Generic Zip and Traverse

Daniël Kamphorst (6050107)
Supervisor: Wouter Swierstra

Master of Science
Computing Science
Utrecht University
2023

Abstract

Zipping and traversal are two of the best-known list operations. Contrary to what one might expect, these operations are in no way limited to lists. In this thesis, we generalize the zip function to all polynomial functors and the traverse function to all finitary polynomial functors. Polynomial functors include essentially all data types with a single type parameter. We implement a zip bijection for polynomial functors in the Coq proof assistant. Such a bijection consists of a zip function, an unzip function and a proof that they are mutually inverse. We also implement a traverse function for finitary polynomial functors and prove that it has the expected properties.

Acknowledgements

I am deeply indebted to my supervisor, Wouter Swierstra, for his guidance and endless patience throughout the project. I would also like to express my appreciation to Study Advisor Sara O'Keeffe for her invaluable advice and encouragement. Lastly, my parents deserve special thanks for their emotional support.

Contents

Chapter 1

Introduction

Zipping is arguably one of the best-known list operations, especially in the field of functional programming. The zip function combines two lists of equal length into a single list of pairs. Informally, it is given by

$$\mathsf{zip}((a_1, a_2, \ldots, a_n), (b_1, b_2, \ldots, b_n)) \coloneqq ((a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)),$$

where lists are thought of as finite sequences and n is the length of the input lists. In a sense, this is just matrix transposition. Like matrix transposition, zipping is an invertible operation. The inverse operation is known as *unzipping*.

The type $\mathsf{List}(A)$ of lists of elements of a type A may be defined as the inductive type whose constructors are $\mathsf{nil} : \mathsf{List}(A)$ and $\mathsf{cons} : A \to \mathsf{List}(A) \to \mathsf{List}(A)$. In practice, the zip function is usually defined recursively as follows:

$$\begin{aligned}
\mathsf{zip}(\mathsf{nil}, \mathsf{nil}) &\coloneqq \mathsf{nil}, \\
\mathsf{zip}(\mathsf{nil}, \mathsf{cons}(_)(_)) &\coloneqq \mathsf{nil}, \\
\mathsf{zip}(\mathsf{cons}(_)(_), \mathsf{nil}) &\coloneqq \mathsf{nil}, \\
\mathsf{zip}(\mathsf{cons}(a)(\ell), \mathsf{cons}(b)(\ell')) &\coloneqq \mathsf{cons}(a, b)(\mathsf{zip}(\ell, \ell')).
\end{aligned}$$

Note that this function is not injective and therefore not bijective, unlike the previous one. For example, it sends both $(\mathsf{nil}, \mathsf{nil})$ and $(\mathsf{nil}, \mathsf{cons}(4)(\mathsf{nil}))$ to nil.

One should not get the idea that zipping is specific to lists. Consider, for example, the inductive type $\mathsf{Tree}(A)$ of binary trees labelled with elements of a type A, whose constructors are $\mathsf{leaf} : \mathsf{Tree}(A)$ and $\mathsf{node} : \mathsf{Tree}(A) \to A \to \mathsf{Tree}(A) \to \mathsf{Tree}(A)$. A function that zips binary trees is defined as follows:

$$\begin{aligned}
\mathsf{zip}(\mathsf{leaf}, \mathsf{leaf}) &\coloneqq \mathsf{leaf}, \\
\mathsf{zip}(\mathsf{leaf}, \mathsf{node}(_)(_)(_)) &\coloneqq \mathsf{leaf}, \\
\mathsf{zip}(\mathsf{node}(_)(_)(_), \mathsf{leaf}) &\coloneqq \mathsf{leaf}, \\
\mathsf{zip}(\mathsf{node}(t_1)(a)(t_2), \mathsf{node}(t_1')(b)(t_2')) &\coloneqq \mathsf{node}(\mathsf{zip}(t_1, t_1'))(a, b)(\mathsf{zip}(t_2, t_2')).
\end{aligned}$$

Zipping is, without a doubt, a useful operation for any kind of data structure. However, defining a separate zip function for each data structure is a

repetitive and time-consuming task. Ideally, we would like a single zip function that works for all data structures, or more precisely, all inductive types with a single type parameter. Hereafter, we refer to these as *polymorphic data types*.

The process of writing functions that work for a variety of data types is commonly known as *generic programming*. A standard reference on the subject is [5]. Morris [23] provides a comprehensive overview of different approaches to generic programming. We use the approach described in the fifth chapter of [23], which is based on the concept of *polynomial functors*. Polynomial functors, also called container functors, are discussed in Chapter 4.

As we shall see in Chapter 4, polymorphic data types are polynomial functors. Abbott et al. [2] were the first to show that 'strictly positive types in one variable', which include polymorphic data types, are polynomial functors. To obtain a zip function for polymorphic data types, it is therefore sufficient to define one for polynomial functors.

In Chapter 3, we formally define the standard zip bijection briefly described in the opening paragraph. The standard zip bijection is generalized to all polynomial functors in Chapter 5. As Jay and Cockett [14] noted, the existence of a zip bijection for polynomial functors follows from the categorical fact that polynomial functors preserve *pullbacks*. Pullbacks, along with other relevant theoretical concepts, are introduced in Chapter 2. The papers [11] and [22] are noteworthy for generalizing the standard zip bijection in ways that are different from ours.

As Moggi et al. [22] point out, the zip function is conceptually similar to the *traverse function*. Jaskelioff and Rypáček [13], among others, have examined traversability of functors. In brief, a traversable functor is one for which there is a traverse function that satisfies certain conditions. Chapter 6 firstly shows that List is traversable. It turns out that not all polymorphic data types are traversable: only *finitary* ones are. After explaining the concept of finitary (polynomial) functors, we will prove that finitary polynomial functors are traversable.

To date, little attention has been paid to the formalization of these results using a proof assistant, such as Coq. Our contributions include the formalization of the results in homotopy type theory and their implementation in Coq. The Coq code, which builds on the HoTT library, is available here.

Chapter 2

Theoretical Background

This chapter presents the theoretical background. Much of the information presented here comes from standard textbooks on homotopy type theory [27, 28] and category theory [17, 18].

2.1 Dependent Types

A *dependent type*, or *type family*, is a function A from a type I to the type Type of all (small) types. If i is an element of I, then $A(i)$ is a type.

2.2 Basic Types

Functions

If A and B are types, the *function type* $A \to B$ is the type of functions from A to B. More generally, given a type A and a dependent type $B : A \to \mathsf{Type}$, the *dependent function type* $\prod_{a:A} B(a)$ is the type of functions sending each element $a : A$ to an element of $B(a)$. Such a function is called a *dependent function* because its codomain depends on its argument. If B is a constant family, then the dependent function type $\prod_{a:A} B$ is simply the function type $A \to B$.

Natural Numbers

The type $\mathbb{N}$ of natural numbers is the inductive type whose constructors are $0 : \mathbb{N}$ and $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$. Natural numbers are conventionally written as decimals. For example, $\mathsf{succ}(\mathsf{succ}(0))$ is written 2.

A natural number is either zero or the successor of another natural number. This means that we can show that a given statement is true for all natural numbers by first proving it for 0, and then for $\mathsf{succ}(n)$, assuming the statement is true for n.

Given a type X, an element $x_1 : X$ and a function $x_2 : X \to X$, there is a function $f : \mathbb{N} \to X$, given by

$$f(0) \coloneqq x_1,$$
$$f(\mathsf{succ}(n)) \coloneqq x_2(f(n)).$$

This is the *recursion principle* for $\mathbb{N}$, represented by a function

$$\mathsf{rec}^{\mathbb{N}} : \prod_{X:\mathsf{Type}} (X \to (X \to X) \to (\mathbb{N} \to X)).$$

The recursion principle for $\mathbb{N}$ is a special case of the *induction principle* for $\mathbb{N}$, which states: given a dependent type $X : \mathbb{N} \to \mathsf{Type}$, an element $x_1 : X(0)$ and a dependent function $x_2 : \prod_{n:\mathbb{N}}(X(n) \to X(\mathsf{succ}(n)))$, there is a dependent function $f : \prod_{n:\mathbb{N}} X(n)$, given by

$$f(0) \coloneqq x_1,$$
$$f(\mathsf{succ}(n)) \coloneqq x_2(n)(f(n)).$$

The corresponding function is denoted by $\mathsf{ind}^{\mathbb{N}}$. The recursion principle for $\mathbb{N}$ can be derived directly from the induction principle for $\mathbb{N}$:

$$\mathsf{rec}^{\mathbb{N}}_X(x_1)(x_2) \coloneqq \mathsf{ind}^{\mathbb{N}}_{n \mapsto X}(x_1)(n \mapsto x_2).$$

We can use the recursion principle for $\mathbb{N}$ to obtain, for example, a function $\mathsf{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ that adds two natural numbers:

$$\mathsf{add} \coloneqq \mathsf{rec}^{\mathbb{N}}_{\mathbb{N} \to \mathbb{N}}(\mathsf{id}_{\mathbb{N}})(\mathsf{succ} \circ -).$$

Here, $\mathsf{id} : \prod_{A:\mathsf{Type}}(A \to A)$ is the polymorphic identity function, given by $\mathsf{id}_A(a) \coloneqq a$. Subscripts, such as A, will often be omitted for readability. The sum $\mathsf{add}(m)(n)$ of two natural numbers m and n is usually written $m + n$.

Clearly, deriving a recursion principle from an induction principle is a trivial task. When specifying an inductive type, we shall henceforth give only the induction principle.

Lists

If A is a type, the type $\mathsf{List}(A)$ of lists of elements of A is the inductive type whose constructors are $\mathsf{nil} : \mathsf{List}(A)$ and $\mathsf{cons} : A \to \mathsf{List}(A) \to \mathsf{List}(A)$. The induction principle for $\mathsf{List}(A)$ says that, given a family $X : \mathsf{List}(A) \to \mathsf{Type}$, an element $x_1 : X(\mathsf{nil})$ and a function

$$x_2 : \prod_{a:A} \prod_{\ell:\mathsf{List}(A)} (X(\ell) \to X(\mathsf{cons}(a)(\ell))),$$

there is a function $f : \prod_{\ell:\mathsf{List}(A)} X(\ell)$, given by

$$f(\mathsf{nil}) \coloneqq x_1,$$
$$f(\mathsf{cons}(a)(\ell)) \coloneqq x_2(a)(\ell)(f(\ell)).$$

The polymorphic data type $\mathsf{List} : \mathsf{Type} \to \mathsf{Type}$ is a functor. It sends a function $f : A \to B$ to the function $\mathsf{List}(f) : \mathsf{List}(A) \to \mathsf{List}(B)$ given by

$$\begin{aligned} \mathsf{List}(f)(\mathsf{nil}) &\coloneqq \mathsf{nil}, \\ \mathsf{List}(f)(\mathsf{cons}(a)(\ell)) &\coloneqq \mathsf{cons}(f(a))(\mathsf{List}(f)(\ell)). \end{aligned}$$

Binary Trees

If A is a type, the type $\mathsf{Tree}(A)$ of binary trees in which each node is labelled with an element of A is the inductive type whose constructors are $\mathsf{leaf} : \mathsf{Tree}(A)$ and $\mathsf{node} : \mathsf{Tree}(A) \to A \to \mathsf{Tree}(A) \to \mathsf{Tree}(A)$. The induction principle for $\mathsf{Tree}(A)$ states: given a family $X : \mathsf{Tree}(A) \to \mathsf{Type}$, an element $x_1 : X(\mathsf{leaf})$ and a function

$$\prod_{t_1 : \mathsf{Tree}(A)} \Big(X(t_1) \to \prod_{a:A} \prod_{t_2 : \mathsf{Tree}(A)} (X(t_2) \to X(\mathsf{node}(t_1)(a)(t_2))) \Big),$$

there is a function $f : \prod_{t:\mathsf{Tree}(A)} X(t)$, given by

$$\begin{aligned} f(\mathsf{leaf}) &\coloneqq x_1, \\ f(\mathsf{node}(t_1)(a)(t_2)) &\coloneqq x_2(t_1)(f(t_1))(a)(t_2)(f(t_2)). \end{aligned}$$

Like List, the polymorphic data type $\mathsf{Tree} : \mathsf{Type} \to \mathsf{Type}$ is a functor. It sends a function $f : A \to B$ to the function $\mathsf{Tree}(f) : \mathsf{Tree}(A) \to \mathsf{Tree}(B)$ given by

$$\begin{aligned} \mathsf{Tree}(f)(\mathsf{leaf}) &\coloneqq \mathsf{leaf}, \\ \mathsf{Tree}(f)(\mathsf{node}(t_1)(a)(t_2)) &\coloneqq \mathsf{node}(\mathsf{Tree}(f)(t_1))(f(a))(\mathsf{Tree}(f)(t_2)). \end{aligned}$$

Pairs

If A and B are types, the *product type* $A \times B$ is the type of pairs consisting of an element of A and an element of B. The induction principle for $A \times B$ says: given a family $X : A \times B \to \mathsf{Type}$ and a function $x : \prod_{a:A} \prod_{b:B} X(a,b)$, there is a function $f : \prod_{(a,b):A\times B} X(a,b)$, given by $f(a,b) \coloneqq x(a)(b)$. Applying the induction principle for $A \times B$ is referred to as *uncurrying*.

