonical elements** 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 elements, either directly or indirectly, as bearing a certain form and thus belonging to the set, so the elements of the set are intuitively clear to us as a certain type of mental constructions.

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 if and only if 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 (element) of $A \wedge B$ is a pair of a proof (element) of A and a proof (element) of B. This is the intuitive (canonical) way we admit as proving a conjunction, and 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 elements — 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 elements. This is called the **propositions-as-types principle**, which reflects the observation that proofs are nothing but a certain kind of mental constructions.

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 elements, it is possible to express an effective method that mimics the construction of an element by saying that the computation has the same structure as how the element is constructed. 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 elements of Nat. (Since we only aim to present a casual sketch of type theory, we take the liberty of using Agda functions in places where terms under contexts should have been used.) If we have an element z of $P\ \mathsf{zero}$ and a method s that, for any $n : \mathsf{Nat}$, transforms an element of $P\ n$ to an element of $P\ (\mathsf{suc}\ n)$, then we can compute an element 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 symbol Nat-elim symbolises the method described above, which, given P, z, and s, transforms every natural number n into something of type $P\ n$. The actual computation performed by Nat-elim is stated as two **computation rules** in the form of equality judgements (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 $n : \mathsf{Nat}$ if the base case z and the inductive case s can be proved. 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.

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 which is then used to talk 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 they are used to talk about. Programming systems based on such principle (e.g., Hoare logic [Hoare, 1969]) then need to have a meta-level logic language to reason about properties of programs. In languages based on type theory, however, the two traditional levels are coherently integrated into one, so programs can be naturally constructed along with their correctness proofs. 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 result of type B is guaranteed to be related to the input of type A by R. 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 is introduced in Section 2.5, which opens the central topic studied by this thesis. Before that, we make a transition from type theory to practical programming in Agda by discussing

the relationship between elimination and pattern matching (Section 2.2) and our position on equality (Section 2.3). We also introduce the notion of universes and construct a universe of index-first datatypes (Section 2.4) which is used throughout this thesis.

2.2 Elimination vs pattern matching

The formation rule and the introduction rules for a set directly translate into an algebraic datatype declaration in functional languages. For example, the type of natural numbers is translated into Agda as

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

Having a datatype, naturally we wish to write programs on that datatype. In functional programming, the pattern matching syntax is widely used for defining programs. It is key to the clarity of functional programs because it not only allows a function to be intuitively defined by several equations but also clearly conveys the strategy of splitting a problem into subproblems by case analysis. On the other hand, computations in type theory are specified using eliminators. Besides keeping the basic theory simple, one reason is that programs in type theory are demanded to be total, for a program must terminate if it is intended as a proof, and using eliminators enforces totality. Pattern matching and elimination are basically equivalent in expressive power, as eliminators can be easily defined by dependent pattern matching, and conversely dependent pattern matching can be reduced to elimination if **uniqueness of identity proofs** — also known as the **K axiom** [Streicher, 1993] — is assumed [Goguen et al., 2006]. Nevertheless, the use of pattern matching together with an interactive development environment is more informative and helpful in dependently typed languages than in simply typed ones, because splitting a problem into subproblems by case analysis in dependently typed programming often leads to nontrivial refinement of the goal type and even the context.

To illustrate, let us look at an example of interactive development in Agda, whose design was inspired by McBride and McKinna [2004]. Consider the following inductively defined less-than-or-equal-to binary relation on natural numbers.

```
data _⩽_ (m : Nat) : Nat → Set where
  refl : m ⩽ m
  step : (n : Nat) → m ⩽ n → m ⩽ suc n
```

Suppose we are asked to prove that $\_\leqslant\_$ is transitive, i.e., the term

```
trans : (x y z : Nat) → x ⩽ y → y ⩽ z → x ⩽ z
```

can be constructed. We define *trans* interactively by first putting pattern variables for the arguments on the left of its defining equation and leaving an "interaction point" on the right. Agda then tells us a term of type $x \leqslant z$ is expected.

```
trans : (x y z : Nat) → x ⩽ y → y ⩽ z → x ⩽ z
trans x y z p q = {x ⩽ z }0
```

