Reason Isomorphically!

Ralf Hinze Daniel W. H. James

Computing Laboratory, University of Oxford,
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
{ralf.hinze, daniel.james}@comlab.ox.ac.uk

Abstract

When are two types the same? In this paper we argue that isomorphism is a more useful notion than equality. We explain a succinct and elegant approach to establishing isomorphisms, with our focus on showing their existence over deriving the witnesses. We use category theory as a framework, but rather than chasing diagrams or arguing with arrows, we present our proofs in a calculational style. In particular, we hope to showcase to the reader why the Yoneda lemma and adjunctions should be in their reasoning toolbox.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.4 [*Software/ Program Verification*]: Correctness Proofs; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Specification techniques

General Terms Theory, Verification

Keywords adjunctions, category theory, isomorphism, Yoneda lemma

1. Introduction

Generic programming is about making programs more adaptable by making them more general. In this paper we embrace and extend this slogan: generic *reasoning* is about making *proofs* more adaptable by making them more general. Our 'generic reasoning' is performed in the setting of category theory with our 'adaptability' coming from the fact that our proofs are category agnostic. The mention of category theory should be no cause for alarm: our flavour is functor focused, an unfearsome concept for generic programmers.

Let us start with our feet firmly on the ground. Here are two simple inductive definitions for rose trees and binary trees.

$$\begin{aligned}&\mathbf{data}\,\mathsf{Rose}\;a = \mathit{Rose}\,(a, [\mathsf{Rose}\;a])\\ &\mathbf{data}\,\mathsf{Tree}\;a = \mathit{Empty} \mid \mathit{Node}\,(a, \mathsf{Tree}\;a, \mathsf{Tree}\;a)\end{aligned}$$

Forests of rose trees and binary trees are in a so-called natural correspondence [14, p. 334–335], illustrated in Figure 1. We can represent any forest as a binary tree, where the left child represents the subtrees of the first rose tree, and the right child its siblings. The diagram also illustrates a second correspondence, one between topped binary trees, sometimes called pennants, and rose trees.

WGP '10 September 26, 2010, Baltimore, Maryland, USA.

Reprinted from WGP '10, Proceedings of the ACM SIGPLAN workshop on Generic programming, September 26, 2010, Baltimore, Maryland, USA., pp. 85–96.

sums

$$X + 0 \cong X \quad X + Y \cong Y + X$$
$$(X + Y) + Z \cong X + (Y + Z)$$

products

$$X \times 1 \cong X \quad 0 \times X \cong 0 \quad X \times Y \cong Y \times X$$
$$(X \times Y) \times Z \cong X \times (Y \times Z)$$

product over sum

$$(X + Y) \times Z \cong (X \times Z) + (Y \times Z)$$

exponentials

$$1^X \cong 1 \quad X^0 \cong 1 \quad X^1 \cong X$$
$$Z^{X+Y} \cong Z^X \times Z^Y \quad (Z^Y)^X \cong Z^{X \times Y}$$

exponential over product

$$(Y \times Z)^X \cong Y^X \times Z^X$$

Table 1. Laws of high-school algebra

We can implement the isomorphism $\mathsf{Tree}\;a \cong \mathsf{List}\,(\mathsf{Rose}\;a)$ by giving two functions that convert to and fro,

$$\begin{aligned}&\mathit{toTree} : [\mathsf{Rose}\;a] && \to \mathsf{Tree}\;a\\ &\mathit{toTree}\;\;[\,] && = \mathit{Empty}\\ &\mathit{toTree}\;\;(\mathit{Rose}\,(a, ts) : us) && = \mathit{Node}\,(a, \mathit{toTree}\;ts, \mathit{toTree}\;us)\\ &\mathit{toForest} : \mathsf{Tree}\;a && \to [\mathsf{Rose}\;a]\\ &\mathit{toForest}\;\;\mathit{Empty} && = [\,]\\ &\mathit{toForest}\;\;(\mathit{Node}\,(a, l, r)) && = \mathit{Rose}\,(a, \mathit{toForest}\;l) : \mathit{toForest}\;r\end{aligned}$$

and furthermore, two proofs that $\mathit{toTree} \cdot \mathit{toForest} = \mathit{id}$ and $\mathit{toForest} \cdot \mathit{toTree} = \mathit{id}$. We *could* dispatch these proofs with inductive reasoning, or reasoning with catamorphisms. In this paper, we offer an alternative and show how to tackle this isomorphism using reasoning on the type level: $\mathsf{Tree} \cong \mathsf{List} \circ \mathsf{Rose}$.

A second, and perhaps not entirely obvious correspondence is one between rose trees and leaf trees: $\mathsf{Rose} \cong \mathsf{Fork}$.

$$\mathbf{data}\,\mathsf{Fork}\;a = \mathit{Leaf}\;a \mid \mathit{Fork}\,(\mathsf{Fork}\;a, \mathsf{Fork}\;a)$$

We shall abstain from talking about arrows as much as possible. This sounds as though it would run counter to the spirit of category theory. Instead our haunt will be one rung up on the abstraction ladder where we will be talking in hom-sets (sets of arrows). This means we will be establishing isomorphisms as far as existence, but stopping short of deriving witnesses.

Now is a good opportunity to cast our minds back to simpler times and recall our high-school algebra. For nostalgia's sake, these basic laws are listed in Table 1. We shall derive these laws from first principles, showcasing two important tools: the Yoneda lemma and adjunctions. We hope to demonstrate to the reader why these concepts should be in their reasoning toolbox.

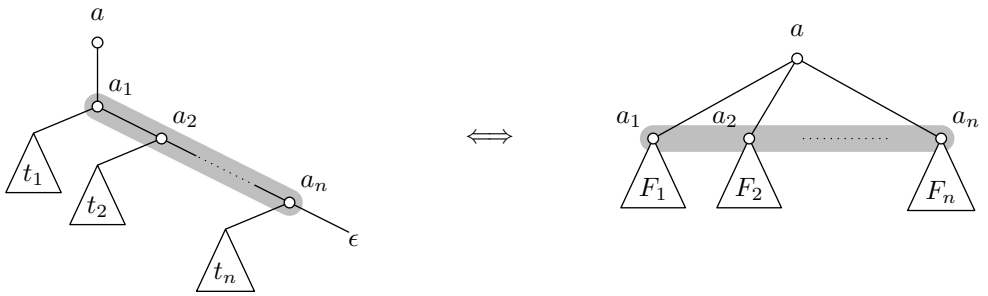


Figure 1. Natural correspondence between binary trees and forests of rose trees, and between pennants and rose trees.

The rest of the paper is structured as follows. The central concept of an isomorphism is introduced in Section 2, along with some basic reasoning principles. The Yoneda lemma, Section 3, answers the question set out in the abstract "When are two types the same?" in an appealing way: two types are isomorphic if they have the same relationships to other types. This observation can be turned into a definitional principle as illustrated in Section 4, where sum and pair types are defined. A definition introduces a new concept in terms of a known concept. A *good* definition is one where the given concept is simple, but the emerging concept is intriguing or interesting. Adjunctions, Section 5, allow us to do exactly this: to define interesting new concepts in terms of simple old ones. Section 6 solves the problem set out in this introduction, providing a set of rules to reason about recursive types. Finally, Section 7 gives suggestions for further reading and Section 8 concludes.

We assume a basic knowledge of category theory, along the lines of the categorical trinity: categories, functors and natural transformations. Readers not familiar with these notions are advised to consult any standard textbook, such as [16].

2. Isomorphisms

Before we turn to the technical material, it is worthwhile ruminating on the question "Why is isomorphism, not equality, the right concept?" Equality is a very strong notion — too strong for our interests. It applies when two things are exactly the same, whereas isomorphism simply requires that there is a correspondence from one to the other and back. We care about the *what*, not the *how*; the *what* being the relationships that an object has with other objects. In types and programming language terms, we are concerned with the specification not the implementation, much in the spirit of abstract data types where we cannot inspect, just simply observe.

Formally, an arrow $f : \mathbb{C}(A, B)$ is an isomorphism if there is an arrow $g : \mathbb{C}(B, A)$, the inverse of f, with $g \cdot f = id_A$ and $f \cdot g = id_B$. If g exists, it is unique and is written as f°.[1] For the proof, assume that we have two inverses, g_1 and g_2. Then

$$\begin{array}{ll} & g_1 \\ = & \quad \{\ f \cdot g_2 = id_B\ \} \\ & g_1 \cdot f \cdot g_2 \\ = & \quad \{\ g_1 \cdot f = id_A\ \} \\ & g_2\ . \end{array}$$

Having just given a simple equational proof, now is a good opportunity to explain our proof format. Proof steps are related by an operator in the left column, with justifications written between curly braces. The transitivity of the operators relates the first and last lines.

[1] We have chosen f° as the name of the inverse rather than f^{-1}.

Two objects A and B are isomorphic, written $A \cong B$, if there is an isomorphism $f : \mathbb{C}(A, B)$. We write $f : A \cong B$ if the category $\mathbb{C}$ is evident from the context. The notation $f : A \cong B : f^\circ$ is used if we additionally want to record the inverse. If we ignore the witnesses of isomorphisms, the relation $\cong$ is an equivalence relation: it is reflexive ($id_A : A \cong A : id_A$), symmetric ($f : A \cong B : f^\circ$ implies $f^\circ : B \cong A : f$) and transitive ($f : A \cong B : f^\circ$ and $g : B \cong \mathbb{C} : g^\circ$ imply $g \cdot f : A \cong \mathbb{C} : f^\circ \cdot g^\circ$).

A common approach for establishing an isomorphism is to provide two proofs: one for $f^\circ \cdot f = id_A$ and one for $f \cdot f^\circ = id_B$. The following characterisation often allows us to replace these two proofs by a single one. The arrow $f : \mathbb{C}(A, B)$ is an isomorphism with inverse $f^\circ : \mathbb{C}(B, A)$ if and only if

$$a = f^\circ \cdot b \iff f \cdot a = b\ , \tag{1}$$

for all $a : \mathbb{C}(X, A)$ and $b : \mathbb{C}(X, B)$, for some $X : \mathbb{C}$. An arrow is a total function in the category $\mathbf{Set}$, and it is an isomorphism if and only if it is bijective. For this particular category, we can simplify (1) by replacing function composition by function application:

$$a = f^\circ\ b \iff f\ a = b\ , \tag{2}$$

for all $a : A$ and $b : B$.

To showcase the characterisation (1), let us prove that f is an isomorphism if and only if post-composition $f\cdot$ is an isomorphism. The property also holds if we replace post- by pre-composition:

$$f : A \cong B : f^\circ \iff (f\cdot) : \mathbb{C}(X, A) \cong \mathbb{C}(X, B) : (f^\circ\cdot)\ , \tag{3}$$

$$f : A \cong B : f^\circ \iff (\cdot f) : \mathbb{C}(B, X) \cong \mathbb{C}(A, X) : (\cdot f^\circ)\ , \tag{4}$$

where X is some fixed object. Post- and pre-composition take an arrow to an arrow, so they are functions that live in $\mathbf{Set}$. The proof of (3) shows that the property is almost content-free.

$$\begin{array}{ll} & a = f^\circ \cdot b \iff f \cdot a = b \\ \iff & \quad \{\ \text{definition of post-composition: } (g\cdot)\, f = g \cdot f\ \} \\ & a = (f^\circ\cdot)\ b \iff (f\cdot)\ a = b \end{array}$$

2.1 Two Special Cases

We have defined isomorphism of objects, but what about isomorphism of functors? Perhaps surprisingly, no new definition is needed as functors are *objects* in a functor category. If the ambient category is $\mathbb{D}^\mathbb{C}$, then an isomorphism is one between functors $\mathsf{F} \cong \mathsf{G}$, and the witness of the isomorphism is a natural transformation. (Some authors say that F and G are *naturally* isomorphic.)