Product types are essentially special cases of *dependent pair types*. Given a type A and a dependent type $B : A \to \mathsf{Type}$, the dependent pair type $\sum_{a:A} B(a)$ is the type of pairs consisting of an element $a : A$ and an element of $B(a)$. Not surprisingly, such pairs are called *dependent pairs*. The induction principle for $\sum_{a:A} B(a)$ states that, given a family $X : \sum_{a:A} B(a) \to \mathsf{Type}$ and a function $x : \prod_{a:A} \prod_{b:B(a)} X(a,b)$, there is a function $f : \prod_{(a,b):\sum_{a:A} B(a)} X(a,b)$, given by $f(a,b) \coloneqq x(a)(b)$.

Product types and dependent pair types will be discussed further after we introduce *equivalences*.

2.3 Equality

We distinguish two kinds of equality: *judgmental equality* and *typal equality*.

Two terms a and a' of the same type are said to be judgmentally equal—written $a \equiv a'$—if they reduce to the same canonical form [31]. For example, the natural numbers $\mathsf{id}(2)$ and $(n \mapsto \mathsf{succ}(n))(1)$ are judgmentally equal because they both reduce to 2. In contrast, the sum $2 + n$ of 2 and a natural number n is not judgmentally equal to $n + 2$. They are, rather, typally equal, which can be proved by induction on n.

Two elements a and a' of a type A are said to be typally equal if the *identity type* $a =_A a'$ (usually written simply $a = a'$) is inhabited. The identity type $a = a'$ is the type of *identifications* of a with a'. The type $a = a'$ itself is not inductively defined. Instead, the family $(a = -) : A \to \mathsf{Type}$ is, whose only constructor is $\mathsf{refl}_a : a = a$. The induction principle for $a = -$ says that, given a function $X : \prod_{a':A}((a = a') \to \mathsf{Type})$ and an element $x : X(a)(\mathsf{refl}_a)$, there is a function $f : \prod_{a':A} \prod_{w:a=a'} X(a')(w)$, given by $f(a)(\mathsf{refl}_a) \coloneqq x$.

Two judgmentally equal terms are necessarily also typally equal. If $a \equiv a'$, then $(a = a) \equiv (a = a')$, which means that refl_a is not only an element of $a = a$ but also of $a = a'$ [31].

Let A be a type, and let $P : A \to \mathsf{Type}$ be a dependent type. For each identification $w : a =_A a'$, there is a *transport function* $w_* : P(a) \to P(a')$, thought of as transporting elements of $P(a)$ along w.

Identifications can be composed: for any two identifications $w : a =_A a'$ and $w' : a' =_A a''$, there is an identification $w \cdot w' : a =_A a''$. Furthermore, every identification $w : a =_A a'$ has an inverse $w^{-1} : a' =_A a$. This reflects the well-known fact that equality is an equivalence relation.

A function $f : A \to B$ can be seen as a functor in the sense that there is a function

$$\mathsf{ap}_f : \prod_{a:A} \prod_{a':A} ((a =_A a') \to (f(a) =_B f(a')))$$

with the property that $\mathsf{ap}_f(\mathsf{refl}_a) \equiv \mathsf{refl}_{f(a)}$ and $\mathsf{ap}_f(w \cdot w') = \mathsf{ap}_f(w) \cdot \mathsf{ap}_f(w')$.

2.4 Homotopies

In most cases, it is more difficult to show that two functions are equal than it is to prove that they are pointwise equal. In fact, it is often impossible to show that two functions are equal without the function extensionality axiom, discussed in Section 2.9. This is why pointwise equality is, in general, to be preferred. Equality implies pointwise equality, but not vice versa. The function extensionality axiom serves to remedy the problem.

Let $f, g : \prod_{a:A} B(a)$ be two dependent functions. A *homotopy* from f to g is an element of the type

$$(f \sim g) \coloneqq \prod_{a:A} (f(a) = g(a)).$$

Like regular equality, pointwise equality is an equivalence relation. The identity homotopy $f \sim f$ is the function $a \mapsto \mathsf{refl}_{f(a)}$, the composite $f \sim h$ of two homotopies $\alpha : f \sim g$ and $\beta : g \sim h$ is the function $a \mapsto \alpha(a) \cdot \beta(a)$, and the inverse $g \sim f$ of a homotopy $\alpha : f \sim g$ is the function $a \mapsto \alpha(a)^{-1}$.

Finally, the horizontal composite $g \circ f \sim g \circ f'$ of a homotopy $\alpha : f \sim f'$ and a function g is the function $a \mapsto \mathsf{ap}_g(\alpha(a))$, and the horizontal composite $g \circ f \sim g' \circ f$ of a function f and a homotopy $\beta : g \sim g'$ is the function $a \mapsto \beta(f(a))$.

2.5 Contractibility

A type is said to be *contractible* if it contains exactly one element, or more precisely, if it contains an element (called the *centre of contraction*) to which all others are equal. In other words, a type A is contractible if the type

$$\sum_{a:A} \prod_{a':A} (a = a')$$

is inhabited.

The unit type, discussed in Section 2.7, is the most obvious example of a contractible type. A further example is the type $\sum_{a':A}(a = a')$, given an element $a : A$. For this type, the centre of contraction is (a, refl_a). A function

$$\prod_{(a',w):\sum_{a':A}(a=a')} ((a, \mathsf{refl}_a) = (a', w))$$

is obtained using the induction principles for $\sum_{a':A}(a = a')$ and $a = -$.

2.6 Equivalences

An *equivalence* between two types is analogous to a bijection between two sets.

A function is said to be an equivalence if it is *bi-invertible*, that is, if it has both a *retraction* and a *section*. A retraction of a function $f : A \to B$ is a function $r : B \to A$ with the property that $r \circ f \sim \mathsf{id}_A$, and a section of a function $f : A \to B$ is a function $s : B \to A$ such that $f \circ s \sim \mathsf{id}_B$. Put differently, a function $f : A \to B$ is an equivalence if the type

$$\mathsf{IsEquivalence}(f) :\equiv \sum_{r:B\to A} (r \circ f \sim \mathsf{id}_A) \times \sum_{s:B\to A} (f \circ s \sim \mathsf{id}_B)$$

is inhabited. An equivalence from a type A to a type B is an element of the type

$$(A \simeq B) :\equiv \sum_{f:A\to B} \mathsf{IsEquivalence}(f).$$

A function is an equivalence if and only if it has an inverse. An inverse of a function $f : A \to B$ is an element of the type

$$\mathsf{Inverse}(f) :\equiv \sum_{g:B\to A} ((g \circ f \sim \mathsf{id}_A) \times (f \circ g \sim \mathsf{id}_B)).$$

If $((r, \eta), (s, \epsilon))$ is an element of $\mathsf{IsEquivalence}(f)$, then (r, η, ϵ') is in $\mathsf{Inverse}(f)$, where ϵ' is given by

$$\epsilon'(b) : f(r(b)) \xrightarrow{\mathsf{ap}_{f \circ r}(\epsilon(b)^{-1})} f(r(f(s(b)))) \xrightarrow{\mathsf{ap}_f(\eta(s(b)))} f(s(b)) \xrightarrow{\epsilon(b)} b.$$

Conversely, if (g, η, ϵ) is an element of $\mathsf{Inverse}(f)$, then $((g, \eta), (g, \epsilon))$ is in $\mathsf{IsEquivalence}(f)$.

A type is said to be a (mere) proposition if any two of its elements are equal. Such a type contains either exactly one element or none, indicating respectively the truth or falsity of the fact it states. The type of equivalences from a type A to a type B is not defined as $\sum_{f:A\to B} \mathsf{Inverse}(f)$ because $\mathsf{Inverse}(f)$ is not a proposition, unlike $\mathsf{IsEquivalence}(f)$ [31]. Nevertheless, showing that a function has an inverse is sufficient to prove that is an equivalence, as explained above.

Type equivalence, as the name implies, is an equivalence relation. For any type A, the identity function $\mathsf{id}_A : A \to A$ is an equivalence. Furthermore, equivalences are closed under composition, and the inverse of an equivalence is itself an equivalence.

2.7 Limits and Colimits

Several limits and colimits in the category Type and in the category of functors from Type to Type are described in this section.

Initial Object

The initial object of the category Type is the *empty type* $\mathsf{0}$, which, as the name suggests, has no constructors and therefore contains no elements. There is a unique dependent function $\mathsf{ind}^{\mathsf{0}}_X : \prod_{i:\mathsf{0}} X(i)$ for any dependent type $X : \mathsf{0} \to \mathsf{Type}$. As a result, there is a unique function $\mathsf{rec}^{\mathsf{0}}_X : \mathsf{0} \to X$ for any type X.

Terminal Object

The terminal object of the category Type is the *unit type* $\mathsf{1}$, whose only constructor is $\star : \mathsf{1}$. The unit type is, by definition, contractible. For any type X, the constant function $x \mapsto \star$ is clearly the only function from X to $\mathsf{1}$.

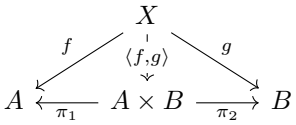
The induction principle for $\mathsf{1}$ states: given a family $X : \mathsf{1} \to \mathsf{Type}$ and an element $x : X(\star)$, there is a function $f : \prod_{i:\mathsf{1}} X(i)$, given by $f(\star) \coloneqq x$. The function $\mathsf{ind}^{\mathsf{1}}_X : X(\star) \to \prod_{i:\mathsf{1}} X(i)$ is an equivalence; its inverse is the function $f \mapsto f(\star)$.

Products

The product of two types A and B is the product type $A \times B$. The projections $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ are defined as follows:

$$\pi_1(a, b) \coloneqq a,$$
$$\pi_2(a, b) \coloneqq b.$$

Given a type X and two functions $f : X \to A$ and $g : X \to B$, the unique function $\langle f, g\rangle : X \to A \times B$ making the diagram



commute is given by $\langle f, g\rangle(x) \coloneqq (f(x), g(x))$. The function

$$((f, g) \mapsto \langle f, g\rangle) : (X \to A) \times (X \to B) \to (X \to A \times B)$$

is an equivalence; its inverse is the function $h \mapsto (\pi_1 \circ h, \pi_2 \circ h)$.

The operation $\times : \mathsf{Type} \times \mathsf{Type} \to \mathsf{Type}$ is a functor. It sends a pair of functions $f : A \to A'$ and $g : B \to B'$ to the function

$$f \times g \coloneqq \langle f \circ \pi_1, g \circ \pi_2\rangle : A \times B \to A' \times B'.$$

More generally, given a function $f : A \to A'$ and a dependent function $g : \prod_{a:A}(B(a) \to B'(f(a)))$, there is a function

$$\sum_f g : \sum_{a:A} B(a) \to \sum_{a':A'} B'(a'),$$

defined as $(\sum_f g)(a, b) \coloneqq (f(a), g(a)(b))$. This function is an equivalence if f is one and, for every $a : A$, the function $g(a)$ is an equivalence. The inverse of $\sum_f g$ is $\sum_{f^{-1}} g'$, where g' is given by

$$g'(a') : B'(a') \xrightarrow{(\epsilon(a')^{-1})_*} B'(f(f^{-1}(a'))) \xrightarrow{g(f^{-1}(a'))^{-1}} B(f^{-1}(a')).$$

Here, ϵ is the homotopy $f \circ f^{-1} \sim \mathsf{id}_{A'}$.

Limits and colimits in functor categories are computed pointwise [17, 18]. The product of two functors $F : \mathsf{Type} \to \mathsf{Type}$ and $G : \mathsf{Type} \to \mathsf{Type}$ is the functor $F \times G : \mathsf{Type} \to \mathsf{Type}$ that sends each type A to the type $F(A) \times G(A)$ and each function $f : A \to B$ to the function $F(f) \times G(f)$.

Coproducts

Binary Coproducts

The coproduct of two types A and B is the *coproduct type* $A + B$, whose constructors are $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$. The induction principle for $A + B$ says that, given a family $X : A + B \to \mathsf{Type}$ and two functions $f : \prod_{a:A} X(\mathsf{inl}(a))$ and $g : \prod_{b:B} X(\mathsf{inr}(b))$, there is a function $[f, g] : \prod_{i:A+B} X(i)$, given by

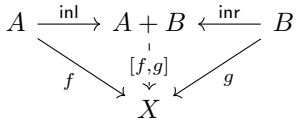
$$[f, g](\mathsf{inl}(a)) \coloneqq f(a),$$
$$[f, g](\mathsf{inr}(b)) \coloneqq g(b).$$

The function

$$((f,g) \mapsto [f,g]) : \prod_{a:A} X(\mathsf{inl}(a)) \times \prod_{b:B} X(\mathsf{inr}(b)) \to \prod_{i:A+B} X(i)$$

is an equivalence; its inverse is the function $h \mapsto (h \circ \mathsf{inl}, h \circ \mathsf{inr})$.