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

```
trans : (x y z : Nat) → x ⩽ y → y ⩽ z → x ⩽ z
trans x .z z       p refl      = {x ⩽ z  ‖ p : x ⩽ z }1
trans x y .(suc w) p (step w r) = {x ⩽ suc w }2
```

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, on enquiry, Agda tells us that the type of p in the context is now $x \leqslant z$ (which was originally $x \leqslant y$). Similarly for Goal 2, z is unified with suc w and the goal type is rewritten accordingly. We see that the case analysis has led to two subproblems 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.

```
trans : (x y z : Nat) → x ⩽ y → y ⩽ z → x ⩽ z
```

$$
\begin{array}{lll}
\mathit{trans}\ x\ .z\ z & p\ \mathsf{refl} & = p \\
\mathit{trans}\ x\ y\ .(\mathsf{suc}\ w)\ p\ (\mathit{step}\ w\ r) & = & \{\, x \leqslant \mathsf{suc}\ w \,\}_2
\end{array}
$$

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

$$
\begin{array}{lll}
\mathit{trans} : (x\ y\ z : \mathsf{Nat}) \to x \leqslant y \to y \leqslant z \to x \leqslant z & & \\
\mathit{trans}\ x\ .z\ z & p\ \mathsf{refl} & = p \\
\mathit{trans}\ x\ y\ .(\mathsf{suc}\ w)\ p\ (\mathit{step}\ w\ r) & = & \mathit{step}\ w\ \{\, x \leqslant w \,\}_3
\end{array}
$$

Now we see that the induction hypothesis term $\mathit{trans}\ x\ y\ w\ p\ r : x \leqslant w$ (note that r is a sub-term of $\mathit{step}\ w\ r$) has the right type. Filling the term into Goal 3 completes the program.

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

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

$$
\begin{array}{l}
\leqslant\text{-elim} : (m : \mathsf{Nat})\ (P : (n : \mathsf{Nat}) \to m \leqslant n \to \mathsf{Set}) \to \\
\qquad ((t : m \leqslant m) \to P\ m\ t) \to \\
\qquad ((n : \mathsf{Nat})\ (t : m \leqslant n) \to P\ n\ t \to P\ (\mathsf{suc}\ n)\ (\mathit{step}\ n\ t)) \to \\
\qquad (n : \mathsf{Nat})\ (t : m \leqslant n) \to P\ n\ t
\end{array}
$$

and write

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

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. Comparing the two versions, we see that elimination

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.

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 desirable. McBride and McKinna [2004] thus proposed *with*-**matching**, which generalises pattern guards and in effect shifts pattern matching on intermediate computation from the right of an equation to the left, sitting along with the arguments. For a plain example:

$$
\begin{aligned}
&\mathit{insert} \;:\; \mathsf{Nat} \to \mathsf{List}\;\mathsf{Nat} \to \mathsf{List}\;\mathsf{Nat}\\
&\mathit{insert}\; y\; [\,] \;=\; y :: [\,]\\
&\mathit{insert}\; y\; (x :: xs)\; \mathbf{with}\; y \leqslant_? x\\
&\mathit{insert}\; y\; (x :: xs) \;\mid\; \mathit{true} \;=\; y :: x :: xs\\
&\mathit{insert}\; y\; (x :: xs) \;\mid\; \mathit{false} \;=\; x :: \mathit{insert}\; y\; xs
\end{aligned}
$$