A second choice of ambient category is $\mathbf{Cat}$. In this instance, the objects are all the small categories, an isomorphism is one between categories $\mathbb{C} \cong \mathbb{D}$, and the witness is a functor. In general, isomorphism of categories is too strong a notion to be useful, because equality of objects is too strong a notion: F and G are inverses if $\mathsf{G} \circ \mathsf{F} = \mathsf{Id}_\mathbb{C}$ and $\mathsf{F} \circ \mathsf{G} = \mathsf{Id}_\mathbb{D}$. If we replace equality by

(natural) isomorphism, $\mathsf{G} \circ \mathsf{F} \cong \mathsf{Id}_{\mathbb{C}}$ and $\mathsf{F} \circ \mathsf{G} \cong \mathsf{Id}_{\mathbb{D}}$, we obtain the weaker, but more useful notion of *equivalence* of categories.

2.2 Preservation of Isomorphisms

The equivalence relation $\cong$ is in fact a congruence relation as functors preserve $\cong$:

$$A \cong B \implies \mathsf{F}\,A \cong \mathsf{F}\,B \ . \tag{5}$$

Let $f : A \cong B$ and $\mathsf{F} : \mathbb{C} \to \mathbb{D}$. We show $\mathsf{F}\,f : \mathsf{F}\,A \cong \mathsf{F}\,B$ with $(\mathsf{F}\,f)^\circ = \mathsf{F}\,(f^\circ)$:

$$\begin{array}{ll} & f^\circ \cdot f = id_A \ \wedge \ f \cdot f^\circ = id_B \\ \implies & \quad \{ \text{ Leibniz } \} \\ & \mathsf{F}\,(f^\circ \cdot f) = \mathsf{F}\,id_A \ \wedge \ \mathsf{F}\,(f \cdot f^\circ) = \mathsf{F}\,id_B \\ \iff & \quad \{ \ \mathsf{F} \text{ functor } \} \\ & \mathsf{F}\,(f^\circ) \cdot \mathsf{F}\,f = id_{\mathsf{F}\,A} \ \wedge \ \mathsf{F}\,f \cdot \mathsf{F}\,(f^\circ) = id_{\mathsf{F}\,B} \ . \end{array}$$

Property (5) has far-reaching consequences. For example, suppose that we have two isomorphisms, one on functors, one on objects. Then, the following is an instance of (5):

$$\mathsf{F} \cong \mathsf{G} \ \wedge \ A \cong B \implies \mathsf{F}\,A \cong \mathsf{G}\,B \ . \tag{6}$$

This follows from the fact that functor application is itself a functor. Functor composition is also functorial, so it too preserves $\cong$:

$$\mathsf{F}_1 \cong \mathsf{F}_2 \ \wedge \ \mathsf{G}_1 \cong \mathsf{G}_2 \implies \mathsf{F}_1 \circ \mathsf{G}_1 \cong \mathsf{F}_2 \circ \mathsf{G}_2 \ . \tag{7}$$

2.3 Reflection of Isomorphisms

Preservation of isomorphism follows directly from the basic properties of functors. It turns out that functors that satisfy two additional properties also reflect isomorphisms, the converse of (5). These are the properties of *fullness* and *faithfulness*. For a pair of objects $A, B : \mathbb{C}$, the arrow part of functor $\mathsf{F} : \mathbb{C} \to \mathbb{D}$ maps each arrow $f : \mathbb{C}(A, B)$ to an arrow $\mathsf{F}\,f : \mathbb{D}(\mathsf{F}\,A, \mathsf{F}\,B)$ and so defines a function on hom-sets,

$$\mathsf{F} : \mathbb{C}(A, B) \to \mathbb{D}(\mathsf{F}\,A, \mathsf{F}\,B) \ .$$

The functor F is *full* when every such function is surjective: for every arrow $g : \mathbb{D}(\mathsf{F}\,A, FB)$, there is an arrow $f : \mathbb{C}(A, B)$ with $g = \mathsf{F}\,f$. Similarly the functor F is *faithful* if this function between hom-sets is injective: for every pair of arrows $f_1, f_2 : \mathbb{C}(A, B)$, the equality $\mathsf{F}\,f_1 = \mathsf{F}\,f_2$ implies $f_1 = f_2$. For a functor that is *fully faithful*, F is a bijection on the hom-sets.

$$\mathsf{F} : \mathbb{C}(A, B) \cong \mathbb{D}(\mathsf{F}\,A, \mathsf{F}\,B) \tag{8}$$

The function F is invertible, and we shall call the inverse F°. Since F and F° live in $\mathbf{Set}$, we can write (8) as an equivalence:

$$f = \mathsf{F}^\circ\,g \iff \mathsf{F}\,f = g \ . \tag{9}$$

Fully faithful functors reflect $\cong$:

$$A \cong B \impliedby \mathsf{F}\,A \cong \mathsf{F}\,B \ . \tag{10}$$

The proof is similar to the one for (5):

$$\begin{array}{ll} & g^\circ \cdot g = id_{\mathsf{F}\,A} \ \wedge \ g \cdot g^\circ = id_{\mathsf{F}\,B} \\ \implies & \quad \{ \ \mathsf{F} \text{ full, let } g = \mathsf{F}\,f \text{ and } g^\circ = \mathsf{F}\,(f^\circ) \ \} \\ & \mathsf{F}\,(f^\circ) \cdot \mathsf{F}\,f = id_{\mathsf{F}\,A} \ \wedge \ \mathsf{F}\,f \cdot \mathsf{F}\,(f^\circ) = id_{\mathsf{F}\,B} \\ \iff & \quad \{ \ \mathsf{F} \text{ functor } \} \\ & \mathsf{F}\,(f^\circ \cdot f) = \mathsf{F}\,id_A \ \wedge \ \mathsf{F}\,(f \cdot f^\circ) = \mathsf{F}\,id_B \\ \implies & \quad \{ \ \mathsf{F} \text{ faithful } \} \\ & f^\circ \cdot f = id_A \ \wedge \ f \cdot f^\circ = id_B \ . \end{array}$$

Since functors preserve $\cong$, property (10) can be strengthened to an equivalence for fully faithful functors:

$$A \cong B \iff \mathsf{F}\,A \cong \mathsf{F}\,B \ . \tag{11}$$

Furthermore, we can lift the property to an equivalence between functors:

$$\mathsf{G} \cong \mathsf{H} \iff \mathsf{F} \circ \mathsf{G} \cong \mathsf{F} \circ \mathsf{H} \ . \tag{12}$$

This is a consequence of the fact that if $\mathsf{F} : \mathbb{C} \to \mathbb{D}$ is fully faithful, then post-composition $\mathsf{F}\circ : \mathbb{C}^{\mathbb{I}} \to \mathbb{D}^{\mathbb{I}}$ is fully faithful, as well. We leave the details to the interested reader.

3. Yoneda Lemma

Suppose we are working with pre-orders and we wish to prove the relation $a \lesssim b$. We could attempt to give a direct proof of this property, however, an alternative strategy is to give an *indirect* proof using the equivalence $a \lesssim b \iff (\forall x \ . \ b \lesssim x \implies a \lesssim x)$. In words, a is at most b if and only if b is at most x implies a is at most x, for any x. A pre-order forms a simple category, so one may wonder whether we can generalise the proof principle to an arbitrary category. It turns out that the equivalence is actually an instance of something in category theory called the Yoneda lemma. Our aim is to make use of the Yoneda lemma in the same way, namely as a technique for indirect proof of isomorphism.

As a means of introduction, we shall approach the Yoneda lemma through its relation to continuation passing style (CPS), something more familiar to functional programmers. The following is the factorial function written in CPS style:

$$\begin{array}{l} fac : \forall\, x \ . \ (Nat \to x) \to (Nat \to x) \\ fac\,\kappa\,0 \qquad\ = \kappa\,1 \\ fac\,\kappa\,(n+1) = fac\,(\lambda r \to \kappa\,(r * (n+1)))\,n \ . \end{array}$$

The first argument to fac is the *continuation*. CPS functions do not 'return', they pass their result to a continuation. Using the identity function as the initial continuation, the call $fac\,id\,5$ yields 120. If we loosely equate parametricity with naturality, the type signature identifies fac as a natural transformation, natural in x: it satisfies the naturality property $h \cdot fac\,\kappa = fac\,(h \cdot \kappa)$, which implies $h\,(fac\,id\,n) = fac\,h\,n$. That $fac\,id$ is indeed the factorial function can now be seen by two simple calculations:

$$\begin{array}{ll} & fac\,id\,0 \\ = & \quad \{ \text{ definition of } fac \ \} \\ & id\,1 \\ = & \quad \{ \text{ identity } \} \\ & 1 \end{array}$$

and

$$\begin{array}{ll} & fac\,id\,(n+1) \\ = & \quad \{ \text{ definition of } fac \ \} \\ & fac\,(\lambda r \to id\,(r * (n+1)))\,n \\ = & \quad \{ \text{ identity } \} \\ & fac\,(\lambda r \to r * (n+1))\,n \\ = & \quad \{ \text{ naturality of } fac \text{ and } \beta\text{-reduction } \} \\ & (fac\,id\,n) * (n+1) \ . \end{array}$$

There is an isomorphism between the types of fac, the factorial function in CPS style, and the direct factorial function of type $Nat \to Nat$. Applying the identity function is one direction of the isomorphism, from CPS to direct style. The factorial isomorphism is an instance of the isomorphism of the types $A \to B$ and $\forall X \ . \ (B \to X) \to (A \to X)$. Compare this to the pre-order equivalence above, replacing $\lesssim$ and $\implies$ with $\to$. From left to right,

the isomorphism is $\mathbb{b}f = \lambda\kappa \;.\; \kappa \cdot f$, and from right to left, $\mathbb{b}^\circ\, \mathbb{a} = \mathbb{a}\, id$.

Let us pause for a moment to introduce the so-called hom-functor, $\mathbb{C}(-,=) : \mathbb{C}^{\mathsf{op}} \times \mathbb{C} \to \mathbf{Set}$, which takes two objects to the *set* of arrows between them. (We use the notation of $-$ and $=$ for implicitly bound variables.) The hom-functor is a bifunctor, contravariant in its first argument and covariant in its second. We will use it with either its first or its second argument fixed, making it covariant and contravariant, respectively. The covariant hom-functor for a fixed object A is $\mathbb{C}(A,-) : \mathbb{C} \to \mathbf{Set}$, with the action on arrows given by $\mathbb{C}(A,f)\, h = f \cdot h$. The functor $\mathbb{C}(A,-)$ maps an object B to the *set* of arrows from a fixed A to B, and it takes an arrow $f : \mathbb{C}(X,\, Y)$ to a *function* $\mathbb{C}(A,f) : \mathbb{C}(A,X) \to \mathbb{C}(A,\, Y)$. Conversely, $\mathbb{C}(-,B) : \mathbb{C}^{\mathsf{op}} \to \mathbf{Set}$ is a covariant functor defined $\mathbb{C}(f,B)\, h = h \cdot f$.