Given a type X and two functions $f : A \to X$ and $g : B \to X$, the function $[f,g] : A + B \to X$ is the only one making the following diagram commute:



Arbitrary Coproducts

Let I be a type. The coproduct of a family $A : I \to \mathsf{Type}$ of types is the dependent pair type $\sum_{i:I} A(i)$. It is for this reason that dependent pair types are also called *dependent sum types*. For each $i : I$, the inclusion $A(i) \to \sum_{i:I} A(i)$ is the function $(i, -)$. Given a type X and a dependent function $f : \prod_{i:I}(A(i) \to X)$, the function

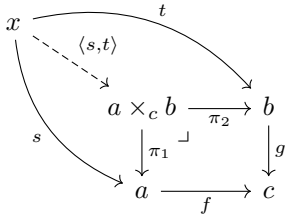
$$[f] :\equiv \mathsf{rec}_X^{\sum_{i:I} A(i)}(f) : \sum_{i:I} A(i) \to X$$

is the only one with the property that $[f] \circ (i, -) = f(i)$ for every $i : I$.

The coproduct of a family $F : I \to (\mathsf{Type} \to \mathsf{Type})$ of functors from Type to Type is the functor $\sum_{i:I} F(i) : \mathsf{Type} \to \mathsf{Type}$ that sends each type A to the type $\sum_{i:I} F(i)(A)$ and each function $f : A \to B$ to the function $[i \mapsto (i, -) \circ F(i)(f)]$.

Pullbacks

The pullback of two morphisms $f\colon a \to c$ and $g\colon b \to c$ of a category $\mathcal{C}$ is an object $a \times_c b \in \mathcal{C}$ together with a pair of morphisms $\pi_1\colon a \times_c b \to a$ and $\pi_2\colon a \times_c b \to b$ having the property that $f \circ \pi_1 = g \circ \pi_2$, such that for any other object $x \in \mathcal{C}$ together with a pair of morphisms $s\colon x \to a$ and $t\colon x \to b$ having the property that $f \circ s = g \circ t$, there is a unique morphism $\langle s, t\rangle\colon x \to a \times_c b$ making the following diagram commute:



Although similar, pullbacks in type theory are slightly more complex. The (homotopy) pullback of two functions $f : A \to C$ and $g : B \to C$ is a type

$A \times_C B$ together with two functions $\mathrel{⌟}\pi_1 : A \times_C B \to A$ and $\mathrel{⌟}\pi_2 : A \times_C B \to B$ and a homotopy $\mathrel{⌟}\pi_3 : f \circ \mathrel{⌟}\pi_1 \sim g \circ \mathrel{⌟}\pi_2$, such that for any other type X together with two functions $s : X \to A$ and $t : X \to B$ and a homotopy $\alpha : f \circ s \sim g \circ t$, there is a unique function $\langle s, t, \alpha \rangle : X \to A \times_C B$ having the property that

$$(\mathrel{⌟}\pi_1 \circ \langle s, t, \alpha \rangle, \mathrel{⌟}\pi_2 \circ \langle s, t, \alpha \rangle, \mathrel{⌟}\pi_3 \circ \langle s, t, \alpha \rangle) = (s, t, \alpha).$$

The pullback of two functions $f : A \to C$ and $g : B \to C$ is the type

$$A \times_C B :\equiv \sum_{a:A} \sum_{b:B} (f(a) = g(b)).$$

The projections $\mathrel{⌟}\pi_1$, $\mathrel{⌟}\pi_2$ and $\mathrel{⌟}\pi_3$ are defined as follows:

$$\mathrel{⌟}\pi_1(a, b, w) :\equiv a$$
$$\mathrel{⌟}\pi_2(a, b, w) :\equiv b$$
$$\mathrel{⌟}\pi_3(a, b, w) :\equiv w.$$

Let X be a type. Given two functions $s : X \to A$ and $t : X \to B$ and a homotopy $\alpha : f \circ s \sim g \circ t$, the function $\langle s, t, \alpha \rangle : X \to A \times_C B$ is given by

$$\langle s, t, \alpha \rangle(x) :\equiv (s(x), t(x), \alpha(x)).$$

Every type A is equivalent to the pullback

$$A \times_A A \equiv \sum_{a:A} \sum_{a':A} (a = a')$$

of the identity function id_A and itself. The projection $\mathrel{⌟}\pi_1 : A \times_A A \to A$ is an equivalence because the type $\sum_{a':A}(a = a')$ is contractible for every $a : A$ (see Section 2.5). Its inverse is the function $a \mapsto (a, a, \mathsf{refl}_a)$.

2.8 Representable Functors

The type family $(A \to -) : \mathsf{Type} \to \mathsf{Type}$ is a functor. It sends a function $h : B \to B'$ to the function $(h \circ -) : (A \to B) \to (A \to B')$. The functor $A \to -$ is denoted by y^A, as the notation $A \to h$ is potentially confusing. The symbol y stands for Yoneda [18, 34].

A *representable functor* is one that is naturally equivalent to the functor y^A for some type A. The functor y^A itself is referred to as the functor represented by A.

2.9 Equality of Structures

As mentioned earlier, giving a homotopy between two functions tends to be easier than giving an identification between them. In the same way, giving

two identifications $a = a'$ and $b = b'$ is generally easier than giving a single identification $(a, b) = (a', b')$. It is often possible to provide a simpler yet equivalent characterization of a given identity type. More concretely, we can often show that a given identity type is equivalent to a type that is more convenient to work with. Two important examples are given below.

Functions

Traditionally, two functions $f, g : \prod_{a:A} B(a)$ are considered equal if and only if they are pointwise equal. This suggests that there is an equivalence

$$(f = g) \simeq (f \sim g),$$

but this is not the case. Although it is easy to define a function $\mathsf{happly}_{f,g}$ from $f = g$ to $f \sim g$, it is impossible to prove that it is an equivalence. We therefore have no choice but to assume the existence of an inverse. The axiom stating that $\mathsf{happly}_{f,g}$ is an equivalence is called *function extensionality*.

Natural Numbers

We can determine whether two natural numbers are equal by taking a closer look at them. The successor of a natural number m is equal to the successor of a natural number n if and only if m and n are equal. On the other hand, 0 is equal only to itself. For any two natural numbers m and n, there is an equivalence

$$(m = n) \simeq \mathsf{Equals}^{\mathbb{N}}(m)(n),$$

where $\mathsf{Equals}^{\mathbb{N}} : \mathbb{N} \to \mathbb{N} \to \mathsf{Type}$ is given by

$$\begin{aligned}
\mathsf{Equals}^{\mathbb{N}}(0)(0) &\coloneqq 1, \\
\mathsf{Equals}^{\mathbb{N}}(0)(\mathsf{succ}(_)) &\coloneqq 0, \\
\mathsf{Equals}^{\mathbb{N}}(\mathsf{succ}(_))(0) &\coloneqq 0, \\
\mathsf{Equals}^{\mathbb{N}}(\mathsf{succ}(m))(\mathsf{succ}(n)) &\coloneqq \mathsf{Equals}^{\mathbb{N}}(m)(n).
\end{aligned}$$

The Equals relation is referred to as *observational equality* and is also denoted by 'Eq' [28] or 'code' [27]. The construction of the above equivalence is described in detail in [27] and [28].

Chapter 3

Standard Zip

We are now ready to formally define the standard zip equivalence, which is an equivalence between the type of pairs of lists of equal length and the type of lists of pairs. To specify the domain, we need a function $\mathsf{length} : \prod_{A:\mathsf{Type}}(\mathsf{List}(A) \to \mathbb{N})$ that sends any list to its length. Such a function is defined as follows:

$$\begin{aligned}\mathsf{length}_A(\mathsf{nil}) &:\equiv 0,\\ \mathsf{length}_A(\mathsf{cons}(\_)(\ell)) &:\equiv \mathsf{succ}(\mathsf{length}_A(\ell)).\end{aligned}$$

A pair of lists $\ell_1 : \mathsf{List}(A)$ and $\ell_2 : \mathsf{List}(B)$ of equal length is precisely an element of the pullback

$$\mathsf{List}(A) \times_{\mathbb{N}} \mathsf{List}(B) \equiv \sum_{\ell_1:\mathsf{List}(A)} \sum_{\ell_2:\mathsf{List}(B)} (\mathsf{length}(\ell_1) = \mathsf{length}(\ell_2))$$

of length_A and length_B. Thus, the standard zip equivalence is a polymorphic equivalence

$$\mathsf{zip} : \prod_{A:\mathsf{Type}} \prod_{B:\mathsf{Type}} (\mathsf{List}(A) \times_{\mathbb{N}} \mathsf{List}(B) \simeq \mathsf{List}(A \times B)).$$

For any two types A and B, the equivalence $\mathsf{zip}_{A,B}$ can be defined as the composite

$$\begin{aligned}&\mathsf{List}(A) \times_{\mathbb{N}} \mathsf{List}(B)\\ \equiv &\sum_{\ell_1:\mathsf{List}(A)} \sum_{\ell_2:\mathsf{List}(B)} (\mathsf{length}(\ell_1) = \mathsf{length}(\ell_2))\\ \simeq &\sum_{\ell_1:\mathsf{List}(A)} \sum_{\ell_2:\mathsf{List}(B)} \mathsf{Equals}^{\mathbb{N}}(\mathsf{length}(\ell_1))(\mathsf{length}(\ell_2))\\ \simeq &\mathsf{List}(A \times B).\end{aligned}$$

The first equivalence follows from the fact that $(m =_{\mathbb{N}} n) \simeq \mathsf{Equals}^{\mathbb{N}}(m)(n)$ for any two natural numbers m and n (see Section 2.9), and the second arises from

the fact that the type

$$\mathsf{List}'(A,B) :\equiv \sum_{\ell_1:\mathsf{List}(A)} \sum_{\ell_2:\mathsf{List}(B)} \mathsf{Equals}^{\mathbb{N}}(\mathsf{length}(\ell_1))(\mathsf{length}(\ell_2))$$

satisfies the same induction principle as $\mathsf{List}(A \times B)$ (see below). As stated in [27], a type is equivalent to a given inductive type if it satisfies the same induction principle.

The constructor $\mathsf{nil} : \mathsf{List}(A \times B)$ corresponds to the triple

$$\mathsf{nil}' :\equiv (\mathsf{nil}, \mathsf{nil}, \star) : \mathsf{List}'(A,B),$$

and the constructor $\mathsf{node} : A \times B \to \mathsf{List}(A \times B) \to \mathsf{List}(A \times B)$ corresponds to the function $\mathsf{node}' : A \times B \to \mathsf{List}'(A,B) \to \mathsf{List}'(A,B)$ given by

$$\mathsf{node}'(a,b)(\ell_1,\ell_2,c) :\equiv (\mathsf{cons}(a)(\ell_1), \mathsf{cons}(b)(\ell_2), c).$$

The type $\mathsf{List}'(A,B)$ contains exactly those elements that can be constructed using nil' and cons'. We can prove that $\mathsf{List}'(A,B)$, together with nil' and cons', satisfies the same induction principle as $\mathsf{List}(A \times B)$. Given a family $X : \mathsf{List}'(A,B) \to \mathsf{Type}$, an element $x_1 : X(\mathsf{nil}')$ and a function

$$x_2 : \prod_{(a,b):A\times B} \prod_{\ell':\mathsf{List}'(A,B)} (X(\ell') \to X(\mathsf{cons}'(a,b)(\ell'))),$$

there is a function $f : \prod_{\ell':\mathsf{List}'(A,B)} X(\ell')$, given by

$$\begin{aligned} f(\mathsf{nil},\mathsf{nil},\star) &:\equiv x_1, \\ f(\mathsf{nil},\mathsf{cons}(\_)(\_),c) &:\equiv \mathsf{rec}^{\mathsf{0}}(c), \\ f(\mathsf{cons}(\_)(\_),\mathsf{nil},c) &:\equiv \mathsf{rec}^{\mathsf{0}}(c), \\ f(\mathsf{cons}(a)(\ell_1),\mathsf{cons}(b)(\ell_2),c) &:\equiv x_2(a,b)(\ell_1,\ell_2,c)(f(\ell_1,\ell_2,c)). \end{aligned}$$

The recursion principle

$$\mathsf{rec}^{\mathsf{List}'(A,B)} : \prod_{X:\mathsf{Type}} (X \to (A \times B \to X \to X) \to (\mathsf{List}'(A,B) \to X))$$

can be derived in the usual way. We can use the recursion principles for $\mathsf{List}'(A,B)$ and $\mathsf{List}(A \times B)$ to obtain functions

$$\begin{aligned} \mathsf{asList}_{A,B} &: \mathsf{List}'(A,B) \to \mathsf{List}(A \times B), \\ \mathsf{asList}'_{A,B} &: \mathsf{List}(A \times B) \to \mathsf{List}'(A,B) \end{aligned}$$

respectively:

$$\begin{aligned} \mathsf{asList}_{A,B} &:\equiv \mathsf{rec}^{\mathsf{List}'(A,B)}_{\mathsf{List}(A\times B)}(\mathsf{nil})(\mathsf{cons}), \\ \mathsf{asList}'_{A,B} &:\equiv \mathsf{rec}^{\mathsf{List}(A\times B)}_{\mathsf{List}'(A,B)}(\mathsf{nil}')(\mathsf{cons}'). \end{aligned}$$

The induction principles are used to prove that asList' is the inverse of asList (see StandardZip.v).