This is essentially no different from a normal case expression, except that using **with** renders the result of $y \leqslant_? x$ as an additional argument in the context, which is then immediately matched with *true* or *false*. In this case, the original context — y, x, and xs — is not affected by the pattern matching, but in more interesting cases it can be. For example, Wadler's views [1987] can be adapted to dependently typed programming in a more accurate manner, which are supported by **with** in Agda. Suppose we wish to implement a snoc-list view for cons-lists. We define the following view type

$$
\begin{aligned}
&\mathbf{data}\; \mathit{SnocView}\; \{A \;:\; \mathsf{Set}\} \;:\; \mathsf{List}\; A \to \mathsf{Set}\; \mathbf{where}\\
&\quad \mathsf{nil} \;\;\; : \mathit{SnocView}\; [\,]\\
&\quad \mathsf{snoc} \;:\; (xs \;:\; \mathsf{List}\; A)\; (x \;:\; A) \to \mathit{SnocView}\; (xs +\!\!+ (x :: [\,]))
\end{aligned}
$$

intending to say that a list is either empty or has the form $xs +\!\!+ (x :: [\,])$, which

is proved by the following covering function (whose accuracy is not possible in languages with simpler type disciplines):

```
snocView : {A : Set} → (xs : List A) → SnocView xs
snocView []                  = nil
snocView (x :: xs)           with snocView xs
snocView (x :: .[])          | nil          = snoc [] x
snocView (x :: .(ys ++ (y :: []))) | snoc ys y = snoc (x :: ys) y
```

Then, for example, the function *init* which removes the last element (if any) can be implemented simply as

```
init : {A : Set} → List A → List A
init xs                  with snocView xs
init .[]                 | nil          = []
init .(ys ++ (y :: [])) | snoc ys y = ys
```

We see that, in both *snocView* and *init*, performing pattern matching on the result of *snocView xs* refines *xs* in the context to either $[]$ or $ys +\!\!+ (y :: [])$ in the two cases. The refined context can be shown explicitly for each case because the matching on *snocView xs* is moved to the left, which is the same difference between using pattern matching and using eliminators. Hence **with**-matching is preferred to traditional case expressions for the same reason that pattern matching is preferred to eliminators: The former clearly expresses context/goal refinements in subproblems in an equational style that is easy to follow, especially when supported by an interactive development environment.

McBride and McKinna [2004] described how programs using pattern matching can be translated into eliminator-based programs. They in fact proposed a general mechanism for invoking any programmer-defined eliminator using the pattern matching syntax, so programmers can choose whichever problem-splitting strategy they need and express that with pattern matching. For example, the standard eliminator for Nat says that to solve a programming problem $P\ n$ for any n : Nat, it is sufficient to solve the more specialised subproblems P zero and P (suc n) (assuming an answer of $P\ n$). This is not the only way to cover all natural numbers, of course; for example, we might split

the problem into the two subproblems $P\ i$ where $i < k$ and $P\ (j + k)$ where $j : \mathsf{Nat}$, for some fixed k. We should be able to match a natural number against such nonstandard patterns if that is the strategy we use to divide and solve the problem. Problem specifications can be made more precise by using dependent types, but the solutions would have to be equally precise as a result. Reintroducing pattern matching into dependently typed languages is one step towards helping programmers to describe such solutions naturally and clearly.

2.3 Equality

In logic, the **intension** of a concept is its internal, defining content, while the **extension** of the concept is the range of objects it refers to. In mathematics, for example, the intension of the set $S = \{\, x \mid x \in Nat \text{ is even} \,\}$ is the description that the elements are even natural numbers, and the extension of the set is the enumeration 0, 2, 4, 6, 8, ... Different intensions may nevertheless lead to the same extension, for example $T = \{\, x - 1 \mid x \in Nat \text{ is odd} \,\}$ is intensionally different from S, but they have the same extension. In other words, S and T use different ways to describe the same range of objects. The axiom of extensionality in set theory defines set equality to be the extensional one, so we consider S and T to be the same set because the extension of S and T are the same, even though they have different intensions. In intuitionistic mathematics, however, the default, fundamental equality is intensional. The reason is that objects in intuitionistic mathematics are given to us as mental **constructions**. For example, the construction of S is to find all the even natural numbers, while the construction of T is to find all the odd natural numbers and subtract 1 from each of them. The two constructions, i.e., descriptions, are different. We can still talk about extensional equality if needed, but that requires a separate definition, which can be a complex proposition in general. For sets, the definition would be $\forall\, x.\ x \in S \Leftrightarrow x \in T$, i.e., a bi-implication, and we can prove that two sets are extensionally equal in intuitionistic mathematics by proving the bi-implication as we do in classical mathematics. The difference is that in clas-