Let us continue by transliterating the CPS isomorphism into more categorical language,

$$\mathbb{C}(A,B) \;\cong\; \forall X \;.\; \mathbb{C}(B,X) \to \mathbb{C}(A,X) \;\;.$$

We can continue generalizing by extracting out the covariant hom-functor $\mathbb{C}(A,-)$ and naming it H.

$$\mathsf{H}\, B \;\cong\; \forall X \;.\; \mathbb{C}(B,X) \to \mathsf{H}\, X \;\;. \tag{13}$$

The covariant hom-functor has type $\mathbb{C} \to \mathbf{Set}$, and in one final abstraction we can let $\mathsf{H} : \mathbb{C} \to \mathbf{Set}$ be any set-valued functor. The isomorphism we have arrived at is a statement of the Yoneda lemma. It remains to show that the following arrows are indeed witnesses of the Yoneda isomorphism:

$$\mathbb{b}\, s = \Lambda\, X \;.\; \lambda f : B \to X \;.\; \mathsf{H} f\, s \;\text{ and }\; \mathbb{b}^\circ\, \mathbb{a} = \mathbb{a}\, B\, id_B \;\;.$$

Observe that $\mathbb{b}$ is just H with the two arguments flipped. It is easy to see that $\mathbb{b}^\circ$ is the left-inverse of $\mathbb{b}$.

$$\begin{array}{ll} & \mathbb{b}^\circ\, (\mathbb{b}\, s) \\ = & \quad \{\ \text{definition of } \mathbb{b}^\circ\ \} \\ & \mathbb{b}\, s\, B\, id_B \\ = & \quad \{\ \text{definition of } \mathbb{b}\ \} \\ & \mathsf{H}\, id_B\, s \\ = & \quad \{\ \mathsf{H} \text{ functor}\ \} \\ & s \end{array}$$

For the opposite direction, we make use of the naturality of $\mathbb{a}$, that is, $\mathsf{H}\, h \cdot \mathbb{a}\, X = \mathbb{a}\, Y \cdot \mathbb{C}(B,h)$, or written in a pointwise style: $\mathsf{H}\, h\, (\mathbb{a}\, X\, g) = \mathbb{a}\, Y\, (h \cdot g)$, with $h : X \to Y$ and $g : B \to X$.

$$\begin{array}{ll} & \mathbb{b}\, (\mathbb{b}^\circ\, \mathbb{a})\, X \\ = & \quad \{\ \text{definition of } \mathbb{b}\ \} \\ & \lambda f \;.\; \mathsf{H} f\, (\mathbb{b}^\circ\, \mathbb{a}) \\ = & \quad \{\ \text{definition of } \mathbb{b}^\circ\ \} \\ & \lambda f \;.\; \mathsf{H} f\, (\mathbb{a}\, B\, id_B) \\ = & \quad \{\ \text{naturality of } \mathbb{a}\ \} \\ & \lambda f \;.\; \mathbb{a}\, X\, (f \cdot id_B) \\ = & \quad \{\ \text{identity}\ \} \\ & \lambda f \;.\; \mathbb{a}\, X\, f \\ = & \quad \{\ \text{extensionality}\ \} \\ & \mathbb{a}\, X \end{array}$$

The Yoneda lemma implies the proof principle of indirect isomorphism. To this end we define a functor $\mathsf{Y}\, A = \Lambda\, B \;.\; \mathbb{C}(A,B)$, which is a mapping of objects to functors and arrows to natural transformations. This is known as the Yoneda functor $\mathsf{Y} : \mathbb{C}^{\mathsf{op}} \to \mathbf{Set}^{\mathbb{C}}$ (or the Yoneda embedding by some authors). Note that Y is nothing more than the curried hom-functor $\mathbb{C}(-,=) : \mathbb{C}^{\mathsf{op}} \times \mathbb{C} \to \mathbf{Set}$. The Yoneda lemma implies that Y is both full and faithful. The proof proceeds by instantiating the set-valued functor H to $\mathsf{Y}\, B$ and rewriting to show the bijection of hom-sets as in (8):

$$\begin{array}{ll} & \mathsf{H}\, A \cong \forall X \;.\; \mathbb{C}(A,X) \to \mathsf{H}\, X \\ \Longrightarrow & \quad \{\ \text{set } \mathsf{H} := \mathsf{Y}\, B\ \} \\ & \mathsf{Y}\, B\, A \cong \forall X \;.\; \mathbb{C}(A,X) \to \mathsf{Y}\, B\, X \\ \Longleftrightarrow & \quad \{\ \text{definition of } \mathsf{Y}\ \} \\ & \mathbb{C}(B,A) \cong \forall X \;.\; \mathsf{Y}\, A\, X \to \mathsf{Y}\, B\, X \\ \Longleftrightarrow & \quad \{\ \text{definition of } \mathbb{C}^{\mathsf{op}} \text{ and functor categories}\ \} \\ & \mathbb{C}^{\mathsf{op}}(A,B) \cong \mathbf{Set}^{\mathbb{C}}(\mathsf{Y}\, A, \mathsf{Y}\, B) \;\;. \end{array}$$

As one might expect, the Yoneda functor has a dual, $\bar{\mathsf{Y}} : \mathbb{C} \to \mathbf{Set}^{\mathbb{C}^{\mathsf{op}}}$ with $\bar{\mathsf{Y}}\, B = \Lambda\, A \;.\; \mathbb{C}(A,B)$. The dual functor $\bar{\mathsf{Y}}$ is full and faithful too. (This is an instance of the development above as $\bar{\mathsf{Y}}\, B = \mathbb{C}^{\mathsf{op}}(B,-)$.) Consequently,

$$B \cong A \iff \mathbb{C}(A,-) \cong \mathbb{C}(B,-) \;\;, \tag{14}$$

$$A \cong B \iff \mathbb{C}(-,A) \cong \mathbb{C}(-,B) \;\;. \tag{15}$$

Thus, we can prove that two objects, A and B, are isomorphic by showing an isomorphism between the set of arrows from A and the set of arrows from B (or to A and B). Earlier we said that we were interested in the *what*, not the *how*; we have just shown that an object is fully determined, up to isomorphism, by what relationships it has with other objects in $\mathbb{C}$. Naturality is the key. Notice how (14) and (15) have generalised (3) and (4). On the right hand side of the equivalence we have a *natural* isomorphism between hom-functors, and this equation between functors is not for some fixed object, but for any object of $\mathbb{C}$.

Later on we will use the indirect isomorphism of (14) and (15) to prove the type isomorphisms of high-school algebra.

Finally, let us climb up one more step on the abstraction ladder. Since Y is fully faithful and consequently $\mathsf{Y}\circ$ we have

$$\mathsf{F} \cong \mathsf{G} \iff \mathsf{Y} \circ \mathsf{F} \cong \mathsf{Y} \circ \mathsf{G} \;\;, \tag{16}$$

where $\mathsf{F}, \mathsf{G} : \mathbb{I} \to \mathbb{C}$. This translates to

$$\mathsf{F} \cong \mathsf{G} \iff \mathbb{C}(\mathsf{F}-,=) \cong \mathbb{C}(\mathsf{G}-,=) \;\;, \tag{17}$$

$$\mathsf{F} \cong \mathsf{G} \iff \mathbb{C}(-,\mathsf{F}=) \cong \mathbb{C}(-,\mathsf{G}=) \;\;. \tag{18}$$

These two rules offer yet another approach to showing the isomorphism of two functors.

4. Representing Objects/Representable Functors

The gist of the Yoneda lemma is that an object is fully determined by its relation to other objects. This observation can be turned into a definitional principle. We illustrate the principle using coproducts and then apply it to the other arithmetic objects of high-school algebra.

A set-valued functor $\mathsf{H} : \mathbb{C} \to \mathbf{Set}$ is *representable* if there exists a *representing object* $X : \mathbb{C}$ such that

$$\mathbb{C}(X,-) \;\cong\; \mathsf{H} \;\;. \tag{19}$$

The functor equation defines X up to isomorphism. Let X_1 and X_2 be objects that solve (19), then

$$\begin{array}{ll} & \mathbb{C}(X_1,-) \\ \cong & \quad \{\ X_1 \text{ solves (19)}\ \} \\ & \mathsf{H} \\ \cong & \quad \{\ X_2 \text{ solves (19)}\ \} \\ & \mathbb{C}(X_2,-) \end{array}$$

Indirect isomorphism (14) implies $X_1 \cong X_2$.

As an example, for the functor $\mathsf{H} = \mathbb{C}(A_1, -) \times \mathbb{C}(A_2, -)$ the representing object is the coproduct of A_1 and A_2. Putting the focus on the object instead of the functor, we can use (19) as the definition of a coproduct: An object X is the *coproduct of* A_1 *and* A_2 if it solves the equation

$$\mathbb{C}(X, -) \;\cong\; \mathbb{C}(A_1, -) \times \mathbb{C}(A_2, -) \;. \tag{20}$$

The equation between functors demands that a pair of functions to a common target can be represented by a single function from X to the target, and vice versa. We write X as $A_1 + A_2$. This is a *representational* approach to defining coproducts, where we define an object in terms of a representation rather than its construction.

The arithmetic objects, 0, 1, $+$, $\times$ are defined as follows:

$$\Lambda\, B \,.\, \mathbb{C}(0, B) \cong \Lambda\, B \,.\, 1 \;, \tag{21}$$
$$\Lambda\, B \,.\, \mathbb{C}(A_1 + A_2, B) \cong \Lambda\, B \,.\, \mathbb{C}(A_1, B) \times \mathbb{C}(A_2, B) \;, \tag{22}$$
$$\Lambda\, A \,.\, \mathbb{C}(A, 1) \cong \Lambda\, A \,.\, 1 \;, \tag{23}$$
$$\Lambda\, A \,.\, \mathbb{C}(A, B_1 \times B_2) \cong \Lambda\, A \,.\, \mathbb{C}(A, B_1) \times \mathbb{C}(A, B_2) \;. \tag{24}$$

(This time we have made explicit that the equations relate functors, not objects.) An object is initial, written 0, if there is exactly one arrow from 0 to any object. The last two equations are dual to the first two: final objects are dual to initial objects and products are dual to coproducts. It is worth pointing out that the last two equations are not circular — it may appear that we are defining 1 in terms of 1 or $\times$ in terms of $\times$. The constructions on the right-hand side live in $\mathbf{Set}$. The final object in $\mathbf{Set}$ is a one-element set, which is why (21) and (23) express that the cardinality of $\mathbb{C}(0, B)$ and $\mathbb{C}(A, 1)$ is one.