We conclude this chapter with a brief demonstration of the standard zip equivalence just constructed. Consider the lists

$$\begin{aligned}\mathsf{cons}(2)(\mathsf{cons}(4)(\mathsf{nil})) &: \mathsf{List}(\mathbb{N}),\\ \mathsf{cons}(\mathsf{inl}(\star))(\mathsf{cons}(\mathsf{inr}(\star))(\mathsf{nil})) &: \mathsf{List}(1+1).\end{aligned}$$

Since

$$\begin{aligned}\mathsf{length}(\mathsf{cons}(2)(\mathsf{cons}(4)(\mathsf{nil}))) &\equiv 2\\ &\equiv \mathsf{length}(\mathsf{cons}(\mathsf{inl}(\star))(\mathsf{cons}(\mathsf{inr}(\star))(\mathsf{nil}))),\end{aligned}$$

the identification refl_2 is in the identity type

$$\mathsf{length}(\mathsf{cons}(2)(\mathsf{cons}(4)(\mathsf{nil}))) = \mathsf{length}(\mathsf{cons}(\mathsf{inl}(\star))(\mathsf{cons}(\mathsf{inr}(\star))(\mathsf{nil}))).$$

The function $\mathsf{zip}_{\mathbb{N},1+1}$ sends the triple

$$(\mathsf{cons}(2)(\mathsf{cons}(4)(\mathsf{nil})), \mathsf{cons}(\mathsf{inl}(\star))(\mathsf{cons}(\mathsf{inr}(\star))(\mathsf{nil})), \mathsf{refl}_2)$$

to the list $\mathsf{cons}(2, \mathsf{inl}(\star))(\mathsf{cons}(4, \mathsf{inr}(\star))(\mathsf{nil}))$. On the other hand, the function $\mathsf{unzip}_{\mathbb{N},1+1} :\equiv \mathsf{zip}^{-1}_{\mathbb{N},1+1}$ sends, for example, the list

$$\mathsf{cons}(0, \mathsf{inr}(\star))(\mathsf{cons}(1, \mathsf{inl}(\star))(\mathsf{nil})) : \mathsf{List}(\mathbb{N} \times (1+1))$$

to the triple $(\mathsf{cons}(0)(\mathsf{cons}(1)(\mathsf{nil})), \mathsf{cons}(\mathsf{inr}(\star))(\mathsf{cons}(\mathsf{inl}(\star))(\mathsf{nil})), \mathsf{refl}_2)$.

An implementation of the standard zip equivalence in Coq can be found here.

The next chapter explains that polymorphic data types, such as List, are polynomial functors. This fact is used in Chapter 5 to define a zip equivalence that works for all polymorphic data types.

Chapter 4

Polynomial Functors

Objects like lists and binary trees are essentially containers, used for storing data. They can be separated into two parts: shape and contents. For example, the shape of a list is its length n, and its contents are n data items. More formally, a list of elements of a type A is the same thing as a pair consisting of a natural number n and a function from $\mathsf{Fin}(n)$ to A. As will soon become apparent, this is a simple way of saying that List is a *polynomial functor*. Perhaps unsurprisingly, polynomial functors are also called container functors. Such functors are frequently used for generic programming [2, 23]. Polynomial functors are precisely those that preserve connected limits, including pullbacks [8]. There is a zip equivalence for any such functor [14]. Moreover, traversable functors are exactly finitary polynomial functors [12]. This is why we opted for an approach to generic programming based on polynomial functors.

4.1 Polynomials

A polynomial functor is the 'categorification' of a polynomial function [19].

A *polynomial* (in one variable) is an element p of the type

$$\sum_{I:\mathsf{Type}} (I \to \mathsf{Type}).$$

Elements of the type $\overset{\bullet}{p} \coloneqq \pi_1(p)$ are referred to as *positions* in p. The dependent type $\overrightarrow{p} \coloneqq \pi_2(p)$ sends each position $i : \overset{\bullet}{p}$ to the type $\overrightarrow{p}(i)$, whose elements are called *directions* at i. Polynomials are defined in Coq as follows:

```
Record Polynomial :=
  { Position : Type
  ; Direction : Position -> Type
  }.
```

The *extension* of a polynomial p is the coproduct

$$\sum_{i:\overset{\bullet}{p}} y^{\overrightarrow{p}(i)} : \mathsf{Type} \to \mathsf{Type}$$

of the family $i \mapsto y^{\overrightarrow{p}(i)}$ of representable functors. It sends a type X to the type $\sum_{i:\overset{\bullet}{p}} (\overrightarrow{p}(i) \to X)$ and a function $f : X \to Y$ to the function $(i, x) \mapsto (i, f \circ x)$. For the sake of simplicity, a polynomial and its extension are denoted by the same letter or expression. It should always be clear from the context which meaning is intended.

A *polynomial functor* (in one variable) is one that is naturally equivalent to the extension of a polynomial:

```
Class IsPolynomialFunctor (F : Type -> Type) `{Is0Functor F} :=
  { polynomial : Polynomial
  ; equivalenceToExtension : NatEquiv F (Extension polynomial)
  }.
```

Historical Remarks

During the last two decades, polynomial functors have become an important research topic in several academic fields, including computer science. Gambino and Kock [8, 15] provide a comprehensive survey of the history of the subject. Over time, many different definitions of the concept of polynomial functors have been proposed. An overview of these definitions and the relationships between them can be found in [8] and [9]. We have adopted the definition given in [34]. Researchers in computer science generally use the same definition but a different notation and terminology [1, 2]. Polynomials are commonly known as containers in the computer science literature. Furthermore, positions and directions are often referred to as shapes and positions respectively. Although heavily influenced by the work of Spivak and Niu, our notation differs slightly from theirs. They use the notation $p(1)$ for the type $\pi_1(p)$ of positions in a polynomial p. We, however, use the notation $\overset{\bullet}{p}$ to emphasize that, despite being equivalent, $p(1)$ and $\pi_1(p)$ are distinct types. The notation $\overrightarrow{p}$ is used instead of $p[-]$ for the sake of consistency.

4.2 Polymorphic Data Types

As suggested earlier, List is a polynomial functor. It is naturally equivalent to the extension of the polynomial $\mathsf{List}' \coloneqq (\mathbb{N}, \mathsf{Fin})$ (see PolynomialFunctors.v). For example, the list

$$\mathsf{cons}(2)(\mathsf{cons}(4)(\mathsf{nil})) : \mathsf{List}(\mathbb{N})$$

corresponds to the pair

$$(2, [[\mathsf{rec}^0, x \mapsto 4], x \mapsto 2]) : \mathsf{List}'(\mathbb{N}).$$

A further example of a polynomial functor is Tree. The shape of a tree is the tree with the labels removed [4], or more precisely, with every label replaced by the element $\star : \mathsf{1}$. For any type A, the shape of a tree $t : \mathsf{Tree}(A)$ is thus the tree $\mathsf{Tree}(a \mapsto \star)(t) : \mathsf{Tree}(\mathsf{1})$ [14, 23]. The functor Tree is naturally equivalent to the extension of the polynomial Tree' given by

$$\begin{aligned} \overset{\bullet}{\mathsf{Tree}}' &:= \mathsf{Tree}(\mathsf{1}), \\ \overset{\rightarrow}{\mathsf{Tree}}'(\mathsf{leaf}) &:= \mathsf{0}, \\ \overset{\rightarrow}{\mathsf{Tree}}'(\mathsf{node}(t_1)(\_)(t_2)) &:= \overset{\rightarrow}{\mathsf{Tree}}'(t_1) + \mathsf{1} + \overset{\rightarrow}{\mathsf{Tree}}'(t_2). \end{aligned}$$

A proof of

```
Instance TreeIsPolynomialFunctor : IsPolynomialFunctor Tree
```

can be found here. The tree

$$\mathsf{node}(\mathsf{leaf})(2)(\mathsf{node}(\mathsf{leaf})(4)(\mathsf{leaf})) : \mathsf{Tree}(\mathbb{N}),$$

to take one example, corresponds to the pair

$$(\mathsf{node}(\mathsf{leaf})(\star)(\mathsf{node}(\mathsf{leaf})(\star)(\mathsf{leaf})), [[\mathsf{rec}^0, x \mapsto 2], [[\mathsf{rec}^0, x \mapsto 4], \mathsf{rec}^0]])$$

in $\mathsf{Tree}'(\mathbb{N})$.

It has been shown that all polymorphic data types are polynomial functors [2, 4]. What follows is a brief sketch of the proof. Let T be a polymorphic data type. There is a polynomial functor F in two variables such that $T(A)$ is equivalent to the *least fixed point* of the polynomial functor $F(A, -)$ for every type A. This equivalence is natural in A. The functor that sends each type A to the least fixed point of $F(A, -)$ is itself a polynomial functor, which in turn makes T a polynomial functor.

The proof has been fully formalized in the Lean proof assistant [4], which is similar to Coq in many respects. It does not seem particularly valuable to formalize the very same result in Coq. Doing so would only complicate matters and distract attention from our main contributions. It is for this reason that we only showed that List and Tree are polynomial functors.

Chapter 5

Generic Zip

In the previous chapter, we established that polymorphic data types are polynomial functors. The next step is to generalize the standard zip equivalence defined in Chapter 3 to all polynomial functors. We will begin by defining a zip equivalence for extensions of polynomials. This zip equivalence is used to define one for all polynomial functors. The chapter concludes with a demonstration of the generic zip equivalence.

5.1 Extensions of Polynomials

As mentioned earlier, the length of a list can be regarded as its shape. The standard zip equivalence can therefore be described as an equivalence between the type of pairs of lists of the same shape and the type of lists of pairs.

Let p be a polynomial. For any type A, a pair in $p(A)$ consists of a position $i : \overset{\bullet}{p}$ and a function from $\overrightarrow{p}(i)$ to A. Such a pair should be thought of as an object whose shape is i and whose contents are represented by the function.

The zip equivalence for the extension of p is given by the composite

$$
\begin{aligned}
& p(A) \times_{\overset{\bullet}{p}} p(B) \\
\equiv & \sum_{(i,a):p(A)} \sum_{(i',b):p(B)} (i = i') \\
\simeq & \sum_{(i,i',w):\overset{\bullet}{p}\times_{\overset{\bullet}{p}}\overset{\bullet}{p}} ((\overrightarrow{p}(i) \to A) \times (\overrightarrow{p}(i') \to B)) \\
\simeq & \sum_{i:\overset{\bullet}{p}} ((\overrightarrow{p}(i) \to A) \times (\overrightarrow{p}(i) \to B)) \\
\simeq & \sum_{i:\overset{\bullet}{p}} (\overrightarrow{p}(i) \to A \times B) \\
\equiv & \, p(A \times B).
\end{aligned}
$$

The first equivalence simply rearranges the input data, and the third follows from the universal property of the product (see Section 2.7). Now let us clarify the second equivalence. The projection $\overset{\cdot}{\pi}_1 : \overset{\bullet}{p} \times_{\overset{\bullet}{p}} \overset{\bullet}{p} \to \overset{\bullet}{p}$ is an equivalence (see Section 2.7), and so is its inverse. Furthermore, for every position $i : \overset{\bullet}{p}$, the identity function $\mathsf{id}_{(\overrightarrow{p}(i)\to A)\times(\overrightarrow{p}(i)\to B)}$ is an equivalence

$$\begin{aligned}&(i \mapsto (\overrightarrow{p}(i) \to A) \times (\overrightarrow{p}(i) \to B))(i)\\ \simeq\ &((i, i', w) \mapsto (\overrightarrow{p}(i) \to A) \times (\overrightarrow{p}(i') \to B))(\overset{\cdot}{\pi}_1^{-1}(i))\end{aligned}$$

because

$$\begin{aligned}&(i \mapsto (\overrightarrow{p}(i) \to A) \times (\overrightarrow{p}(i) \to B))(i)\\ \equiv\ &(\overrightarrow{p}(i) \to A) \times (\overrightarrow{p}(i) \to B)\\ \equiv\ &((i, i', w) \mapsto (\overrightarrow{p}(i) \to A) \times (\overrightarrow{p}(i') \to B))(i, i, \mathsf{refl}_i)\\ \equiv\ &((i, i', w) \mapsto (\overrightarrow{p}(i) \to A) \times (\overrightarrow{p}(i') \to B))(\overset{\cdot}{\pi}_1^{-1}(i)).\end{aligned}$$