Luo [1994]

sical mathematics we talk exclusively about extensions and deliberately ignore intensions, so for example "set equality" always refers to the extensional one, whereas in intuitionistic mathematics intensions are also given emphasis. In other words, whereas in both intuitionistic and classical mathematics one can talk about extensionality, an intensional layer about syntactic descriptions of objects is present in intuitionistic mathematics, which is transparent in classical mathematics.

The fundamental equality is formulated as **judgemental equality** in type theory. For intuitionistic mathematics it is the intensional, syntactic equality, also known as **definitional equality**, whereas for classical mathematics it is extensional equality. A characteristic feature of judgemental equality is that it is fully substitutive: judgementally equal terms can be freely substituted for one another. So after we prove that two sets are extensionally equal in classical mathematics, we can simply substitute one for the other because they are judgementally equal in the classical, extensional setting. Judgemental equality cannot be expressed as propositions or have proofs inside the theory, though, since it is a meta-theoretical concept, which, for example, is used in type checking in a language implementation and hence is not an entity in the language. To state equality between two objects as a proposition and have proof for that proposition inside the theory, we need **propositional equality**, which can be defined by the following inductive family.

$$\begin{array}{l}\textbf{data}\ \_\equiv\_\ \{A\ :\ \mathsf{Set}\}\ (x\ :\ A)\ :\ A \to \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{refl}\ :\ x\ \equiv\ x\end{array}$$

The canonical way to prove an equality proposition $x \equiv y$ is refl, which is permitted when x and y are judgementally equal. In general, however, computation may be required to prove an equality proposition. For example, the following "catamorphic" identity function on natural numbers

$$\begin{array}{lcl} id'\ :\ \mathsf{Nat} \to \mathsf{Nat} & & \\ id'\ \mathsf{zero} & = & \mathsf{zero} \\ id'\ (\mathsf{suc}\ n) & = & \mathsf{suc}\ (id'\ n) \end{array}$$

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

$$id\ :\ \{A\ :\ \mathsf{Set}\} \to A \to A$$
$$id\ x\ =\ x$$

by proving the proposition

$$(n\ :\ \mathsf{Nat}) \to id\ n\ \equiv\ id'\ n$$

whose proof is by induction on n and thus requires computation. It might be said that propositional equality is "delayed" judgemental equality in propositional form: The terms $id\ n$ (which is definitionally just n) and $id'\ n$ are not judgementally equal, but they will compute to the same canonical term (and hence become judgementally equal) after substituting a concrete natural number for n, allowing the computation to complete. Streicher [1993, page 19] suggested that we "consider the identity type $[t\ \equiv\ s]$ as a proposition stating a relation between the **objects denoted by the terms** t and s, respectively, whereas the judgement $t\ =\ s\ \in\ A$ is a statement of a relation between the **terms** t and s[.]" Indeed, in an intensional setting, if we regard canonical terms to be the semantic objects denoted by terms, then it might be said that two terms are judgementally equal if their normal forms are syntactically identical, while two terms are propositionally equal if they can be proved to compute to the same canonical term after instantiating the context to canonical terms, i.e., they denote the same semantic object. Practically, when used for substitution, a proof of an equality proposition needs to be eliminated by applying the following standard eliminator commonly called J.

$$J\ :\ \{A\ :\ \mathsf{Set}\}\ \{x\ :\ A\}\ (P\ :\ (y\ :\ A) \to x\ \equiv\ y \to \mathsf{Set}) \to$$
$$\quad P\ x\ \mathsf{refl} \to \{y\ :\ A\} \to (eq\ :\ x\ \equiv\ y) \to P\ y\ eq$$

A more convenient substitution operator can be defined in terms of J.

$$subst\ :\ \{A\ :\ \mathsf{Set}\}\ (T\ :\ A \to \mathsf{Set}) \to \{x\ y\ :\ A\} \to x\ \equiv\ y \to T\ x \to T\ y$$
$$subst\ T\ eq\ t\ =\ J\ (\lambda z\ \mapsto\ \_)\ T\ z)\ t\ eq$$

It is like type-casting in programming languages and serves as an explicit proof inside the theory that y can be regarded as x. On the other hand, judgemental equality identifies terms at a more fundamental level and allows a term to be directly substituted for any other term identified with it, without need of any justification inside the theory.