As an immediate benefit of defining the arithmetic objects as solutions of functor equations, we can easily prove their properties. For example, the initial object and coproducts form a commutative monoid: the coproduct $+$ is associative and commutative with the initial object 0 as its neutral element.

$$X + 0 \cong X \tag{25}$$
$$X + Y \cong Y + X \tag{26}$$
$$X + (Y + Z) \cong (X + Y) + Z \tag{27}$$

For the proofs we reduce the properties to corresponding properties of 1 and $\times$ in $\mathbf{Set}$.

$$\begin{aligned}
&\mathbb{C}(X + 0, -) \\
\cong\;& \{ \text{ definition of } + \text{ and } 0 \text{ (22, 21) } \} \\
&\mathbb{C}(X, -) \times 1 \\
\cong\;& \{ \; A \times 1 \cong A \text{ in } \mathbf{Set} \; \} \\
&\mathbb{C}(X, -)
\end{aligned}$$

Here we used neutrality of 1. As to be expected, the next proof relies on the commutativity of $\times$ in $\mathbf{Set}$.

$$\begin{aligned}
&\mathbb{C}(X + Y, -) \\
\cong\;& \{ \text{ definition of } + \text{ (22) } \} \\
&\mathbb{C}(X, -) \times \mathbb{C}(Y, -) \\
\cong\;& \{ \; A \times B \cong B \times A \text{ in } \mathbf{Set} \; \} \\
&\mathbb{C}(Y, -) \times \mathbb{C}(X, -) \\
\cong\;& \{ \text{ definition of } + \text{ (22) } \} \\
&\mathbb{C}(Y + X, -)
\end{aligned}$$

The final proof leverages the associativity of $\times$ in $\mathbf{Set}$.

$$\begin{aligned}
&\mathbb{C}((X + Y) + Z, -) \\
\cong\;& \{ \text{ definition of } + \text{ (22) } \} \\
&(\mathbb{C}(X, -) \times \mathbb{C}(Y, -)) \times \mathbb{C}(Z, -) \\
\cong\;& \{ \; (A \times B) \times C \cong A \times (B \times C) \text{ in } \mathbf{Set} \; \} \\
&\mathbb{C}(X, -) \times (\mathbb{C}(Y, -) \times \mathbb{C}(Z, -)) \\
\cong\;& \{ \text{ definition of } + \text{ (22) } \} \\
&\mathbb{C}(X + (Y + Z), -)
\end{aligned}$$

We can dualise and enjoy the same properties. The terminal object and products form a commutative monoid: $\times$ is associative and commutative with 1 as its neutral element.

$$X \times 1 \cong X \tag{28}$$
$$X \times Y \cong Y \times X \tag{29}$$
$$X \times (Y \times Z) \cong (X \times Y) \times Z \tag{30}$$

The isomorphism (22) defines a coproduct in $\mathbb{C}$ in terms of a product in $\mathbf{Set}$ — this is where the product on the right-hand side of (22) lives. Alternatively, we can define the coproduct using the product in $\mathbf{Cat}$, that is, a product category. An object X is the *coproduct of* A_1 *and* A_2 if it solves the equation

$$\mathbb{C}(X, -) \;\cong\; (\mathbb{C} \times \mathbb{C})(\langle A_1, A_2 \rangle, \Delta -) \;. \tag{31}$$

The so-called diagonal functor $\Delta : \mathbb{C} \to \mathbb{C} \times \mathbb{C}$ duplicates its argument $\Delta\, A = \langle A, A \rangle$ and $\Delta\, f = \langle f, f \rangle$. Because of the equality $(\mathbb{C}_1 \times \mathbb{C}_2)(\langle A_1, A_2 \rangle, \langle B_1, B_2 \rangle) = \mathbb{C}_1(A_1, B_1) \times \mathbb{C}_2(A_2, B_2)$, the two approaches are in fact equivalent.

5. Adjunctions

The notion of an *adjunction* was introduced by Daniel Kan in 1958. Adjunctions have proved to be one of the most important ideas in category theory, predominantly due to their ubiquity. Many mathematical constructions turn out to be adjoint functors that form adjunctions, with Mac Lane famously saying, "Adjoint functors arise everywhere."

There are several routes to introducing and defining adjunctions. Cognisant of the fact that adjunctions are such a significant topic, we will cover them from a limited perspective and see what bearing they have on the constructions we have covered so far, as well as a new construction, exponentials. To us, adjunctions will be a means to concisely express a lot of structure about these constructions, and also a useful tool for proving isomorphisms.

We have bumped into an adjunction already. Recall the definition of coproducts using products in $\mathbf{Cat}$ (31):

$$\mathbb{C}(+\langle A_1, A_2 \rangle, -) \;\cong\; (\mathbb{C} \times \mathbb{C})(\langle A_1, A_2 \rangle, \Delta -) \;.$$

We have substituted in the coproduct for X and written $+$ in prefix notation. Going one step further, we can bring out the symmetry.

$$\mathbb{C}(+(-), =) \;\cong\; (\mathbb{C} \times \mathbb{C})(-, \Delta\,(=)) \;.$$

We have turned $+$ into a functor, a bifunctor to be precise, and we are implying a symmetry between it and the diagonal functor.

$$\mathbb{C} \;\underset{\Delta}{\overset{+}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\; \mathbb{C} \times \mathbb{C}$$

This data defines an adjunction; let us see the general case.

Let $\mathbb{C}$ and $\mathbb{D}$ be categories. The functors L and R are *adjoint*, written $\mathsf{L} \dashv \mathsf{R} : \mathbb{C} \to \mathbb{D}$,

$$\mathbb{C} \;\underset{\mathsf{R}}{\overset{\mathsf{L}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\; \mathbb{D}$$

if there is a natural isomorphism,

$$\phi : \mathbb{C}(\mathsf{L}\,-,=) \cong \mathbb{D}(-,\mathsf{R}\,=) \ . \tag{32}$$

An adjunction, then, is the triple of the left adjoint L, the right adjoint R and the adjoint transposition ϕ. The type in $\mathsf{L} \dashv \mathsf{R} : \mathbb{C} \to \mathbb{D}$ describes the type of R, with L going in the opposite direction. Since the adjoint transposition lives in $\mathbf{Set}$, we can render it as an equivalence, see (2):

$$f = \phi^\circ\, g \iff \phi f = g \ , \tag{33}$$

for all $f : \mathbb{C}(\mathsf{L}\,A, B)$ and $g : \mathbb{D}(A, \mathsf{R}\,B)$. (The left-hand side lives in $\mathbb{C}$, and the right-hand side in $\mathbb{D}$.) We shall see that the computational content of an adjunction surfaces in this equivalence.

Adjoints are unique up to isomorphism: if $\mathsf{L}_1 \dashv \mathsf{R}$ and $\mathsf{L}_2 \dashv \mathsf{R}$ then $\mathsf{L}_1 \cong \mathsf{L}_2$. (Likewise, for right adjoints.) This is a direct consequence of (17) using:

$$\begin{array}{ll} & \mathbb{C}(\mathsf{L}_1\,-,=) \\ \cong & \{\ \mathsf{L}_1 \dashv \mathsf{R}\ \} \\ & \mathbb{D}(-,\mathsf{R}\,=) \\ \cong & \{\ \mathsf{L}_2 \dashv \mathsf{R}\ \} \\ & \mathbb{C}(\mathsf{L}_2\,-,=) \ . \end{array}$$

5.1 Coproducts and Products, Initial and Final Objects

Coproducts and products form a double adjunction with $+$ left adjoint to Δ and $\times$ right adjoint to Δ.

$$\mathbb{C} \;\underset{\Delta}{\overset{+}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\; \mathbb{C} \times \mathbb{C} \;\underset{\times}{\overset{\Delta}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\; \mathbb{C}$$

The adjunction between $+$ and Δ expresses the isomorphism:

$$\mathbb{C}(A_1 + A_2, B) \cong (\mathbb{C} \times \mathbb{C})(\langle A_1, A_2\rangle, \Delta\, B) : \triangledown \ .$$

Note that this is an instantiation of the representable functor for coproducts (31). Reading the isomorphism from right to left, it takes two functions $f_1 : \mathbb{C}(A_1, B)$ and $f_2 : \mathbb{C}(A_2, B)$, and combines them into a function we call $f_1 \triangledown f_2 : \mathbb{C}(A_1 + A_2, B)$ (pronounced "join"). This should appear familiar as this is case analysis on binary sum types. The isomorphism is usually stated as an equivalence, see (33), the so-called universal property of coproducts.

$$f = g_1 \triangledown g_2 \iff \Delta f \cdot \langle \iota_1, \iota_2\rangle = \langle g_1, g_2\rangle \tag{34}$$

The arrows $\iota_1 : \mathbb{C}(A_1, A_1 + A_2)$ and $\iota_2 : \mathbb{C}(A_2, A_1 + A_2)$ are the injection functions into the coproduct.

Similarly for the adjunction between $\times$ and Δ, the isomorphism is:

$$\triangle : (\mathbb{C} \times \mathbb{C})(\Delta\, A, \langle B_1, B_2\rangle) \cong \mathbb{C}(A, B_1 \times B_2) \ .$$

Earlier, we did not give the representable functor for products in $\mathbf{Cat}$, but this is its instantiation. Reading the isomorphism from left to right, it takes two functions $f_1 : \mathbb{C}(A, B_1)$ and $f_2 : \mathbb{C}(A, B_2)$, and combines them into a function we call $f_1 \triangle f_2 : \mathbb{C}(A, B_1 \times B_2)$ (pronounced "split"). The isomorphism for coproducts echoes a familiar programming language construct, but here the $\triangle$ operator is simply the combinator that performs the parallel application of two functions to a common argument. Again, the isomorphism is often stated as an equivalence, the universal property of products.

$$\langle f_1, f_2\rangle = \langle \pi_1, \pi_2\rangle \cdot \Delta\, g \iff f_1 \triangle f_2 = g \tag{35}$$

The arrows $\pi_1 : \mathbb{C}(A_1 \times A_2, A_1)$ and $\pi_2 : \mathbb{C}(A_1 \times A_2, A_2)$ are the projection functions out of the product.

So, these adjunctions not only give meaning to coproducts and products as functors, but they also exhibit computational content.

Initial and final objects define a rather trivial adjunction between a category $\mathbb{C}$ and a discrete category $\mathbf{1}$.

$$\mathbb{C} \;\underset{\Delta}{\overset{0}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\; \mathbf{1} \;\underset{1}{\overset{\Delta}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\; \mathbb{C}$$

For readers interested in the technical details: The category $\mathbf{1}$ consists of a single object $①$ and a single arrow $id_{①}$. In this degenerate case, the diagonal functor $\Delta : \mathbb{C} \to \mathbf{1}$ maps all objects of $\mathbb{C}$ to $①$ and all arrows to $id_{①}$. The initial and final objects of $\mathbb{C}$ are seen as constant functors from $\mathbf{1}$. (An object $A \in \mathbb{C}$ seen as a functor $A : \mathbf{1} \to \mathbb{C}$ maps $①$ to A and $id_{①}$ to id_A.)

The adjunction on the left expresses the natural isomorphism $\mathbb{C}(0\,①, -) \cong \mathbf{1}(①, \Delta\,-)$. This reduces to $\mathbb{C}(0, -) \cong \mathbf{1}(①, ①)$, which is exactly (21). Similarly the isomorphism of the second adjunction, $\mathbf{1}(①, ①) \cong \mathbb{C}(-, 1)$, expresses (23).