As described in Section 2.7, it follows that the function

$$\sum_{\overset{\cdot}{\pi}_1^{-1}} (i \mapsto \mathsf{id}_{(\overrightarrow{p}(i)\to A)\times(\overrightarrow{p}(i)\to B)})$$

is an equivalence

$$\sum_{i:\overset{\bullet}{p}} ((\overrightarrow{p}(i) \to A) \times (\overrightarrow{p}(i) \to B)) \simeq \sum_{(i,i',w):\overset{\bullet}{p}\times_{\overset{\bullet}{p}}\overset{\bullet}{p}} ((\overrightarrow{p}(i) \to A) \times (\overrightarrow{p}(i') \to B)).$$

The unzip function for the extension of p sends a pair (i, x) to the triple $((i, \pi_1 \circ x), (i, \pi_2 \circ x), \mathsf{refl}_i)$. On the other hand, the zip function for the extension of p sends a triple $((i, a), (i', b), w)$ to the pair $(i, \langle \pi_1(w'_*(a, b)), \pi_2(w'_*(a, b))\rangle)$, where w' is the identification $(i, i', w) = \overset{\cdot}{\pi}_1^{-1}(\overset{\cdot}{\pi}_1(i, i', w))$. Computationally, the involvement of the transport function w'_* may seem problematic. However, this is impossible to avoid if i and i' are typally rather than judgmentally equal. Fortunately, $(i, \langle \pi_1(w'_*(a, b)), \pi_2(w'_*(a, b))\rangle)$ reduces to $(i, \langle a, b\rangle)$ as long as $i \equiv i'$ and $w \equiv \mathsf{refl}_i$, since

$$\begin{aligned}w'_* &\equiv (\mathsf{ap}_{(i,-)}(\mathsf{contr}_{(i,\mathsf{refl}_i)})^{-1})_*\\ &\equiv (\mathsf{ap}_{(i,-)}(\mathsf{refl}_{(i,\mathsf{refl}_i)})^{-1})_*\\ &\equiv ((\mathsf{refl}_{(i,i,\mathsf{refl}_i)})^{-1})_*\\ &\equiv (\mathsf{refl}_{(i,i,\mathsf{refl}_i)})_*\\ &\equiv \mathsf{id}.\end{aligned}$$

As we shall see later, it is relatively easy to ensure that this condition is always satisfied.

5.2 Polynomial Functors

Let F be a polynomial functor, meaning that there is a natural equivalence α from F to the extension of some polynomial p. For any type A, the shape of an element $x : F(A)$ is the first component of the pair $\alpha_A(x) : p(A)$.

The zip equivalence for F is given by the composite

$$\begin{aligned} & F(A) \times_{\overset{\bullet}{p}} F(B) \\ \simeq\ & p(A) \times_{\overset{\bullet}{p}} p(B) \\ \simeq\ & p(A \times B) \\ \simeq\ & F(A \times B). \end{aligned}$$

Contrary to what one might expect, the first equivalence, which is an equivalence from

$$F(A) \times_{\overset{\bullet}{p}} F(B) \equiv \sum_{x:F(A)} \sum_{y:F(B)} (\pi_1(\alpha_A(x)) = \pi_1(\alpha_B(y)))$$

to

$$p(A) \times_{\overset{\bullet}{p}} p(B) \equiv \sum_{x':p(A)} \sum_{y':p(B)} (\pi_1(x') = \pi_1(y')),$$

is not defined as

$$\sum_{\alpha_A} \Big(x \mapsto \sum_{\alpha_B} (y \mapsto \mathsf{id}_{\pi_1(\alpha_A(x)) = \pi_1(\alpha_B(y))}) \Big).$$

Given a triple $(x', y', w) : p(A) \times_{\overset{\bullet}{p}} p(B)$, the inverse of this equivalence transports the whole pair (y', w) along the identification $x' = \alpha_A(\alpha_A^{-1}(x'))$ even though only the type of w depends on x'. So that it transports just w, the first equivalence is defined as the composite

$$\begin{aligned} & \sum_{x:F(A)} \sum_{y:F(B)} (\pi_1(\alpha_A(x)) = \pi_1(\alpha_B(y))) \\ \simeq\ & \sum_{(x,y):F(A)\times F(B)} (\pi_1(\alpha_A(x)) = \pi_1(\alpha_B(y))) \\ \simeq\ & \sum_{(x',y'):p(A)\times p(B)} (\pi_1(x') = \pi_1(y')) \\ \simeq\ & \sum_{x':p(A)} \sum_{y':p(B)} (\pi_1(x') = \pi_1(y')). \end{aligned}$$

The second of these equivalences is

$$\sum_{\alpha_A \times \alpha_B} ((x, y) \mapsto \mathsf{id}_{\pi_1(\alpha_A(x)) = \pi_1(\alpha_B(y))}).$$

The zip equivalence for polynomial functors is declared in Coq as follows:

```
Definition zip (F : Type -> Type) `{IsPolynomialFunctor F}
  {A B : Type}
  : Pullback
      (pr1 o equivalenceToExtension F A)
      (pr1 o equivalenceToExtension F B)
    <~> F (A * B).
```

5.3 Polymorphic Data Types

Since polymorphic data types are polynomial functors, the zip equivalence defined in the previous section works for any polymorphic data type.

Having given a proof of `IsPolynomialFunctor Tree` in Section 4.2, we obtain a polymorphic zip equivalence

```
zip Tree
  : forall (A : Type) (B : Type)
  , Pullback
      (pr1 o equivalenceToExtension Tree A)
      (pr1 o equivalenceToExtension Tree B)
    <~> Tree (A * B).
```

This means that, for any two types A and B, we have a zip function

```
zip Tree A B
  : Pullback
      (pr1 o equivalenceToExtension Tree A)
      (pr1 o equivalenceToExtension Tree B)
    -> Tree (A * B),
```

an unzip function

```
(zip Tree A B)^-1
  : Tree (A * B)
    -> Pullback
      (pr1 o equivalenceToExtension Tree A)
      (pr1 o equivalenceToExtension Tree B)
```

and two proofs

```
eissect (zip Tree A B)
  : (zip Tree A B)^-1 o zip Tree A B == idmap
```

and

```
eisretr (zip Tree A B)
  : zip Tree A B o (zip Tree A B)^-1 == idmap.
```

Consider, for example, the trees

```
node leaf 2    (node leaf 4     leaf) : Tree NaturalNumber,
node leaf true (node leaf false leaf) : Tree Boolean,
```

which have the same shape

```
node leaf tt (node leaf tt leaf) : Tree NaturalNumber.
```

The command

```
Eval compute in
  zip Tree NaturalNumber Boolean
    ( node leaf 2    (node leaf 4     leaf)
    , node leaf true (node leaf false leaf)
    , idpath
    )
```

returns the tree

```
node leaf (2, true) (node leaf (4, false) leaf).
```

On the other hand, the command

```
Eval compute in
  (zip Tree NaturalNumber Boolean)^-1
    (node leaf (0, false) (node leaf (1, true) leaf))
```

returns the triple

```
( node leaf 0     (node leaf 1    leaf)
, node leaf false (node leaf true leaf)
, ...
).
```

The proof is omitted for brevity.

The complete implementation can be accessed here.

Chapter 6

Traverse

The traverse function performs each of a given list of computations in sequence and collects the results in a list. The function is also known as 'dist' [20] or 'sequence' [29]. The traverse function is generally considered to be a polymorphic function

$$\tau_{F,A} : \mathsf{List}(F(A)) \to F(\mathsf{List}(A)),$$

where A is a type and F is a *lax monoidal functor* from the cartesian monoidal category Type to itself.

The idea of modelling computational effects as monads was originally developed by Moggi [21]. McBride and Paterson [20] demonstrated that many computational effects can be modelled as lax monoidal functors, which are more general than monads. Lax monoidal functors are commonly known as applicative functors in the functional programming literature. Unlike monads, they possess the desirable quality of being closed under composition.

The following section introduces lax monoidal functors and morphisms between them. The traverse function for List is formally defined in Section 6.2, and it is generalized to all *finitary polynomial functors* in Section 6.3.

6.1 Lax Monoidal Functors

Lax monoidal functors are the morphisms between monoidal categories. For our purposes, it is sufficient to consider only lax monoidal functors from the cartesian monoidal category Type to itself. A more detailed treatment of monoidal functors can be found in [3], and the relationship between lax monoidal and applicative functors is explored in [25].

A lax monoidal functor from Type to Type is a functor $F : \mathsf{Type} \to \mathsf{Type}$ together with a function $\eta^F : 1 \to F(1)$ and a natural transformation

$$\mu^F_{A,B} : F(A) \times F(B) \to F(A \times B)$$

that satisfy the unitality and associativity conditions [17]. We do not elaborate on these conditions for the simple reason that we will not need them. The conditions are also omitted from the definition in Coq.

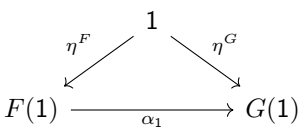
The identity functor $\mathsf{id} : \mathsf{Type} \to \mathsf{Type}$ is a lax monoidal functor:

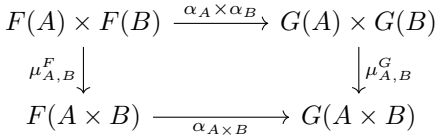
$$\eta^{\mathsf{id}} : 1 \xrightarrow{\mathsf{id}} 1,$$
$$\mu^{\mathsf{id}}_{A,B} : A \times B \xrightarrow{\mathsf{id}} A \times B.$$

In addition, the composite $G \circ F : \mathsf{Type} \to \mathsf{Type}$ of two lax monoidal functors $F, G : \mathsf{Type} \to \mathsf{Type}$ is itself a lax monoidal functor [3]:

$$\eta^{G\circ F} : 1 \xrightarrow{\eta^G} G(1) \xrightarrow{G(\eta^F)} G(F(1)),$$
$$\mu^{G\circ F}_{A,B} : G(F(A)) \times G(F(B)) \xrightarrow{\mu^G} G(F(A) \times F(B)) \xrightarrow{G(\mu^F)} G(F(A \times B)).$$

A morphism $\alpha \colon F \to G$ between two lax monoidal functors from Type to Type, called a *monoidal natural transformation*, is a natural transformation $\alpha \colon F \to G$ that makes the following diagrams commute:





Lax monoidal functors from Type to Type will henceforth be referred to simply as lax monoidal functors.

Example Perhaps the best-known example of a lax monoidal functor is the Maybe monad. If A is a type, the inductive type $\mathsf{Maybe}(A)$ is the one whose constructors are $\mathsf{some} : A \to \mathsf{Maybe}(A)$ and $\mathsf{none} : \mathsf{Maybe}(A)$, indicating the presence and lack of a value, respectively. The functor $\mathsf{Maybe} : \mathsf{Type} \to \mathsf{Type}$ sends a function $f : A \to B$ to the function $\mathsf{Maybe}(f) : \mathsf{Maybe}(A) \to \mathsf{Maybe}(B)$ given by

$$\mathsf{Maybe}(f)(\mathsf{some}(a)) \coloneqq \mathsf{some}(f(a)),$$
$$\mathsf{Maybe}(f)(\mathsf{none}) \coloneqq \mathsf{none}.$$

The function η^{Maybe} is simply the constructor $\mathsf{some} : 1 \to \mathsf{Maybe}(1)$. For any two types A and B, the component

$$\mu^{\mathsf{Maybe}}_{A,B} : \mathsf{Maybe}(A) \times \mathsf{Maybe}(B) \to \mathsf{Maybe}(A \times B)$$

sends a pair of the form $(\mathsf{some}(a), \mathsf{some}(b))$ to $\mathsf{some}(a, b)$ and everything else to none.

6.2 Standard Traverse

The traverse function $\tau_{F,A} : \mathsf{List}(F(A)) \to F(\mathsf{List}(A))$ is given by

$$\tau_{F,A}(\mathsf{nil}) \coloneqq F(\mathsf{nil}')(\eta^F(\star)),$$
$$\tau_{F,A}(\mathsf{cons}(a')(\ell)) \coloneqq F(\mathsf{cons}')(\mu^F(a', \tau_{F,A}(\ell))),$$

where $\mathsf{nil}' : 1 \to \mathsf{List}(A)$ and $\mathsf{cons}' : A \times \mathsf{List}(A) \to \mathsf{List}(A)$ are defined as follows:

$$\begin{aligned} \mathsf{nil}'(\_) &\coloneqq \mathsf{nil}, \\ \mathsf{cons}'(a, \ell) &\coloneqq \mathsf{cons}(a)(\ell). \end{aligned}$$

Before going on to generalize this function, we demonstrate it. Consider the list

$$\mathsf{cons}(\mathsf{some}(1))(\mathsf{cons}(\mathsf{none})(\mathsf{cons}(\mathsf{some}(3))(\mathsf{nil}))) : \mathsf{List}(\mathsf{Maybe}(\mathbb{N})),$$

thought of as containing the results of three computations that may fail. In this case, the first and last computations succeeded and the second failed. Because one of the computations failed, the traverse function sends the list to none, indicating the failure of the entire chain of computations. Now consider the list

$$\mathsf{cons}(\mathsf{some}(1))(\mathsf{cons}(\mathsf{some}(2))(\mathsf{cons}(\mathsf{some}(3))(\mathsf{nil}))) : \mathsf{List}(\mathsf{Maybe}(\mathbb{N})).$$

The traverse function sends this list to $\mathsf{some}(\mathsf{cons}(1)(\mathsf{cons}(2)(\mathsf{cons}(3)(\mathsf{nil}))))$, indicating that the chain of computations succeeded and yielded the list

$$\mathsf{cons}(1)(\mathsf{cons}(2)(\mathsf{cons}(3)(\mathsf{nil})))$$

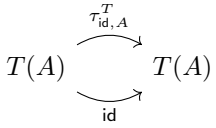
of all the computed values.