The type of refl says that judgementally equal terms are propositionally equal, so judgemental equality is embedded into propositional equality. If we add the converse **equality reflection rule**

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

to the theory, injecting propositional equality back into judgemental equality, then the resulting type theory is called **extensional**. Type theory without the equality reflection rule is called **intensional**, indicating that its judgemental equality is syntactic. Extensional type theory gets the name because merely syntactic comparison no longer suffices to determine whether two terms are judgementally equal; extensional reasoning may be involved. For example, extensionally equal functions become judgementally equal in extensional type theory: Suppose 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

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

In general, however, f and g may have very different intensions, so adopting the equality reflection rule makes judgemental equality extensional. The intensional layer present in intuitionistic mathematics thus collapses: The fundamental equality is extensional equality as in classical mathematics, so there is no longer a separate notion of intensional equality. Having extensional equality as judgemental equality makes extensional reasoning much easier because no justification is needed for substitution of extensionally equal terms inside the theory. This is the norm in classical mathematics, where extensionality dominates. For example, in category theory, a universal function (i.e., a universal arrow in the category of sets and total functions) is unique **up to exten-**

sional equality, and category theorists substitute functions satisfying the same universal property for one another all the time. For a language implementation, this means that the programmer can do more than syntactic substitutions without need of explicitly specifying type casts and what equality proofs to use. For example, to show that *id* is extensionally equal to *id′*, we would write the following:

$$
\begin{aligned}
&\mathit{ideq} \;:\; (n \;:\; \mathsf{Nat}) \to \mathit{id}\; n \;\equiv\; \mathit{id}'\; n \\
&\mathit{ideq}\; \mathsf{zero} \quad\;\; = \; \mathsf{refl} \\
&\mathit{ideq}\; (\mathsf{suc}\; n) \;=\; \{\, \mathsf{suc}\; n \;\equiv\; \mathsf{suc}\; (\mathit{id}'\; n) \,\}
\end{aligned}
$$

How the proof is completed depends on whether we are working intensionally or extensionally. If we are working intensionally, the hole needs to be filled with the term

$$\mathit{cong}\; \mathsf{suc}\; (\mathit{ideq}\; n) \;:\; \mathsf{suc}\; n \;\equiv\; \mathsf{suc}\; (\mathit{id}'\; n)$$

where

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

That is, we need to indicate explicitly that we are using an inductively computed result $\mathit{ideq}\; n \;:\; n \;\equiv\; \mathit{id\_Nat}\; n$, which needs to be further modified by *cong* suc to match the goal type. On the other hand, if we are working extensionally, a simple refl suffices! The typechecker is told by our placement of refl that suc n and suc $(\mathit{id}'\; n)$ are actually judgementally equal, and has to somehow figure out that there is a term that has type $n \;\equiv\; \mathit{id}'\; n$, so n and $\mathit{id}'\; n$ are judgementally equal by equality reflection, and thus suc n and suc $(\mathit{id}'\; n)$ are indeed judgementally equal by congruence. This example illustrates that type checking in an extensional setting cannot simply resort to syntactic equality of normal forms but needs to search for arbitrary equality proofs. The typechecker can ask for hints from the programmer, like in Sheard's Ωmega language [Sheard and Linger, 2007], but type checking becomes undecidable in general. Another perspective to look at this problem is that the rewriting system underlying judgemental equality loses confluence, since different normal forms may be equated due to the equality reflection rule. Consequently, checking syntactic equality of normal forms is no longer a sound way to do

type checking. Losing confluence also means that the computational meaning is disrupted since term reduction becomes nondeterministic. Therefore, the reasoning power offered by extensional type theory may be tempting, but to preserve good computational behaviour we have to stick to intensional type theory, namely giving up the equality reflection rule and keeping judgemental equality intensional.