Adjunctions have interesting interactions with the categorical constructions we have considered so far. A functor that is left adjoint preserves initial objects: $\mathsf{L}\,0 \cong 0$. This is a direct consequence of (21) using:

$$\begin{array}{ll} & \mathbb{C}(\mathsf{L}\,0, -) \\ \cong & \{\ \mathsf{L} \dashv \mathsf{R}\ \} \\ & \mathbb{D}(0, \mathsf{R}\,-) \\ \cong & \{\ (21)\ \} \\ & 1 \ . \end{array}$$

Such a functor also preserves coproducts: $\mathsf{L}\,(A_1 + A_2) \cong \mathsf{L}\,A_1 + \mathsf{L}\,A_2$. Again, this is a direct consequence of (22) using:

$$\begin{array}{ll} & \mathbb{C}(\mathsf{L}\,(A_1 + A_2), -) \\ \cong & \{\ \mathsf{L} \dashv \mathsf{R}\ \} \\ & \mathbb{D}(A_1 + A_2, \mathsf{R}\,-) \\ \cong & \{\ (22)\ \} \\ & \mathbb{D}(A_1, \mathsf{R}\,-) \times \mathbb{D}(A_2, \mathsf{R}\,-) \\ \cong & \{\ \mathsf{L} \dashv \mathsf{R}\ \} \\ & \mathbb{D}(\mathsf{L}\,A_1, -) \times \mathbb{D}(\mathsf{L}\,A_2, -) \ . \end{array}$$

Dually, right adjoints preserve final objects ($\mathsf{R}\,1 \cong 1$) and products ($\mathsf{R}\,(A_1 \times A_2) \cong \mathsf{R}\,A_1 \times \mathsf{R}\,A_2$).

5.2 Exponentials

Exponential objects give a categorical interpretation to the programming language concepts of higher-order functions and *currying*. In $\mathbf{Set}$, a two-argument function is *curried* by turning it into a single-argument function that yields another single-argument function, which receives the second argument. The exponential object for objects A and B is written B^A. The following transformation expresses the currying of a two-argument function:

$$\lambda : \mathbb{C}(A \times X, B) \ \cong \ \mathbb{C}(A, B^X) \ . \tag{36}$$

From left to right, an arrow $f : \mathbb{C}(A \times X, B)$ is curried into an arrow $\lambda f : \mathbb{C}(A, B^X)$. We call the reverse direction *uncurrying*. Note the type of uncurrying the identity, $\lambda^\circ\, id_{B^X} : \mathbb{C}(B^X \times X, B)$; this is function application. This observation underlies the universal property of exponentials:

$$f = app \cdot (g \times X) \iff \lambda f = g \ . \tag{37}$$

Again, the computational content of the isomorphism surfaces by writing it as an equivalence.

Exponentials also form an adjunction, which we can tease out by noting that $A \times X$ is functorial in A and B^X is functorial in B.

$$\lambda : \mathbb{C}((- \times X)\, A, B) \;\cong\; \mathbb{C}(A, (-)^X\, B)$$

So $- \times X$ is left adjoint to $(-)^X$.

$$\mathbb{C} \;\underset{(-)^X}{\overset{- \times X}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\; \mathbb{C}$$

In the category $\mathbf{Set}$, an exponential object B^A is the set of functions with domain A and codomain B, which is exactly what the hom-set is: $B^A = \mathbf{Set}(A, B)$. For an arbitrary category $\mathbb{C}$ with exponentials, we might say that $\mathbb{C}(-, =)$ is the *external* hom-functor, and that $(=)^{(-)}$ is the *internal* hom-functor, both having similar properties.

We have already established the monoidal properties of $\times$ and 1, so a simple consequence of the adjunction above is $\mathbb{C}(A, B) \cong \mathbb{C}(1, B^A)$, which relates arrows to exponentials.

$$\begin{array}{ll} & \mathbb{C}(A, B) \\ \cong & \quad \{ \text{ (28) and (29) } \} \\ & \mathbb{C}(1 \times A, B) \\ \cong & \quad \{ - \times X \dashv (-)^X \} \\ & \mathbb{C}(1, B^A) \end{array}$$

Being a left adjoint, $- \times X$ preserves 0 and $+$, and dually $(-)^X$, as a right adjoint, preserves 1 and $\times$.

$$0 \times X \;\cong\; 0 \tag{38}$$

$$(X + Y) \times Z \;\cong\; (X \times Z) + (Y \times Z) \tag{39}$$

$$1^X \;\cong\; 1 \tag{40}$$

$$(Y \times Z)^X \;\cong\; Y^X \times Z^X \tag{41}$$

A category is called bicartesian closed if the adjunctions $0 \dashv \Delta \dashv 1$, $+ \dashv \Delta \dashv \times$ and $- \times X \dashv (-)^X$ exist, the latter for each X. The reasoning above shows that these categories are automatically distributive, (38) and (39).

We can form another functor $X^{(-)} : \mathbb{C}^{\mathsf{op}} \to \mathbb{C}$, if we fix the target object of the internal hom-functor, rather than the source. This is right adjoint to itself, $(X^{(-)})^{\mathsf{op}} : \mathbb{C} \to \mathbb{C}^{\mathsf{op}}$.

$$\mathbb{C}^{\mathsf{op}} \;\underset{X^{(-)}}{\overset{(X^{(-)})^{\mathsf{op}}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\; \mathbb{C}$$

This adjunction is a consequence of currying.

$$\begin{array}{ll} & \mathbb{C}^{\mathsf{op}}((X^{(-)})^{\mathsf{op}}\, A, B) \\ \cong & \quad \{ \text{ opposite category } \} \\ & \mathbb{C}(B, X^A) \\ \cong & \quad \{ - \times X \dashv (-)^X \} \\ & \mathbb{C}(B \times A, X) \\ \cong & \quad \{ \text{ (29) } \} \\ & \mathbb{C}(A \times B, X) \\ \cong & \quad \{ - \times X \dashv (-)^X \} \\ & \mathbb{C}(A, X^B) \end{array}$$

Since the functor $X^{(-)}$ is contravariant, it takes 0 to 1 and $+$ to $\times$, giving a categorical setting to the laws of exponentials.

$$X^0 \;\cong\; 1 \tag{42}$$

$$Z^{X+Y} \;\cong\; Z^X \times Z^Y \tag{43}$$

These two laws come 'for free'. The next two use the adjunction for $(-)^X : \mathbb{C} \to \mathbb{C}$ and the monoidal properties of $\times$ with 1.

$$X^1 \;\cong\; X \tag{44}$$

$$(Z^Y)^X \;\cong\; Z^{X \times Y} \tag{45}$$

The first proof requires the neutrality of 1.

$$\begin{array}{ll} & \mathbb{C}(-, X^1) \\ \cong & \quad \{ \text{ currying: } - \times A \dashv (-)^A \} \\ & \mathbb{C}(- \times 1, X) \\ \cong & \quad \{ \text{ (28) } \} \\ & \mathbb{C}(-, X) \end{array}$$

The second proof requires the associativity of $\times$.

$$\begin{array}{ll} & \mathbb{C}(-, (Z^Y)^X) \\ \cong & \quad \{ \text{ currying: } - \times A \dashv (-)^A \} \\ & \mathbb{C}(- \times X, Z^Y) \\ \cong & \quad \{ \text{ currying: } - \times A \dashv (-)^A \} \\ & \mathbb{C}((- \times X) \times Y, Z) \\ \cong & \quad \{ \times \text{ is associative } \} \\ & \mathbb{C}(- \times (X \times Y), Z) \\ \cong & \quad \{ \text{ currying: } - \times A \dashv (-)^A \} \\ & \mathbb{C}(-, Z^{X \times Y}) \end{array}$$

This completes the proofs of the basic laws of high-school algebra. It is worth pointing out that all the laws of Table 1 are natural in all the variables. As an example, $X + 0 \cong X$ is natural in X, that is, $\Lambda\, X \;.\; X + 0 \cong \Lambda\, X \;.\; X$. The proof of the former law, given in Section 4, can be generalised to a proof of the latter.

$$\begin{array}{ll} & \Lambda\langle X, Y\rangle \;.\; \mathbb{C}(X + 0, Y) \\ \cong & \quad \{ + \dashv \Delta,\, 0 \dashv \Delta \text{ and product in } \mathbf{Cat} \} \\ & \Lambda\langle X, Y\rangle \;.\; \mathbb{C}(X, Y) \times 1 \\ \cong & \quad \{ \Lambda\, A \;.\; A \times 1 \cong \Lambda\, A \;.\; A \text{ in } \mathbf{Set} \} \\ & \Lambda\langle X, Y\rangle \;.\; \mathbb{C}(X, Y) \end{array}$$

Each step rests on the naturality of the underlying transformation. The natural isomorphism $\Lambda\, X \;.\; X + 0 \cong \Lambda\, X \;.\; X$ then follows from indirect equality of functors (17).

If a property involves several variables, then the natural transformation is one between functors from a product category. (As an aside, a transformation between bifunctors is natural if and only if it is natural in each argument separately.)

5.3 Composition of Adjunctions

The identity functor forms a trivial adjunction with itself $\mathsf{Id} \dashv \mathsf{Id} : \mathbb{C} \to \mathbb{C}$.

$$\mathbb{C} \;\underset{\mathsf{Id}}{\overset{\mathsf{Id}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\; \mathbb{C}$$

The adjoint transposition is simply the identity natural transformation, natural in A and B:

$$id : \mathbb{C}(\mathsf{Id}\, A, B) \cong \mathbb{C}(A, \mathsf{Id}\, B) \;\;.$$

Like functors, adjunctions can be composed. Given adjunctions $\mathsf{L}_1 \dashv \mathsf{R}_1 : \mathbb{C} \to \mathbb{D}$ and $\mathsf{L}_2 \dashv \mathsf{R}_2 : \mathbb{D} \to \mathbb{E}$, their *composition*

$$\mathbb{C} \underset{\mathsf{R}_2}{\overset{\mathsf{L}_2}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbb{D} \underset{\mathsf{R}_1}{\overset{\mathsf{L}_1}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbb{E}$$

yields an adjunction $\mathsf{L}_2 \circ \mathsf{L}_1 \dashv \mathsf{R}_1 \circ \mathsf{R}_2 : \mathbb{C} \to \mathbb{E}$. Observe that the right adjoints are composed in the reverse order. To establish $\mathbb{C}(\mathsf{L}_2\,(\mathsf{L}_1\,-), =) \cong \mathbb{E}(-, \mathsf{R}_1\,(\mathsf{R}_2\,=))$, we compose the two adjoint transpositions.

$$\begin{aligned} & \mathbb{C}(\mathsf{L}_2\,(\mathsf{L}_1\,-), =) \\ \cong \quad & \{ \text{ assumption: } \mathsf{L}_2 \dashv \mathsf{R}_2 \ \} \\ & \mathbb{D}(\mathsf{L}_1\,-, \mathsf{R}_2\,=) \\ \cong \quad & \{ \text{ assumption: } \mathsf{L}_1 \dashv \mathsf{R}_1 \ \} \\ & \mathbb{E}(-, \mathsf{R}_1\,(\mathsf{R}_2\,=)) \end{aligned}$$

If we view the nested functors $\mathsf{L}_2\,(\mathsf{L}_1\,-)$ as a list, then the proof corresponds to the efficient version of list reversal. Hence the reversal of the functors R_1 and R_2.