6.3 Generic Traverse

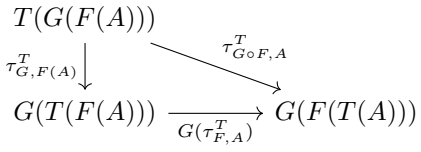
Fokkinga [6] was the first to generalize the traverse function for List to all 'regular' functors. Over time, several definitions of traversability of functors have been proposed [22, 20, 10, 13]. We adopt the definition proposed by Jaskelioff and Rypáček [13], which is the most precise and up to date. According to them, a functor $T : \mathsf{Type} \to \mathsf{Type}$ is said to be *traversable* if there is a polymorphic function

$$\tau^T_{F,A} : T(F(A)) \to F(T(A))$$

that is natural in F and A and makes the following diagrams commute:



Unitarity



Linearity

The corresponding Coq definition is as follows:

```
Class Traversable (T : Type -> Type) `{!IsOFunctor T} :=
  { τ (F : LaxMonoidalFunctor) (A : Type) : T (F A) -> F (T A)
  ; naturality1 (A : Type)
    : Is1Natural
      (T o ApplyLaxMonoidalFunctorTo A)
      (ApplyLaxMonoidalFunctorTo (T A))
      (fun F => τ F A)
  ; naturality2 (F : LaxMonoidalFunctor)
    : Is1Natural (T o F) (F o T) (τ F)
  ; unitarity (A : Type)
    : τ IdentityLaxMonoidalFunctor A == idmap
  ; linearity (F G : LaxMonoidalFunctor) (A : Type)
    : τ (CompositeOfTwoLaxMonoidalFunctors G F) A
      == fmap G (τ F A) o τ G (F A)
  }.
```

Here, `ApplyLaxMonoidalFunctorTo A` denotes the functor that sends each lax monoidal functor F to the type $F(A)$ and each monoidal natural transformation $\alpha\colon F \to G$ to the function $\alpha_A : F(A) \to G(A)$.

It turns out that every finitary polynomial functor is traversable. This was first proved by Jaskelioff and Rypáček [13]. However, they did not provide an implementation, nor does their work directly translate into one. To begin with, they do not define finitary polynomial functors in terms of regular polynomial functors. Using the definition given by them would mean that the generic zip function defined earlier cannot be used for finitary polynomial functors. Secondly, they do not differentiate between a function type $A \to B$ and the $|A|$-fold product $B^{|A|}$. What is more, they blur the distinction between natural equivalence and equality. In conclusion, numerous proofs are missing and their paper is generally lacking in detail.

Using more precise definitions, we will systematically formalize the results presented in [13]. This requires us to prove several additional results.

We will begin by defining finite types and finitary functors. Some important examples of finite types are given, and we describe the conditions under which a polynomial functor is finitary. Next, we formally prove that finitary polynomial functors are traversable. We conclude by demonstrating the generic traverse function.

Finite Types

A finite type is one that contains a finite number of elements. Classically, a set is said to be finite if it is isomorphic to the set $\{k \in \mathbb{N} \mid k < n\}$ for some natural number n [33]. Finite types are defined in a similar way. The set $\{k \in \mathbb{N} \mid k < n\}$ corresponds to the *standard finite type* $\mathsf{Fin}(n)$, where

$\mathsf{Fin} : \mathbb{N} \to \mathsf{Type}$ is given by

$$\begin{aligned} \mathsf{Fin}(0) &\coloneqq 0, \\ \mathsf{Fin}(\mathsf{succ}(n)) &\coloneqq \mathsf{Fin}(n) + 1. \end{aligned}$$

A type is said to be finite if it is equivalent to $\mathsf{Fin}(n)$ for some natural number n:

```
Class IsFiniteType (A : Type) :=
  { cardinality : NaturalNumber
  ; equivalenceToStandardFiniteType : A <~> Fin cardinality
  }.
```

Standard Finite Types

Obviously, every standard finite type is finite because it is equivalent to itself. This includes the empty type $0 \equiv \mathsf{Fin}(0)$.

Types Equivalent to a Finite Type

Every type A that is equivalent to a finite type B is itself finite. If $B \simeq \mathsf{Fin}(n)$, then $A \simeq B \simeq \mathsf{Fin}(n)$.

Unit Type

The unit type 1 is finite because $1 \simeq 0 + 1 \equiv \mathsf{Fin}(1)$.

Binary Coproducts of Finite Types

The coproduct $A + B$ of two finite types A and B is itself finite. If $A \simeq \mathsf{Fin}(m)$ and $B \simeq \mathsf{Fin}(n)$, then

$$A + B \simeq \mathsf{Fin}(m) + \mathsf{Fin}(n) \simeq \mathsf{Fin}(m + n).$$

We prove that $\mathsf{Fin}(m) + \mathsf{Fin}(n) \simeq \mathsf{Fin}(m + n)$ for any two natural numbers m and n by induction on m. If $m \equiv 0$, then

$$\mathsf{Fin}(0) + \mathsf{Fin}(n) \equiv 0 + \mathsf{Fin}(n) \simeq \mathsf{Fin}(n).$$

If $m \equiv \mathsf{succ}(m')$, on the other hand, then

$$\begin{aligned} & \mathsf{Fin}(\mathsf{succ}(m')) + \mathsf{Fin}(n) \\ \equiv\ & (\mathsf{Fin}(m') + 1) + \mathsf{Fin}(n) \\ \simeq\ & \mathsf{Fin}(m') + (1 + \mathsf{Fin}(n)) \\ \simeq\ & \mathsf{Fin}(m') + (\mathsf{Fin}(n) + 1) \\ \simeq\ & (\mathsf{Fin}(m') + \mathsf{Fin}(n)) + 1 \\ \simeq\ & \mathsf{Fin}(m' + n) + 1 \\ \equiv\ & \mathsf{Fin}(\mathsf{succ}(m' + n)) \\ \equiv\ & \mathsf{Fin}(\mathsf{succ}(m') + n). \end{aligned}$$

Finitary Functors

Strictly speaking, functors are said to be *finitary* if they preserve all 'filtered colimits'. Fortunately, it is much easier to define when a polynomial functor is finitary. A polynomial functor is finitary if and only if it is naturally equivalent to the extension of a *finite polynomial* [8]. A polynomial p is said to be finite if the type $\vec{p}(i)$ is finite for every position $i : \overset{\bullet}{p}$ [8, 32]:

```
Class IsFinitePolynomial (p : Polynomial) :=
  DirectionIsFiniteType i :> IsFiniteType (Direction p i).
```

It is important to recall that polymorphic data types are simply inductive types with a single type parameter. A polymorphic data type is a finitary polynomial functor if it is a *finitary inductive type*, as defined by Kraus and Sattler [16, 30]. Finitary inductive types are often referred to as 'regular' functors in the computer science literature [22].

A good example of a finitary polynomial functor is Tree. As mentioned in Section 4.2, Tree is naturally equivalent to the extension of the polynomial Tree'. It is easy to show that $\vec{\mathsf{Tree}'}(t)$ is finite for every tree $t : \mathsf{Tree}(1)$ by induction on t:

```
Instance TreePolynomialIsFinitePolynomial
  : IsFinitePolynomial Tree'.
Proof.
  intro t; induction t; exact _.
Defined.
```

Coq is clever enough to infer the proofs.

Traversability

We are now ready to prove that finitary polynomial functors are traversable. As shown below, any functor that is naturally equivalent to a traversable functor is itself traversable. A finitary polynomial functor is, by definition, naturally equivalent to the extension $\sum_{i:\overset{\bullet}{p}} y^{\vec{p}(i)}$ of a finite polynomial p. Since traversable functors are closed under coproducts (see below), it suffices to show that $y^{\vec{p}(i)}$ is traversable for every position $i : \overset{\bullet}{p}$. The representable functor $y^{\vec{p}(i)}$ is naturally equivalent to $y^{\mathsf{Fin}(|\vec{p}(i)|)}$ because p is a finite polynomial. We can prove by induction that $y^{\mathsf{Fin}(n)}$ is traversable for every natural number n. The functor $y^{\mathsf{Fin}(0)} \equiv y^0$ is naturally equivalent to the constant functor $A \mapsto 1$, denoted by $\Delta(1)$. This is explained by the fact that there is exactly one function from the empty type to a given type. We will show below that the constant functor at the unit type is traversable. According to the universal property of the coproduct, the functor $y^{\mathsf{Fin}(\mathsf{succ}(n))} \equiv y^{\mathsf{Fin}(n)+1}$ is naturally equivalent to the product $y^{\mathsf{Fin}(n)} \times y^1$. Traversable functors are also closed under binary products (see below). By induction, $y^{\mathsf{Fin}(n)}$ is traversable. The functor y^1 is

naturally equivalent to the identity functor: a function from the unit type to a type is the same thing as an element of that type. Below, we will show that the identity functor is traversable.

After proving the necessary results mentioned above, the generic traverse function is demonstrated.

Functors Naturally Equivalent to a Traversable Functor

Let $\phi : \prod_{A:\mathsf{Type}}(S(A) \simeq T(A))$ be a natural equivalence between two functors $S, T : \mathsf{Type} \to \mathsf{Type}$. If T is traversable, then so is S.

The traverse function for S is the composite

$$\tau^S_{F,A} : S(F(A)) \xrightarrow{\phi_{F(A)}} T(F(A)) \xrightarrow{\tau^T_{F,A}} F(T(A)) \xrightarrow{F(\phi_A^{-1})} F(S(A)).$$

It is natural in F because the following diagram commutes for every monoidal natural transformation $\alpha\colon F \to G$:

$$\begin{array}{ccccccc}
S(F(A)) & \xrightarrow{\phi_{F(A)}} & T(F(A)) & \xrightarrow{\tau^T_{F,A}} & F(T(A)) & \xrightarrow{F(\phi_A^{-1})} & F(S(A)) \\
{\scriptstyle S(\alpha_A)}\downarrow & & \downarrow{\scriptstyle T(\alpha_A)} & & \downarrow{\scriptstyle \alpha_{T(A)}} & & \downarrow{\scriptstyle \alpha_{S(A)}} \\
S(G(A)) & \xrightarrow[\phi_{G(A)}]{} & T(G(A)) & \xrightarrow[\tau^T_{G,A}]{} & G(T(A)) & \xrightarrow[G(\phi_A^{-1})]{} & G(S(A))
\end{array}$$

The diagram below, which commutes for every function $f : A \to B$, shows that $\tau^S_{F,A}$ is also natural in A.