heterogeneous equality, Altenkirch et al. [2007], The Univalent Foundations Program [2013]

2.4 Datatypes and universes

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, in which a set of codes for datatypes is called a **universe** (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, 2012b], 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

2.4.1 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 $\mathsf{suc}\ n$ — 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 $\mathsf{suc}\ n$ 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 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
```

$$\begin{aligned}&\mathsf{Vec}\ A\ \mathsf{zero} &&\ni\ [\,]\\ &\mathsf{Vec}\ A\ (\mathsf{suc}\ n) &&\ni\ \_::\_\ (a\ :\ A)\ (as\ :\ \mathsf{Vec}\ A\ n)\end{aligned}$$

$\mathsf{Vec}\ A$ 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:

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

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. □

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

TBC

□

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 $A : \mathsf{Set}$)

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

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:

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

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

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

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

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

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 $\sigma[\,s : S\,]\ D\ s$ 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 $\mathsf{Desc}\ \top$.

$$\textbf{data}\ \mathsf{ListTag} : \mathsf{Set}\ \textbf{where}\ \text{`nil}\ \text{`cons} : \mathsf{ListTag}$$

$$\begin{aligned}
&NatD : \mathsf{Desc}\ \top\\
&NatD\ \blacksquare\ =\ \sigma\ \mathsf{ListTag}\ \lambda\ \{\ \text{`nil}\ \ \ \mapsto\ \mathsf{v}\ [\,]\\
&\qquad\qquad\qquad\qquad ;\ \text{`cons}\ \mapsto\ \mathsf{v}\ (\blacksquare :: [\,])\ \}
\end{aligned}$$

The index request is necessarily $\blacksquare$, and we respond with a field of type $\mathsf{ListTag}$

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 (▪ :: []). □

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.

$$
\begin{aligned}
&\mathit{ListD} \;:\; \mathsf{Set} \to \mathsf{Desc}\ \top \\
&\mathit{ListD}\ A\ \blacksquare \;=\; \sigma\ \mathsf{ListTag}\ \lambda\ \{\ \text{‘}\mathsf{nil} \;\mapsto\; \mathsf{v}\ [] \\
&\qquad\qquad\qquad\qquad ;\ \text{‘}\mathsf{cons} \;\mapsto\; \sigma[\ \_ : A\ ]\ \mathsf{v}\ (\blacksquare :: [])\ \}
\end{aligned}
$$

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 (non-trivially) indexed by Nat, so the declaration of vectors corresponds to

$$
\begin{aligned}
&\mathit{VecD} \;:\; \mathsf{Set} \to \mathsf{Desc}\ \mathsf{Nat} \\
&\mathit{VecD}\ A\ \mathsf{zero} \;=\; \mathsf{v}\ [] \\
&\mathit{VecD}\ A\ (\mathsf{suc}\ n) \;=\; \sigma[\ \_ : A\ ]\ \mathsf{v}\ (n :: [])
\end{aligned}
$$

which is directly comparable to the index-first base functor *VecF′* at the beginning of Section 2.4.2. □

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

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

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

$$
\begin{aligned}
&\mathsf{List} \;:\; \mathsf{Set} \to \mathsf{Set} \\
&\mathsf{List}\ A \;=\; \mu\ (\mathit{ListD}\ A)\ \blacksquare
\end{aligned}
$$

```
mutual
  fold : {I : Set} {D : Desc I} {X : I → Set} → (𝕀 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} → (𝕀 D X ⇉ X) → ⟦ D′ ⟧ (μ D) → ⟦ D′ ⟧ X
  mapFold D (ν [])       f ▪        = ▪
  mapFold D (ν (i :: is)) f (d , ds) = fold f d , mapFold D (ν 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.

```
[] : {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 (assuming that the terms [] and _::_ can be used in pattern matching):