Composition of adjunctions is associative and has $\mathsf{Id} \dashv \mathsf{Id}$ as its neutral element. This data turns categories and adjunctions into a category, but we will not make use of this fact.

6. Fixpoints

The previous two sections have covered the full gamut of type isomorphisms inspired by high-school algebra, and we have done so using the power of adjunctions and the Yoneda lemma. So far all our examples have been on non-recursive types. The goal of this section is to demonstrate various isomorphisms between list and tree types. Since these types are given by recursive definitions we have to delve into the theory of fixpoints [3].

In the introduction we gave Haskell definitions to three tree types, and in which we made use of the inbuilt list type. As a reminder, here it is explicitly.

$$\mathbf{data}\ \mathsf{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ (a, \mathsf{List}\ a)$$

To make use of List and the three tree types in our categorical setting, we will rewrite them as least fixpoints of functors.

$$\begin{aligned} \mathsf{List} &= \Lambda\, A\, .\, \mu\, L\, .\ 1 + A \times L \\ \mathsf{Rose} &= \Lambda\, A\, .\, \mu\, R\, .\ A \times \mathsf{List}\ R \\ \mathsf{Tree} &= \Lambda\, A\, .\, \mu\, T\, .\, 1 + A \times T \times T \\ \mathsf{Fork} &= \Lambda\, A\, .\, \mu\, F\, .\ A + F \times F \end{aligned}$$

Here $\mu\, X\, .\, e$ is a shorthand for $\mu\,(\Lambda\, X\, .\, e)$.

The introduction gave a Haskell witness to the isomorphism $\mathsf{Tree}\ A \cong \mathsf{List}\,(\mathsf{Rose}\ A)$. Later will give a categorical proof for the existence of this isomorphism. We will we also prove $\mathsf{Tree}\ A \times A \cong \mathsf{Fork}\ A$, $A \times \mathsf{Tree}\ A \cong \mathsf{Rose}\ A$ and $\mathsf{Rose}\ A \cong \mathsf{Fork}\ A$.

For reasons of brevity, we will not provide proofs for all of the obligations that arise in this section.

6.1 Algebras

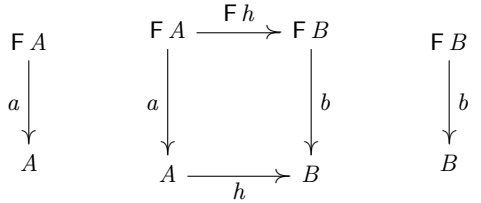
The classic fixpoint theorem due to Knaster and Tarski states that for an order-preserving function $f : L \to L$ over a complete lattice $\langle L, \leqslant \rangle$,

$$\bigsqcap \{\, x \mid fx = x \,\} = \bigsqcap \{\, x \mid fx \leqslant x \,\}\ , \tag{46}$$

$$\bigsqcup \{\, x \mid fx = x \,\} = \bigsqcup \{\, x \mid x \leqslant fx \,\} \tag{47}$$

are the least and greatest fixpoints of f. That is to say, the least fixpoint of f equals its least *pre-fixpoint*, and dually the greatest fixpoint equals its greatest *post-fixpoint*. The categorical counterpart of a pre-fixpoint is an algebra, and coalgebras are the categorical counterpart of post-fixpoints.

Let $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ be a functor. An F-*algebra* is a pair $\langle A, a \rangle$ consisting of an object $A : \mathbb{C}$ and an arrow $a : \mathbb{C}(\mathsf{F}\ A, A)$. The object A is the actual pre-fixpoint of F, the arrow a can be seen as a witness of this fact. If the witness is an isomorphism, then A is a fixpoint of F. An F-*homomorphism* between algebras $\langle A, a \rangle$ and $\langle B, b \rangle$ is an arrow $h : \mathbb{C}(A, B)$ such that $h \cdot a = b \cdot \mathsf{F}\ h$.



The fact that functors preserve identity and composition entails that identity is an F-homomorphism and that F-homomorphisms compose. Consequently, the data defines a category of F-algebras and F-homomorphisms, called $\mathbf{Alg}(\mathsf{F})$. The fixpoints $\langle A, a : \mathsf{F}\ A \cong A \rangle$ of F form a full subcategory of $\mathbf{Alg}(\mathsf{F})$, denoted $\mathbf{Fix}(\mathsf{F})$. The initial object of $\mathbf{Alg}(\mathsf{F})$, the initial F-algebra $\langle \mu\mathsf{F}, \mathit{in} \rangle$, corresponds to the least pre-fixpoint, and the initial object of $\mathbf{Fix}(\mathsf{F})$ corresponds to the least fixpoint. Therefore the categorical counterpart of (46) is that these initial objects in $\mathbf{Alg}(\mathsf{F})$ and $\mathbf{Fix}(\mathsf{F})$ are equal. We ignore questions of existence in this paper.

All of this dualises to coalgebras, where an F-*coalgebra* is a pair $\langle A, a \rangle$ consisting of an object $A : \mathbb{C}$ and an arrow $a : \mathbb{C}(A, \mathsf{F}\ A)$. Again, we have a category of F-coalgebras and F-homomorphisms, called $\mathbf{Coalg}(\mathsf{F})$, with the same relation to $\mathbf{Fix}(\mathsf{F})$. The final object of $\mathbf{Coalg}(\mathsf{F})$, the final F-coalgebra $\langle \nu\mathsf{F}, \mathit{out} \rangle$, corresponds to the greatest post-fixpoint, and coincides with the final object of $\mathbf{Fix}(\mathsf{F})$, the greatest fixpoint.

Interestingly, we can also obtain least and greatest fixpoints through an adjunction. There is a canonical functor from the category of algebras to the underlying category, the so-called forgetful or underlying functor $\mathsf{U} : \mathbf{Alg}(\mathsf{F}) \to \mathbb{C}$ defined $\mathsf{U}\langle A, f \rangle = A$ and $\mathsf{U}\ h = h$. While its definition is deceptively simple, it gives rise to two interesting concepts via a double adjunction.

$$\mathbf{Alg}(\mathsf{F}) \underset{\mathsf{U}}{\overset{\mathsf{Free}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbb{C} \underset{\mathsf{Cofree}}{\overset{\mathsf{U}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbf{Coalg}(\mathsf{F})$$

The functor Free maps an object A to the so-called free F-algebra over A. Dually, Cofree maps an object A to the cofree F-coalgebra over A.[2] Since left adjoints preserve initial objects and right adjoints preserve final ones, we can alternatively define μ and ν by $\mu\mathsf{F} := \mathsf{U}\ 0 \cong \mathsf{U}\,(\mathsf{Free}\ 0)$ and $\nu\mathsf{F} := \mathsf{U}\ 1 \cong \mathsf{U}\,(\mathsf{Cofree}\ 1)$.

6.2 Functor Rules

Both μ and ν are actually functors of type $\mathbb{C}^{\mathbb{C}} \to \mathbb{C}$. They take endofunctors, objects of the category $\mathbb{C}^{\mathbb{C}}$, to objects in $\mathbb{C}$, and they take natural transformations, arrows of the category $\mathbb{C}^{\mathbb{C}}$, to arrows in $\mathbb{C}$. For a functor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$, μ maps it to $\mu\mathsf{F}$, and for another functor $\mathsf{G} : \mathbb{C} \to \mathbb{C}$ and a natural transformation $\mathfrak{a} : \mathsf{F} \dot{\to} \mathsf{G}$, $\mu\mathfrak{a} : \mathbb{C}(\mu\mathsf{F}, \mu\mathsf{G})$. (The functor μ is only well defined for functors that have an initial algebra; we have to assume that $\mathbb{C}$ is ω-cocomplete and that $\mathbb{C}^{\mathbb{C}}$ is really the subcategory of ω-cocontinuous endofunctors.) Using the fact that functors preserve isomorphism (5), we have that two recursive types are isomorphic if the base functors are:

[2] Every adjunction gives rise to a monad, and $\mathsf{U} \circ \mathsf{Free}$ is called the free monad of the functor F. Dually, $\mathsf{U} \circ \mathsf{Cofree}$ is the cofree comonad.

$$\mu \mathsf{F} \cong \mu \mathsf{G} \impliedby \mathsf{F} \cong \mathsf{G} \ , \tag{48}$$

$$\nu \mathsf{F} \cong \nu \mathsf{G} \impliedby \mathsf{F} \cong \mathsf{G} \ . \tag{49}$$

We can use (48) to show, for example, that cons and snoc lists are isomorphic. Fixing some element type A, cons and snoc lists are initial algebras of the base functors $\Lambda\, X \ .\ 1 + A \times X$ and $\Lambda\, X \ .\ 1 + X \times A$, respectively. We reason

$$\begin{array}{ll} & \mu\, X \ .\ 1 + A \times X \cong \mu\, X \ .\ 1 + X \times A \\ \impliedby & \quad \{\ (48)\ \} \\ & \Lambda\, X \ .\ 1 + A \times X \cong \Lambda\, X \ .\ 1 + X \times A \\ \iff & \quad \{\ \times \text{ is commutative (26)}\ \} \\ & \Lambda\, X \ .\ 1 + X \times A \cong \Lambda\, X \ .\ 1 + X \times A \end{array}$$

The proof is simple enough, but it raises an interesting question: why can we replace a term below a Λ-abstraction? Every Λ-term denotes a functor [12]. As their functorial nature preserves isomorphism, we can manipulate below a Λ. Our appeal to the commutativity of $\times$ is really:

$$\begin{array}{ll} & (\Lambda\, \mathsf{F} \ .\ \Lambda\, X \ .\ 1 + \mathsf{F}\, X)\, (\Lambda\, X \ .\ A \times X) \\ \cong & \quad \{\ A \times - \cong - \times A\ \} \\ & (\Lambda\, \mathsf{F} \ .\ \Lambda\, X \ .\ 1 + \mathsf{F}\, X)\, (\Lambda\, X \ .\ X \times A) \end{array}$$

Note that we require a *natural* isomorphism between $A \times -$ and $- \times A$.

6.3 Type fusion

Suppose we have an initial F-algebra $\langle \mu\mathsf{F}, in : \mathsf{F}\, \mu\mathsf{F} \cong \mu\mathsf{F} \rangle$, then the isomorphism in lets us fold and unfold from $\mathsf{F}\, (\mu\mathsf{F})$ to $\mu\mathsf{F}$ and back. This is an elementary rule of fixpoint calculus, but what if we want to *fuse* the application of a different functor? Suppose that F is an endofunctor in a category $\mathbb{D}$ and we have a functor $\mathsf{L} : \mathbb{D} \to \mathbb{C}$. Can we turn $\mathsf{L}\, (\mu\mathsf{F}) : \mathbb{C}$ into another initial algebra, say $\mu\mathsf{G} : \mathbb{C}$, for some $\mathsf{G} : \mathbb{C} \to \mathbb{C}$? The answer is yes, under certain conditions. To motivate the necessary conditions, let us consider lifting the functor L to a functor between the categories of algebras, $\mathbf{Alg}(\mathsf{F})$ and $\mathbf{Alg}(\mathsf{G})$. Given an F-algebra $\langle A, a : \mathsf{F}\, A \to A \rangle$, we can apply L to a to obtain an arrow $\mathsf{L}\, a : \mathsf{L}\, (\mathsf{F}\, A) \to \mathsf{L}\, A$. To be able to construct a G-algebra $\langle \mathsf{L}\, A, x : \mathsf{G}\, (\mathsf{L}\, A) \to \mathsf{L}\, A \rangle$, we have to demand that $\mathsf{L}\, (\mathsf{F}\, A) \cong \mathsf{G}\, (\mathsf{L}\, A)$. This isomorphism should hold uniformly for all A, so we assume the existence of a natural isomorphism $swap : \mathsf{L} \circ \mathsf{F} \cong \mathsf{G} \circ \mathsf{L}$. The lifted variant of L is then $\underline{\mathsf{L}} : \mathbf{Alg}(\mathsf{F}) \to \mathbf{Alg}(\mathsf{G})$ with $\underline{\mathsf{L}} \langle A, a \rangle = \langle \mathsf{L}\, A, \mathsf{L}\, a \cdot swap^\circ \rangle$ and $\underline{\mathsf{L}}\, h = \mathsf{L}\, h$.