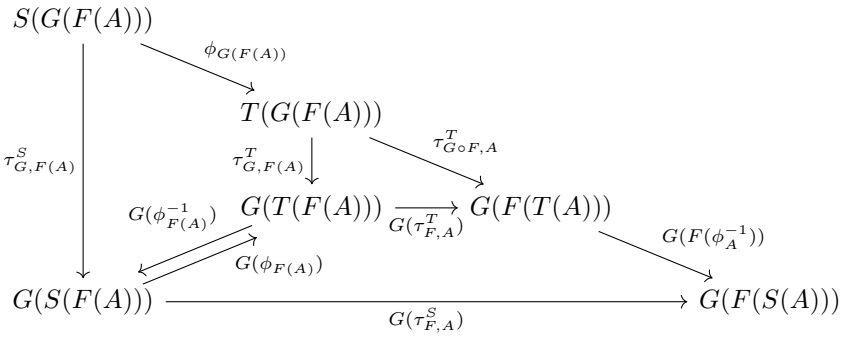
$$\begin{array}{ccccccc}
S(F(A)) & \xrightarrow{\phi_{F(A)}} & T(F(A)) & \xrightarrow{\tau^T_{F,A}} & F(T(A)) & \xrightarrow{F(\phi_A^{-1})} & F(S(A)) \\
{\scriptstyle S(F(f))}\downarrow & & \downarrow{\scriptstyle T(F(f))} & & \downarrow{\scriptstyle F(T(f))} & & \downarrow{\scriptstyle F(S(f))} \\
S(F(B)) & \xrightarrow[\phi_{F(B)}]{} & T(F(B)) & \xrightarrow[\tau^T_{F,B}]{} & F(T(B)) & \xrightarrow[F(\phi_B^{-1})]{} & F(S(B))
\end{array}$$

The traverse function for S is unitary because

$$\begin{aligned}
\tau^S_{\mathsf{id},A} &\equiv \phi_A^{-1} \circ \tau^T_{\mathsf{id},A} \circ \phi_A \\
&\sim \phi_A^{-1} \circ \mathsf{id} \circ \phi_A \\
&\equiv \phi_A^{-1} \circ \phi_A \\
&\sim \mathsf{id}.
\end{aligned}$$

The linearity of the traverse function for S is illustrated by the following

commutative diagram:



Identity Functor

The identity functor $\mathsf{id} : \mathsf{Type} \to \mathsf{Type}$ is traversable. The traverse function for it is simply the identity

$$\tau^{\mathsf{id}}_{F,A} : \mathsf{id}(F(A)) \equiv F(A) \xrightarrow{\mathsf{id}} F(A) \equiv F(\mathsf{id}(A)).$$

It is clear that this function is natural in both F and A. The traverse function for id is, by definition, unitary, and it is linear because

$$\begin{aligned}\tau^{\mathsf{id}}_{G\circ F,A} &\equiv \mathsf{id} \\ &\sim G(\mathsf{id}) \\ &\equiv G(\mathsf{id}) \circ \mathsf{id} \\ &\equiv G(\tau^{\mathsf{id}}_{F,A}) \circ \tau^{\mathsf{id}}_{G\circ F,A}.\end{aligned}$$

Constant Functor at the Unit Type

The constant functor $\Delta(1) : \mathsf{Type} \to \mathsf{Type}$ is traversable. The traverse function for it is simply the function

$$\tau^{\Delta(1)}_{F,A} : \Delta(1)(F(A)) \equiv 1 \xrightarrow{\eta^F} F(1) \equiv F(\Delta(1)(A)).$$

This function is trivially natural in A, and it is natural in F by the definition of a monoidal natural transformation (see Section 6.1). The traverse function for $\Delta(1)$ is unitary because

$$\tau^{\Delta(1)}_{\mathsf{id},A} \equiv \eta^{\mathsf{id}} \equiv \mathsf{id},$$

and it is linear because

$$\begin{aligned}\tau^{\Delta(1)}_{G\circ F,A} &\equiv \eta^{G\circ F} \\ &\equiv G(\eta^F) \circ \eta^G \\ &\equiv G(\tau^{\Delta(1)}_{F,A}) \circ \tau^{\Delta(1)}_{G\circ F,A}.\end{aligned}$$

Binary Products of Traversable Functors

The product $S \times T$ of two traversable functors S and T is itself traversable.

The traverse function for $S \times T$ is the composite

$$\tau_{F,A}^{S\times T} : S(F(A))\times T(F(A)) \xrightarrow{\tau_{F,A}^{S}\times\tau_{F,A}^{T}} F(S(A))\times F(T(A)) \xrightarrow{\mu^{F}} F(S(A)\times T(A)).$$

It is natural in F because the diagram below commutes for every monoidal natural transformation $\alpha\colon F \to G$.

$$\begin{array}{ccccc}
S(F(A)) \times T(F(A)) & \xrightarrow{\tau_{F,A}^{S}\times\tau_{F,A}^{T}} & F(S(A)) \times F(T(A)) & \xrightarrow{\mu^{F}} & F(S(A) \times T(A)) \\
\downarrow{\scriptstyle S(\alpha_A)\times T(\alpha_A)} & & \downarrow{\scriptstyle \alpha_{S(A)}\times\alpha_{T(A)}} & & \downarrow{\scriptstyle \alpha_{S(A)\times T(A)}} \\
S(G(A)) \times T(G(A)) & \xrightarrow[\tau_{G,A}^{S}\times\tau_{G,A}^{T}]{} & G(S(A)) \times G(T(A)) & \xrightarrow[\mu^{G}]{} & G(S(A) \times T(A))
\end{array}$$

The right-hand square commutes by the definition of a monoidal natural transformation (see Section 6.1).

The function is also natural in A, since the following diagram commutes for every function $f : A \to B$:

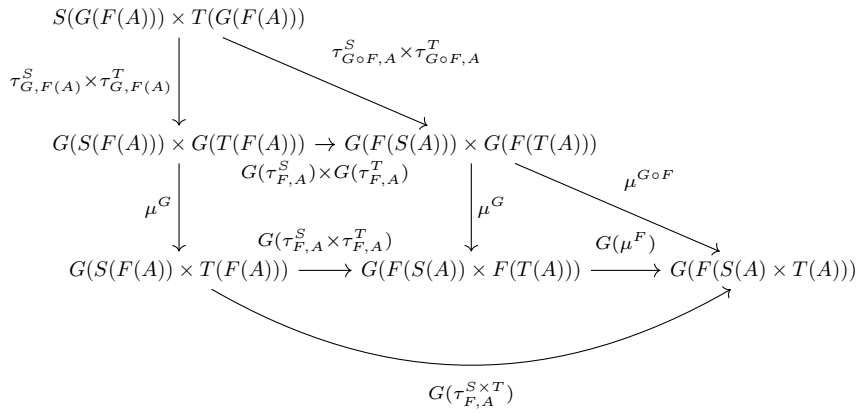
$$\begin{array}{ccccc}
S(F(A)) \times T(F(A)) & \xrightarrow{\tau_{F,A}^{S}\times\tau_{F,A}^{T}} & F(S(A)) \times F(T(A)) & \xrightarrow{\mu^{F}} & F(S(A) \times T(A)) \\
\downarrow{\scriptstyle S(F(f))\times T(F(f))} & & \downarrow{\scriptstyle F(S(f))\times F(T(f))} & & \downarrow{\scriptstyle F(S(f)\times T(f))} \\
S(F(B)) \times T(F(B)) & \xrightarrow[\tau_{F,B}^{S}\times\tau_{F,B}^{T}]{} & F(S(B)) \times F(T(B)) & \xrightarrow[\mu^{F}]{} & F(S(B) \times T(B))
\end{array}$$

The traverse function for $S \times T$ is unitary because

$$\begin{aligned}
\tau_{\mathsf{id},A}^{S\times T} &\equiv \mu^{\mathsf{id}} \circ (\tau_{\mathsf{id},A}^{S} \times \tau_{\mathsf{id},A}^{T}) \\
&\equiv \mathsf{id} \circ (\tau_{\mathsf{id},A}^{S} \times \tau_{\mathsf{id},A}^{T}) \\
&\equiv \tau_{\mathsf{id},A}^{S} \times \tau_{\mathsf{id},A}^{T} \\
&\sim \mathsf{id} \times \mathsf{id} \\
&\equiv \mathsf{id}.
\end{aligned}$$

The commutative diagram below illustrates the linearity of the traverse

function for $S \times T$.



Coproducts of Traversable Functors

The coproduct $\sum_{i:I} T(i)$ of a family $T : I \to (\mathsf{Type} \to \mathsf{Type})$ of traversable functors is itself traversable.

For every $i : I$, there is a function

$$T(i)(F(A)) \xrightarrow{\tau^{T(i)}_{F,A}} F(T(i)(A)) \xrightarrow{F((i,-))} F\Big(\sum_{i:I} T(i)(A)\Big),$$

so the traverse function for $\sum_{i:I} T(i)$ can be defined as the induced function

$$\tau^{\sum_{i:I} T(i)}_{F,A} : \sum_{i:I} T(i)(F(A)) \xrightarrow{[i\mapsto F((i,-))\circ \tau^{T(i)}_{F,A}]} F\Big(\sum_{i:I} T(i)(A)\Big).$$

This function is natural in F if

$$\alpha_{\sum_{i:I} T(i)(A)} \circ \tau^{\sum_{i:I} T(i)}_{F,A} \sim \tau^{\sum_{i:I} T(i)}_{G,A} \circ \sum_{i:I} T(i)(\alpha_A)$$

for every monoidal natural transformation $\alpha\colon F \to G$. As described in Section 2.7, it suffices to show that

$$\alpha_{\sum_{i:I} T(i)(A)} \circ \tau^{\sum_{i:I} T(i)}_{F,A} \circ (i,-) \sim \tau^{\sum_{i:I} T(i)}_{G,A} \circ \sum_{i:I} T(i)(\alpha_A) \circ (i,-)$$

for every $i : I$. A proof is given below.

$$\begin{aligned}
&\alpha_{\sum_{i:I} T(i)(A)} \circ \tau_{F,A}^{\sum_{i:I} T(i)} \circ (i, -) \\
\equiv\ &\alpha_{\sum_{i:I} T(i)(A)} \circ [i \mapsto F((i, -)) \circ \tau_{F,A}^{T(i)}] \circ (i, -) \\
\equiv\ &\alpha_{\sum_{i:I} T(i)(A)} \circ F((i, -)) \circ \tau_{F,A}^{T(i)} \\
\sim\ &G((i, -)) \circ \alpha_{T(i)(A)} \circ \tau_{F,A}^{T(i)} \\
\sim\ &G((i, -)) \circ \tau_{G,A}^{T(i)} \circ T(i)(\alpha_A) \\
\equiv\ &[i \mapsto G((i, -)) \circ \tau_{G,A}^{T(i)}] \circ (i, -) \circ T(i)(\alpha_A) \\
\equiv\ &[i \mapsto G((i, -)) \circ \tau_{G,A}^{T(i)}] \circ \sum_{i:I} T(i)(\alpha_A) \circ (i, -) \\
\equiv\ &\tau_{G,A}^{\sum_{i:I} T(i)} \circ \sum_{i:I} T(i)(\alpha_A) \circ (i, -).
\end{aligned}$$

The function $\tau_{F,A}^{\sum_{i:I} T(i)}$ is also natural in A, since

$$\begin{aligned}
&F\Big(\sum_{i:I} T(i)(f)\Big) \circ \tau_{F,A}^{\sum_{i:I} T(i)} \circ (i, -) \\
\equiv\ &F\Big(\sum_{i:I} T(i)(f)\Big) \circ [i \mapsto F((i, -)) \circ \tau_{F,A}^{T(i)}] \circ (i, -) \\
\equiv\ &F\Big(\sum_{i:I} T(i)(f)\Big) \circ F((i, -)) \circ \tau_{F,A}^{T(i)} \\
\sim\ &F\Big(\sum_{i:I} T(i)(f) \circ (i, -)\Big) \circ \tau_{F,A}^{T(i)} \\
\equiv\ &F((i, -) \circ T(i)(f)) \circ \tau_{F,A}^{T(i)} \\
\sim\ &F((i, -)) \circ F(T(i)(f)) \circ \tau_{F,A}^{T(i)} \\
\sim\ &F((i, -)) \circ \tau_{F,B}^{T(i)} \circ T(i)(F(f)) \\
\equiv\ &[i \mapsto F((i, -)) \circ \tau_{F,B}^{T(i)}] \circ (i, -) \circ T(i)(F(f)) \\
\equiv\ &[i \mapsto F((i, -)) \circ \tau_{F,B}^{T(i)}] \circ \sum_{i:I} T(i)(F(f)) \circ (i, -) \\
\equiv\ &\tau_{F,B}^{\sum_{i:I} T(i)} \circ \sum_{i:I} T(i)(F(f)) \circ (i, -)
\end{aligned}$$

for every $f : A \to B$ and $i : I$.

The traverse function for $\sum_{i:I} T(i)$ is unitary because

$$\begin{aligned}
\tau_{\mathsf{id},A}^{\sum_{i:I} T(i)} \circ (i, -) &\equiv [i \mapsto (i, -) \circ \tau_{\mathsf{id},A}^{T(i)}] \circ (i, -) \\
&\equiv (i, -) \circ \tau_{\mathsf{id},A}^{T(i)} \\
&\sim (i, -) \circ \mathsf{id} \\
&\equiv \mathsf{id} \circ (i, -)
\end{aligned}$$

for every $i : I$, and it is linear because

$$\begin{aligned}
&\tau^{\sum_{i:I} T(i)}_{G\circ F,A} \circ (i, -) \\
\equiv\ &[i \mapsto G(F((i, -))) \circ \tau^{T(i)}_{G\circ F,A}] \circ (i, -) \\
\equiv\ &G(F((i, -))) \circ \tau^{T(i)}_{G\circ F,A} \\
\sim\ &G(F((i, -))) \circ G(\tau^{T(i)}_{F,A}) \circ \tau^{T(i)}_{G,F(A)} \\
\sim\ &G(F((i, -)) \circ \tau^{T(i)}_{F,A}) \circ \tau^{T(i)}_{G,F(A)} \\
\equiv\ &G([i \mapsto F((i, -)) \circ \tau^{T(i)}_{F,A}] \circ (i, -)) \circ \tau^{T(i)}_{G,F(A)} \\
\sim\ &G([i \mapsto F((i, -)) \circ \tau^{T(i)}_{F,A}]) \circ G((i, -)) \circ \tau^{T(i)}_{G,F(A)} \\
\equiv\ &G([i \mapsto F((i, -)) \circ \tau^{T(i)}_{F,A}]) \circ [i \mapsto G((i, -)) \circ \tau^{T(i)}_{G,F(A)}] \circ (i, -) \\
\equiv\ &G(\tau^{\sum_{i:I} T(i)}_{F,A}) \circ \tau^{\sum_{i:I} T(i)}_{G,F(A)} \circ (i, -)
\end{aligned}$$

for every $i : I$.