```
_++_ : List A → List A → List A
[]       ++ bs = bs
(a :: as) ++ bs = a :: (as ++ bs)
```

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 2.4.1. 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 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:

$$\mathit{fold} \;:\; \{I \;:\; \mathsf{Set}\}\; \{D \;:\; \mathsf{Desc}\; I\}\; \{X \;:\; I \to \mathsf{Set}\} \to (\mathbb{F}\; D\; X \rightrightarrows X) \to (\mu\; D \rightrightarrows X)$$

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

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

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. □

2.5 Internalism and externalism

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 the **dependent pair** type.

$$\begin{aligned}&\textbf{data}\ \Sigma\ (A\ :\ \mathsf{Set})\ (B\ :\ A \to \mathsf{Set})\ :\ \mathsf{Set}\ \textbf{where}\\ &\quad \_,\_\ :\ (x\ :\ A) \to B\ x \to \Sigma\ A\ B\end{aligned}$$

When A is interpreted as a ground set and B as a predicate on A, an element of $\Sigma\ A\ B$ is an element x of A paired with a proof that $B\ x$ holds. For example, lists of elements of type A with a certain length n are specified by

$$\Sigma\ (\mathsf{List}\ A)\ (\lambda\, xs \mapsto \mathit{length}\ xs \equiv n)$$

where $\mathit{length}\ :\ \{A\ :\ \mathsf{Set}\} \to \mathsf{List}\ A \to \mathsf{Nat}$ computes the length of a list. This Σ-type can be naturally read as "the lists xs such that the length of xs is n", bearing some similarity to the notation of set comprehension. Besides being deeply rooted in mathematical traditions, in practice this approach offers very good composability: Whenever a new property is needed, the programmer simply defines a new predicate and uses a Σ-type to impose that predicate on an existing datatype. Predicates easily compose by pointwise conjunction, so objects with two or more properties can be conveniently specified. When programs are the objects we reason about, this style naturally suggests a logical 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. Consequently, proofs may be erased as they are irrelevant to the computational behaviour of programs. 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 subset theory was described 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] is also designed to support this proving-after-programming style, which is also famous for supporting program extraction from proof scripts [Paulin-Mohring, 1989].

On the other hand, proponents of dependently typed programming believe that, instead of regarding dependent types as yet another type system we impose on existing programs, we should rethink about what programs can be written in a dependently typed language. One such reconsideration is the movement of using inductive families directly for representing data with constraints. The classic example is vectors, which are lists indexed by their length.

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

A simple inductive argument shows that a vector of type Vec A n must be of length n. This fact holds for a vector **by construction**, in contrast to the previous approach using Σ-types, where the length statement is made about a plain list already constructed.

use the new version of explanations of the names internalism and externalism

To illustrate why it can be beneficial to switch from externalism to internalism, suppose we wish to extract the head element from a nonempty list.

draw from the papers

Perhaps not surprisingly, internalist reasoning is rarely seen in mathematics. Here are two possible explanations. The first, philosophical one is that the platonist character of classical mathematics, i.e., the presupposition that mathematical objects are independently existing entities, naturally leads to externalism. The mathematical objects exist *a priori*, and then our proofs are written about them. There is thus a clear "phase distinction," which makes it strange to mix proofs with objects. The second, practical explanation (which also works for non-platonist mathematics) is that it is hard to justify the correctness of an internalist program in prose without silently converting the internalist program to an externalist one. For example, we would say "a vector of length n" and go on about how its length relates to the result, etc., which is not so different from saying "a **list** of length n" and so on — we still need to talk

and reason about the constraints separately, unlike how we write an internalist program, which only manipulates data. It might be said that the correctness proof of an internalist program is more syntactic in nature, which in general is more suitable for being checked by machines, while mathematical writing aims to describe the intuition behind so human readers can get the high-level ideas. Even if correctness is implied by the syntactic structure, it is still desirable to have an intuitive explanation of why it is so in prose. Therefore, as the same degree of explanation is needed no matter whether constraints are integrated into syntax or not, it is reasonable to just keep the syntax simple, refraining from using internalist types in mathematical writing.