Our lifted functor $\underline{\mathsf{L}}$ should preserve initial algebras, which is certainly the case if $\underline{\mathsf{L}}$ is a left adjoint. It turns out that $\underline{\mathsf{L}}$ is a left adjoint, if the original functor L is one. This motivates the two type fusion rules:

Let $\mathbb{C}$ and $\mathbb{D}$ be categories, let $\mathsf{L} \dashv \mathsf{R} : \mathbb{C} \to \mathbb{D}$ be an adjoint pair of functors, and let $\mathsf{F} : \mathbb{D} \to \mathbb{D}$ and $\mathsf{G} : \mathbb{C} \to \mathbb{C}$ be two endofunctors. Then,

$$\mathsf{L}\, (\mu\mathsf{F}) \cong \quad \mu\mathsf{G} \quad \impliedby \quad \mathsf{L} \circ \mathsf{F} \cong \mathsf{G} \circ \mathsf{L} \ , \tag{50}$$

$$\nu\mathsf{F} \quad \cong \mathsf{R}\, (\nu\mathsf{G}) \impliedby \mathsf{F} \circ \mathsf{R} \cong \mathsf{R} \circ \mathsf{G} \ . \tag{51}$$

Before we sketch the proof, let us first consider two examples — we shall cover more applications in Section 6.6. As the identity functor is left adjoint to itself, we can set L and R to be Id and get (48) and (49) as special cases of type fusion.

The Haskell programmer's favourite adjunction is currying: $- \times A \dashv (-)^A$. This adjunction allows us to relate binary trees, where the values are stored in the branches, to leaf trees, where the values are stored in the leaves. Here we take the functor $\times A$ as L, $\mathsf{Tree}\, A$ as $\mu\mathsf{F}$ and $\mathsf{Fork}\, A$ as $\mu\mathsf{G}$.

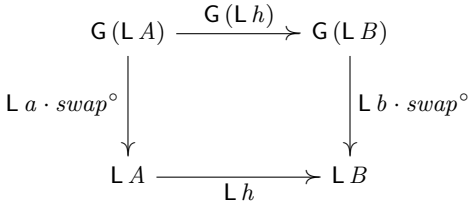
$$\begin{array}{ll} & \mathsf{Tree}\, A \times A \\ \cong & \quad \{\ \text{definition of } \mathsf{Tree}\ \} \\ & (\mu\, T \ .\ 1 + A \times T \times T) \times A \\ \cong & \quad \{\ \text{type fusion (50), see proof obligation below}\ \} \\ & \mu\, T \ .\ A + T \times T \\ \cong & \quad \{\ \text{definition of } \mathsf{Fork}\ \} \\ & \mathsf{Fork}\, A \end{array}$$

For the fusion step we have to show that $(\times A) \circ (\Lambda\, T \ .\ 1 + A \times T \times T) \cong (\Lambda\, T \ .\ A + T \times T) \circ (\times A)$. The obligation is quick to discharge:

$$\begin{array}{ll} & (1 + A \times T \times T) \times A \\ \cong & \quad \{\ \text{distributivity (39) and neutral (28)}\ \} \\ & A + (A \times T \times T) \times A \\ \cong & \quad \{\ \text{associativity (30)}\ \} \\ & A + (A \times T) \times (T \times A) \\ \cong & \quad \{\ \text{commutativity (29)}\ \} \\ & A + (T \times A) \times (T \times A) \ . \end{array}$$

Turning to the proof of type fusion, the essential idea is that the adjunction $\phi : \mathsf{L} \dashv \mathsf{R}$ between the base categories can be lifted to an adjunction $\phi : \underline{\mathsf{L}} \dashv \underline{\mathsf{R}}$ between the corresponding algebras. The proof proceeds in three steps:

First, we have to show that $\underline{\mathsf{L}}$ is indeed a functor between $\mathbf{Alg}(\mathsf{F})$ and $\mathbf{Alg}(\mathsf{G})$. If h is an F-homomorphism between algebras $\langle A, a \rangle$ and $\langle B, b \rangle$ in the category $\mathbf{Alg}(\mathsf{F})$, then there is an arrow $h : \mathbb{D}(A, B)$ such that $h \cdot a = b \cdot \mathsf{F}\, h$. To show that $\underline{\mathsf{L}}$ is a functor, we must show that it takes an F-homomorphism such as h to a G-homomorphism. The G-homomorphism $\underline{\mathsf{L}}\, h$ is between algebras $\underline{\mathsf{L}}\, \langle A, a \rangle$ and $\underline{\mathsf{L}}\, \langle B, b \rangle$ with an arrow $\mathsf{L}\, h : \mathbb{C}(\mathsf{L}\, A, \mathsf{L}\, B)$ such that the following diagram commutes.



So, we have to show that $\mathsf{L}\, h \cdot (\mathsf{L}\, a \cdot swap^\circ) = (\mathsf{L}\, b \cdot swap^\circ) \cdot \mathsf{G}\, (\mathsf{L}\, h)$:

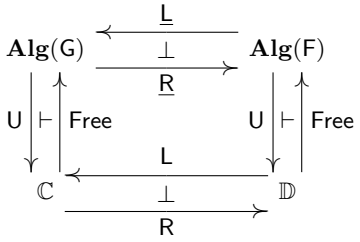
$$\begin{array}{ll} & \mathsf{L}\, h \cdot (\mathsf{L}\, a \cdot swap^\circ) \\ = & \quad \{\ \mathsf{L} \text{ functor}\ \} \\ & \mathsf{L}\, (h \cdot a) \cdot swap^\circ) \\ = & \quad \{\ h \text{ is an } \mathsf{F}\text{-homomorphism}\ \} \\ & \mathsf{L}\, (b \cdot \mathsf{F}\, h) \cdot swap^\circ) \\ = & \quad \{\ \mathsf{L} \text{ functor}\ \} \\ & \mathsf{L}\, b \cdot \mathsf{L}\, (\mathsf{F}\, h) \cdot swap^\circ) \\ = & \quad \{\ \text{naturality of } swap^\circ : \mathsf{G} \circ \mathsf{L} \dot{\to} \mathsf{L} \circ \mathsf{F}\ \} \\ & (\mathsf{L}\, b \cdot swap^\circ) \cdot \mathsf{G}\, (\mathsf{L}\, h) \ . \end{array}$$

Note that the proof works for an arbitrary natural transformation of type $\mathsf{G} \circ \mathsf{L} \dot{\to} \mathsf{L} \circ \mathsf{F}$.

Next, we have to lift the right adjoint R to a functor between algebras. The approach is the same as for L, but this time we require a natural transformation (not necessarily an isomorphism) of type $\mathsf{F} \circ \mathsf{R} \dot{\to} \mathsf{R} \circ \mathsf{G}$. It turns out that we can derive the required transformation from $swap$ — the details are beyond the scope of this paper.

Finally, and this is the most laborious part, we have to show that the adjoint transposition of $\mathsf{L} \dashv \mathsf{R}$ also serves as the adjoint transposition of $\underline{\mathsf{L}} \dashv \underline{\mathsf{R}}$, that is, it preserves and reflects homomorphisms. Again, the details are beyond the scope of this paper.

As an intermediate summary, we have established the following square of adjunctions.



Adjoint functors compose, see Section 5.3, so the square gives rise to two composite adjunctions:

$$\mathsf{Free} \circ \mathsf{L} \dashv \mathsf{R} \circ \mathsf{U} \quad \text{and} \quad \underline{\mathsf{L}} \circ \mathsf{Free} \dashv \mathsf{U} \circ \underline{\mathsf{R}} \ .$$

By definition, $\mathsf{L} \circ \mathsf{U} = \mathsf{U} \circ \underline{\mathsf{L}}$ and $\mathsf{R} \circ \mathsf{U} = \mathsf{U} \circ \underline{\mathsf{R}}$ — note the equality sign. In other words, $\mathsf{R} \circ \mathsf{U}$ has two left adjoints. However, left adjoints are unique up to isomorphism, so we can immediately conclude that

$$\mathsf{Free} \circ \mathsf{L} \ \cong \ \underline{\mathsf{L}} \circ \mathsf{Free} \ . \tag{52}$$

The natural isomorphism lives in $\mathbf{Alg}(\mathsf{G})$, using U we can 'lower' it to an isomorphism in $\mathbb{C}$.

$$\begin{array}{ll} & \mathsf{U} \circ \mathsf{Free} \circ \mathsf{L} \\ \cong & \quad \{ \ (52) \ \} \\ & \mathsf{U} \circ \underline{\mathsf{L}} \circ \mathsf{Free} \\ \cong & \quad \{ \ \mathsf{L} \circ \mathsf{U} = \mathsf{U} \circ \underline{\mathsf{L}} \ \} \\ & \mathsf{L} \circ \mathsf{U} \circ \mathsf{Free} \end{array}$$

Note that $\mathsf{U} \circ \mathsf{Free}$ appears twice, with the first instance in respect to G, that is, $\mathsf{U} \circ \mathsf{Free} : \mathbb{C} \to \mathbf{Alg}(\mathsf{G}) \to \mathbb{C}$ and the second in respect to F, that is, $\mathsf{U} \circ \mathsf{Free} : \mathbb{D} \to \mathbf{Alg}(\mathsf{F}) \to \mathbb{D}$. To highlight this we write the above isomorphism again replacing $\mathsf{U} \circ \mathsf{Free}$ by the more informative notation F^* and G^*.

$$\mathsf{L} \circ \mathsf{F}^* \cong \mathsf{G}^* \circ \mathsf{L} \iff \mathsf{L} \circ \mathsf{F} \cong \mathsf{G} \circ \mathsf{L} \ .$$

This generalises (50). Recall that the free functor Free is left adjoint and so preserves 0. It takes the initial object in $\mathbb{C}$ to the initial object object in $\mathbf{Alg}(\mathsf{F})$, $\langle \mu\mathsf{F}, in : \mathsf{F}\,\mu\,\mathsf{F} \cong \mu\mathsf{F} \rangle$. Therefore $\mu\mathsf{F} := \mathsf{U}\,0 \cong \mathsf{U}\,(\mathsf{Free}\,0)$, or using the alternative notation, $\mu\mathsf{F} := \mathsf{F}^*\,0$. Type fusion emerges then as a special case:

$$\mathsf{L}\,(\mu\mathsf{F}) \cong \mathsf{L}\,(\mathsf{F}^*\,0) \cong \mathsf{G}^*\,(\mathsf{L}\,0) \cong \mathsf{G}^*\,0 \cong \mu\mathsf{G} \ .$$