Finitary Polymorphic Data Types

Let T be a finitary polymorphic data type. To obtain an element of

`Traversable T`,

one has to specify an element of `IsPolynomialFunctor T` and an element of `IsFinitePolynomial (polynomial T)`. Consider, for example, the finitary polymorphic data type Tree. Having given an element of

`IsPolynomialFunctor Tree`

in Section 4.2 and an element of `IsFinitePolynomial Tree'` in Section 6.3, we obtain a traverse function

```
τ Tree
  : forall (F : LaxMonoidalFunctor) (A : Type)
  , Tree (F A) -> F (Tree A),
```

along with the following proofs:

```
naturality1 Tree
  : forall A : Type
  , Is1Natural
    (Tree o ApplyLaxMonoidalFunctorTo A)
    (ApplyLaxMonoidalFunctorTo (Tree A))
    (fun F => τ Tree F A),
naturality2 Tree
  : forall F : LaxMonoidalFunctor
  , Is1Natural (Tree o F) (F o Tree) (τ Tree F),
```

```
unitarity Tree
  : forall A : Type
  , τ Tree IdentityLaxMonoidalFunctor A == idmap,
linearity Tree
  : forall (F G : LaxMonoidalFunctor) (A : Type)
  , τ Tree (CompositeOfTwoLaxMonoidalFunctors G F) A
    == fmap G (τ Tree F A) o τ Tree G (F A).
```

The command

```
Eval compute in
  τ Tree MaybeLaxMonoidalFunctor NaturalNumber (
    node
      (node leaf (some 1) (node leaf (some 2) leaf))
      (some 3)
      leaf
  ),
```

to take one example, returns the value

```
some (node (node leaf 1 (node leaf 2 leaf)) 3 leaf).
```

On the other hand,

```
Eval compute in
  τ Tree MaybeLaxMonoidalFunctor NaturalNumber (
    node
      (node leaf (some 1) (node leaf none leaf))
      (some 3)
      leaf
  )
```

returns `none`.

See GenericTraverse.v for the complete implementation.

Chapter 7

Discussion

We successfully implemented a zip equivalence for polynomial functors. Such an equivalence consists of a zip function, an unzip function and a proof that they are mutually inverse. We also proved in Coq that finitary polynomial functors, such as List and Tree, are traversable. This involved defining a traverse function and proving its naturality, unitarity and linearity. As far as we are aware, these implementations are the first of their kind.

Hoogendijk and Backhouse [11] took a relational approach to generalizing the zip equivalence. In stark contrast to ours, their results are very theoretical in nature and are therefore difficult, if not impossible, to implement.

The use of a proof assistant such as Coq enabled us not only to define a generic zip function and a generic traverse function but also to prove that they possess the expected properties. Surprisingly, it seems that very few researchers prove the correctness of the generic functions they define. A notable exception is [26].

The function extensionality axiom is required to prove that List is a polynomial functor. On the other hand, the zip equivalence for List defined in Chapter 3 works perfectly well without it. Moreover, the corresponding unzip function is guaranteed to return refl as part of the output, unlike the generic one. This suggests that there may be a way to define a generic zip equivalence that works without the function extensionality axiom and whose inverse always returns refl. Future research should explore this possibility. That said, using two different approaches to generic programming is less than ideal.

We should be able to fully automate the process of giving a proof of `IsPolynomialFunctor T` by drawing on the work of Avigad et al. [4]. If T is a finitary inductive type, the type of directions at a given position in `polynomial T` is an inductively defined sum in which any term is either the empty type or the unit type (see, for example, Section 4.2). Having given proofs of `IsFiniteType Empty`, `IsFiniteType Unit` and

```
forall A B : Type
  , IsFiniteType A -> IsFiniteType B -> IsFiniteType (A + B)
```

in Section 6.3, it follows by induction that `polynomial T` is finite:

```
Instance TPolynomialIsFinitePolynomial
  : IsFinitePolynomial (polynomial T).
Proof.
  intro t; induction t; exact _.
Defined.
```

Although certainly possible, automating this further is a low priority.

There is ample opportunity to generalize the generic zip equivalence even further. The zip function can be seen as an instance of the 'zipWith' function, described in [7]. However, a quick look reveals that a generic zipWith function can easily be defined in terms of the generic zip function. Weirich and Casinghino [35] discuss yet another generalization of the zip equivalence: the n-ary zip equivalence. For any natural number n, there is an equivalence between the type of n-tuples of lists of equal length and the type of lists of n-tuples, not just for $n \equiv 2$. Like the regular one, the n-ary zip equivalence can be generalized to all polynomial functors. This claim, though reasonable in itself, can be justified mathematically. Polynomial functors preserve not just pullbacks but all *connected limits* [8, 34]. In particular, they preserve *wide pullbacks*, of which ordinary pullbacks are special cases. A wide pullback is the limit of a family of morphisms with the same codomain (considered as a diagram) [24]. Binary wide pullbacks are the same as ordinary pullbacks. Let $(A_i)_{i \in I}$ be a family of sets indexed by a set I. The wide pullback of the family $(A_i \to 1)_{i \in I}$ of functions to the terminal object is simply the product $\prod_{i \in I} A_i$. For any polynomial functor $F \colon \mathbf{Set} \to \mathbf{Set}$, there is an isomorphism between $F(\prod_{i \in I} A_i)$ and the wide pullback of $(F(A_i) \to F(1))_{i \in I}$, since F preserves the wide pullback of $(A_i \to 1)_{i \in I}$. An element of the wide pullback of $(F(A_i) \to F(1))_{i \in I}$ is a tuple $(x_i)_{i \in I} \in \prod_{i \in I} F(A_i)$ such that $F(a \mapsto \star)(x_i) = F(a \mapsto \star)(x_j)$ for any two indices $i, j \in I$. In other words, all components of $(x_i)_{i \in I}$ have the same shape.

Theoretically, both the generic zip equivalence and the generic traverse function can be generalized to polynomial functors in many variables, although the benefits of doing so are not immediately clear.

Bibliography

[1] Michael Gordon Abbott. "Categories of Containers". PhD thesis. University of Leicester, 2003.

[2] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. "Containers: Constructing strictly positive types". In: *Theoretical Computer Science* 342.1 (2005).

[3] Marcelo Aguiar and Swapneel Mahajan. *Monoidal Functors, Species and Hopf Algebras*. American Mathematical Society, 2010. ISBN: 978-0821847763.

[4] Jeremy Avigad, Mario Carneiro, and Simon Hudon. "Data Types as Quotients of Polynomial Functors". In: *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Ed. by John Harrison, John O'Leary, and Andrew Tolmach. Vol. 141. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 6:1–6:19. ISBN: 978-3959771221.

[5] Roland Carl Backhouse et al. "Generic Programming: An Introduction". In: *Advanced Functional Programming: Third International School, AFP'98, Braga, Portugal, September 12-19, 1998, Revised Lectures*. Ed. by S. Doaitse Swierstra, Pedro Rangel Henriques, and José Nuno Oliveira. Vol. 1608. Lecture Notes in Computer Science. Springer, Jan. 1998, pp. 28–115.

[6] Maarten M. Fokkinga. "Monadic Maps and Folds for Arbitrary Datatypes". In: *Memoranda Informatica* 94-28 (June 1994). ISSN: 0924-3755.

[7] Daniel Fridlender and Mia Indrika. "Do we need dependent types?" In: *Journal of Functional Programming* 10.4 (July 2000), pp. 409–415. ISSN: 0956-7968. DOI: 10.1017/S0956796800003658.

[8] Nicola Gambino and Joachim Kock. "Polynomial functors and polynomial monads". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 154.1 (2013).

[9] Richard Garner. *Polynomials in Categories With Pullbacks*. Workshop on Polynomial Functors. Topos Institute, Mar. 2021. URL: https://youtu.be/B8STLcbEGrE?t=7942.

[10] Jeremy Gibbons and Bruno C. d. S. Oliveira. "The essence of the Iterator pattern". In: *Journal of Functional Programming* 19.3-4 (2009), pp. 377–402.

[11] Paul Hoogendijk and Roland Backhouse. "When do datatypes commute?" In: *Category Theory and Computer Science*. Ed. by Eugenio Moggi and Giuseppe Rosolini. Vol. 1290. Lecture Notes in Computer Sciece. Springer, 1997, pp. 242–260.

[12] Mauro Jaskelioff and Russell O'Connor. "A representation theorem for second-order functionals". In: *Journal of Functional Programming* 25 (Sept. 2015).

[13] Mauro Jaskelioff and Ondřej Rypáček. "An Investigation of the Laws of Traversals". In: *Proceedings Fourth Workshop on Mathematically Structured Functional Programming*. Ed. by James Chapman and Paul Blain Levy. Vol. 76. Electronic Proceedings in Theoretical Computer Science. Mar. 2012, pp. 40–49.

[14] Barry Jay and Robin Cockett. "Shapely Types and Shape Polymorphism". In: *Programming Languages and Systems — ESOP '94*. Ed. by Donald Sannella. Vol. 788. Lecture Notes in Computer Science. Heidelberg, Germany: Springer, 1994, pp. 302–316.

[15] Joachim Kock. "Notes on Polynomial Functors". URL: https://mat.uab.cat/~kock/cat/polynomial.pdf.

[16] Nicolai Kraus and Christian Sattler. *Isomorphism of Finitary Inductive Types*. URL: https://www.cs.nott.ac.uk/~psznk/docs/inductive-isomorphism.pdf.

[17] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1971.

[18] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic*. Springer, 1994.

[19] Aaron D. Lauda and Joshua Sussan. "An Invitation to Categorification". In: *Notices of the American Mathematical Society* 69.1 (Jan. 2022).

[20] Conor McBride and Ross Paterson. "Applicative programming with effects". In: *Journal of Functional Programming* 18.1 (2008), pp. 1–13.

[21] Eugenio Moggi. "Notions of computation and monads". In: *Information and Computation* 93.1 (1991), pp. 55–92.

[22] Eugenio Moggi, Gianna Bellè, and Barry Jay. "Monads, Shapely Functors and Traversals". In: *Category Theory and Computer Science*. Ed. by Martin Hofmann, Giuseppe Rosolini, and Dusko Pavlovic. Vol. 29. Electronic Notes in Theoretical Computer Science. Elsevier, 1999, pp. 187–208.

[23] Peter Morris. "Constructing Universes for Generic Programming". PhD thesis. University of Nottingham, 2007.

[24] Robert Paré. "Simply Connected Limits". In: *Canadian Journal of Mathematics* 42.4 (1990), pp. 731–746.

[25] Ross Paterson. "Constructing Applicative Functors". In: *Mathematics of Program Construction*. Ed. by Jeremy Gibbons and Pablo Nogueira. Heidelberg, Germany: Springer, 2012, pp. 300–323. ISBN: 978-3642311130.

[26] Rawle Prince, Neil Ghani, and Conor McBride. "Proving Properties about Lists Using Containers". In: *Functional and Logic Programming*. Ed. by Jacques Garrigue and Manuel Hermenegildo. Heidelberg, Germany: Springer, 2008, pp. 97–112. ISBN: 978-3540789697.

[27] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, 2013.

[28] Egbert Rijke. *Introduction to Homotopy Type Theory*. arXiv, 2022. URL: https://arxiv.org/abs/2212.11082.

[29] Ondřej Rypáček. "Distributive Laws in Programming Structures". PhD thesis. University of Nottingham, 2010.

[30] Christian Sattler. "On the complexities of polymorphic stream equation systems, isomorphism of finitary inductive types, and higher homotopies in univalent universes". PhD thesis. University of Nottingham, 2015. URL: http://eprints.nottingham.ac.uk/28111/.

[31] Michael Shulman. "Homotopy Type Theory: The Logic of Space". In: *New Spaces in Mathematics: Formal and Conceptual Reflections*. Ed. by Mathieu Anel and Gabriel Catren. Vol. 1. Cambridge University Press, 2021, pp. 322–404.

[32] David Isaac Spivak. *A summary of categorical structures in* **Poly**. 2022. URL: https://arxiv.org/abs/2202.00534.

[33] David Isaac Spivak. *Category Theory for the Sciences*. MIT Press, 2014. ISBN: 978-0262028134.

[34] David Isaac Spivak and Nelson Niu. "Polynomial Functors: A General Theory of Interaction".

[35] Stephanie Weirich and Chris Casinghino. "Arity-Generic Datatype-Generic Programming". In: *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages Meets Program Verification*. PLPV '10. New York, NY: Association for Computing Machinery, 2010, pp. 15–26. ISBN: 978-1605588902. DOI: 10.1145/1707790.1707799.