Type theory makes it possible to mix programs with logic, and thus allows us to bind and manipulate data and proofs together, which is quite different from the programming styles we are used to. We do not know how much potential internalism has, but as its possibility remains to be explored, it seems premature to stick to the traditional, externalist approach that strictly separates programming from logic. Another supporting example is that optimisation of dependently typed programs may exploit value dependencies in types and eliminate a substantial portion of code [Brady et al., 2004] — the length of a vector does not need to be actually stored, *vhead* can just assume that the input vector starts with a cons node, etc. Program optimisation thus does not necessarily take the form of program extraction, which is based on the distinction between programs and proofs. As McBride [2004] said, "[t]here is a tendency to see programming as a fixed notion, essentially untyped. In this view, we make sense of and organise programs by assigning types to them, the way a biologist classifies species, and in order to classify more the exotic creatures, like *printf* or the *zipWith* family, one requires more exotic types. This conception fails to engage with the full potential of types to make a positive contribution to program construction". To support this belief, he presented a development of the first-order unification algorithm, which has long been described using general recursion and required separate termination and correctness proofs, as a structurally recursive, dependently typed program which is correct by construction [McBride, 2003]. The moral is that, to truly adopt

internalism, we need to reconsider how we design datatypes and write programs, so their correctness are simply manifested in their construction, reducing the need of external justification which can be more awkward to produce.

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 16

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, 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. ↰ pages 16 and 33

Thorsten Altenkirch and Peter Morris [2009]. Indexed containers. In *Logic in Computer Science*, LICS'09, pages 277–285. IEEE. doi: 10.1109/LICS.2009.33. ↰ page 24

Yves Bertot and Pierre Castéran [2004]. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag. ↰ page 26

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 *Lec-*

ture Notes in Computer Science, pages 57–99. Springer-Verlag. doi: 10.1007/978-3-642-03153-3_2. ↰ 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. doi: 10.1007/978-3-540-24849-1_8. ↰ pages 18 and 27

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 16

Pierre-Évariste Dagand and Conor McBride [2012a]. Elaborating inductive definitions. arXiv:1210.6390. ↰ page 22

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 16 and 17

Michael Dummett [2000]. *Elements of Intuitionism*. Oxford University Press, second edition. ↰ page 2

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. ↰ pages 16 and 33

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 6

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

C. A. R. Hoare [1969]. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580. doi: 10.1145/363235.363259. ↰ page 5

Hsiang-Shang Ko and Jeremy Gibbons [2013]. Modularising inductive families. *Progress in Informatics*, 10:65–88. doi: 10.2201/NiiPi.2013.10.5. ↰ page 17

Zhaohui Luo [1994]. *Computation and Reasoning: A Type Theory for Computer Science*. Clarendon Press. ↰ pages 11, 25, and 33

Per Martin-Löf [1975]. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier B.V. doi: 10.1016/S0049-237X(08)71945-1. ↰ page 1

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. ↰ page 1

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

Conor McBride [2003]. First-order unification by structural recursion. *Journal of Functional Programming*, 13(6):1061–1075. doi: 10.1017/S0956796803004957. ↰ page 27

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 1 and 27

Conor McBride [2011]. Ornamental algebras, algebraic ornaments. To appear in *Journal of Functional Programming*. ↰ page 23

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, 9, and 10

Bengt Nordström, Kent Peterson, and Jan M. Smith [1990]. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press. ↰ pages 1 and 25

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. doi: 10.1007/978-3-642-04652-0_5. ↰ page 1

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 26

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 15

Thomas Streicher [1993]. Investigations into intentional type theory. Habilitation thesis, Ludwig Maximilian Universität. ↰ pages 6 and 13

The Univalent Foundations Program [2013]. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, Princeton. ↰ pages 16 and 33

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 9

Todo list