6.4 Rolling rules

A simple way to satisfy the precondition in (50) is to set $\mathsf{F} := \mathsf{H} \circ \mathsf{L}$ and $\mathsf{G} := \mathsf{L} \circ \mathsf{H}$ so that *swap* is the identity transformation. It turns out that in this particular case we can forego the assumption that L is a left adjoint. Renaming L and H, we obtain the *rolling rules:*

$$\mathsf{F}\,(\mu(\mathsf{G} \circ \mathsf{F})) \cong \mu(\mathsf{F} \circ \mathsf{G}) \ , \tag{53}$$

$$\mathsf{F}\,(\nu(\mathsf{G} \circ \mathsf{F})) \cong \nu(\mathsf{F} \circ \mathsf{G}) \ . \tag{54}$$

Despite appearances, the rules describe a perfectly symmetric situation. We shall see that the lifted functor $\underline{\mathsf{F}}$, turns a fixpoint of $\mathsf{G} \circ \mathsf{F}$ into a fixpoint of $\mathsf{F} \circ \mathsf{G}$ with $\underline{\mathsf{G}}$ going in the opposite direction. (If we interchange the functors F and G above, we obtain a second set of rules for the reverse direction.)

$$\mathbf{Fix}(\mathsf{F} \circ \mathsf{G}) \underset{\underline{\mathsf{G}}}{\overset{\underline{\mathsf{F}}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\simeq}}}} \mathbf{Fix}(\mathsf{G} \circ \mathsf{F}) \tag{55}$$

Before we tackle the proof, let us first consider an example. The rolling rule (53) allows us to relate the standard list type to the type of non-empty lists.

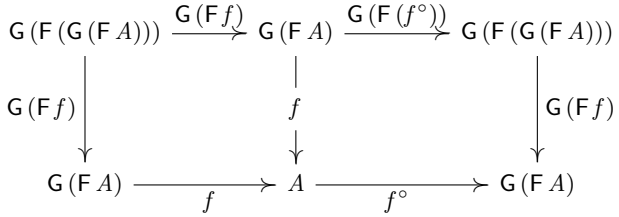
$$A \times (\mu\, X \ .\ 1 + A \times X) \cong \qquad (\mu\, X \ .\ A + A \times X)$$

$$(\mu\, X \ .\ 1 + A \times X) \cong 1 + (\mu\, X \ .\ A + A \times X)$$

The functors of (53) are instantiated as $\mathsf{F} := A \times$ and $\mathsf{G} := 1 +$. The first isomorphism can also be seen as an instance of type fusion, but not the second as $1 +$ is not a left adjoint.

Turning to the proof of the rolling rules, since *swap* is now the identity, the definition of a lifted F simplifies somewhat: $\underline{\mathsf{F}} : \mathbf{Alg}(\mathsf{G} \circ \mathsf{F}) \to \mathbf{Alg}(\mathsf{F} \circ \mathsf{G})$ is given by $\underline{\mathsf{F}}\langle A, f\rangle = \langle \mathsf{F}\,A, \mathsf{F}\,f\rangle$ and $\underline{\mathsf{F}}\,h = \mathsf{F}\,h$. The typings work out nicely: if $f : \mathbb{D}(\mathsf{G}\,(\mathsf{F}\,A), A)$, then $\mathsf{F}\,f : \mathbb{C}(\mathsf{F}\,(\mathsf{G}\,(\mathsf{F}\,A)), \mathsf{F}\,A)$. Since functors preserve isomorphisms, $\underline{\mathsf{F}}$ furthermore takes fixpoints of $\mathsf{G} \circ \mathsf{F}$ to fixpoints of $\mathsf{F} \circ \mathsf{G}$, see Diagram (55).

Moreover, the categories $\mathbf{Fix}(\mathsf{G} \circ \mathsf{F})$ and $\mathbf{Fix}(\mathsf{F} \circ \mathsf{G})$ are equivalent: there are natural isomorphisms $\mathfrak{c} : \underline{\mathsf{G}} \circ \underline{\mathsf{F}} \cong \mathsf{Id}_{\mathbf{Fix}(\mathsf{G} \circ \mathsf{F})}$ and $\mathfrak{n} : \underline{\mathsf{F}} \circ \underline{\mathsf{G}} \cong \mathsf{Id}_{\mathbf{Fix}(\mathsf{F} \circ \mathsf{G})}$. The definition of the isomorphism is surprisingly simple. Recall that a natural transformation maps objects to arrows: $\mathfrak{c}$ has to map a $\mathsf{G} \circ \mathsf{F}$-algebra, say, $\langle A, f\rangle$ to a $\mathsf{G} \circ \mathsf{F}$-homomorphism $\mathbf{Alg}(\mathsf{G} \circ \mathsf{F})(\underline{\mathsf{G}}\,(\underline{\mathsf{F}}\langle A, f\rangle), \langle A, f\rangle)$, that is, an arrow $\mathbb{D}(\mathsf{G}\,(\mathsf{F}\,A), A)$. The arrow f is of this type, which suggests that we define $\mathfrak{c}$ as $\mathfrak{c}\langle A, f\rangle = f$ and $\mathfrak{c}^\circ\langle A, f\rangle = f^\circ$. The diagram below shows that the components of $\mathfrak{c}$ and $\mathfrak{c}^\circ$ are indeed $\mathsf{G} \circ \mathsf{F}$-homomorphisms — it is immediate that they are inverses.



It remains to prove that $\mathfrak{c}$ and $\mathfrak{c}^\circ$ are natural transformations. Let $h : \mathbf{Alg}(\mathsf{G} \circ \mathsf{F})(\langle A, f\rangle, \langle B, g\rangle)$ be a $\mathsf{G} \circ \mathsf{F}$-homomorphism, then the naturality condition,

$$\begin{array}{ll} & h \cdot \mathfrak{c}\langle A, f\rangle = \mathfrak{c}\langle B, g\rangle \cdot \mathsf{G}\,(\mathsf{F}\,h) \\ \iff & \quad \{ \text{ definition of } \mathfrak{c} \ \} \\ & h \cdot f = g \cdot \mathsf{G}\,(\mathsf{F}\,h) \end{array}$$

is just the definition of a $\mathsf{G} \circ \mathsf{F}$-homomorphism.

An equivalence of categories gives rise to two adjunctions: $\mathsf{F} \dashv \mathsf{G}$ and $\mathsf{G} \dashv \mathsf{F}$, see [16]. Consequently, F and G preserve both initial and final objects.

6.5 Diagonal Rules

To be able to establish the natural correspondence between binary trees and forests, see Figure 1, we need one further rule.

Let $\mathsf{F} : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ be a bifunctor. Then,

$$\mu\, X \ .\ \mu\, Y \ .\ \mathsf{F}\langle X, Y\rangle \ \cong \ \mu\, Z \ .\ \mathsf{F}\langle Z, Z\rangle \ , \tag{56}$$

$$\nu\, X \ .\ \nu\, Y \ .\ \mathsf{F}\langle X, Y\rangle \ \cong \ \nu\, Z \ .\ \mathsf{F}\langle Z, Z\rangle \ . \tag{57}$$

The diagonal rules allow us to rewrite a nested fixpoint to a simple fixpoint along the diagonal of the functor F. The rules look innocent enough, but they are needed in every single example in Section 6.6. The proof of the two rules is very technical and omitted.

6.6 Examples

It is time to pick the fruit. Armed with the laws of high-school algebra and fixpoint calculus, we can now tackle the problems set out in the introduction.

The proof of the natural correspondence between forests and binary trees makes essential use of the rolling rule.

$$
\begin{array}{ll}
& \mathsf{List}\,(\mathsf{Rose}\,A) \\
\cong & \quad \{\ \text{definition of } \mathsf{Rose}\ \} \\
& \mathsf{List}\,(\mu\,R\,.\,A \times \mathsf{List}\,R) \\
\cong & \quad \{\ \text{rolling rule (53): } \mathsf{F} := \mathsf{List} \text{ and } \mathsf{G} := A\times\ \} \\
& \mu\,R\,.\,\mathsf{List}\,(A \times R) \\
\cong & \quad \{\ \text{definition of } \mathsf{List}\ \} \\
& \mu\,R\,.\,\mu\,L\,.\,1 + A \times R \times L \\
\cong & \quad \{\ \text{diagonal rule (56)}\ \} \\
& \mu\,T\,.\,1 + A \times T \times T \\
\cong & \quad \{\ \text{definition of } \mathsf{Tree}\ \} \\
& \mathsf{Tree}\,A
\end{array}
$$

Perhaps surprisingly, high-school algebra is not needed — the proof works solely by rearranging the recursive structure. In Haskell, the situation is more subtle, as nested pairs $((A, B), C)$ and triples (A, B, C) are distinguished. It is worth noting that the correspondence is not only natural in a data-structural sense, but also in a categorical one: $\mathsf{List} \circ \mathsf{Rose} \cong \mathsf{Tree}$.

The isomorphism between topped binary trees and rose trees is a direct consequence of the above.

$$
\begin{array}{ll}
& A \times \mathsf{Tree}\,A \\
\cong & \quad \{\ \text{see above}\ \} \\
& A \times \mathsf{List}\,(\mathsf{Rose}\,A) \\
\cong & \quad \{\ \mathsf{F}\,(\mu\,\mathsf{F}) \cong \mu\,\mathsf{F}\ \} \\
& \mathsf{Rose}\,A
\end{array}
$$

The next isomorphism illustrates the use of type fusion. Both rose trees and fork trees can be seen as representing non-empty sequences. To transform one into the other requires some shuffling though, implemented by type fusion.

$$
\begin{array}{ll}
& \mathsf{Rose}\,A \\
\cong & \quad \{\ \text{definition of } \mathsf{Rose}\ \} \\
& \mu\,R\,.\,A \times \mathsf{List}\,R \\
\cong & \quad \{\ \text{definition of } \mathsf{List}\ \} \\
& \mu\,R\,.\,A \times (\mu\,L\,.\,1 + R \times L) \\
\cong & \quad \{\ \text{type fusion (50)}\ \} \\
& \mu\,R\,.\,\mu\,L\,.\,A + R \times L \\
\cong & \quad \{\ \text{diagonal rule (56)}\ \} \\
& \mu\,T\,.\,A + T \times T \\
\cong & \quad \{\ \text{definition of } \mathsf{Fork}\ \} \\
& \mathsf{Fork}\,A
\end{array}
$$

For the fusion step we have to show that $(A\times)\circ(\Lambda\,L\,.\,1+R\times L) \cong (\Lambda\,L\ .\ A + R \times L) \circ (A\times)$. The obligation is again quick to discharge:

$$
\begin{array}{ll}
& A \times (1 + R \times L) \\
\cong & \quad \{\ \text{distributivity (39) and neutral (28)}\ \} \\
& A + A \times (R \times L) \\
\cong & \quad \{\ \text{associativity (30) and commutativity (29)}\ \} \\
& A + R \times (A \times L)\ .
\end{array}
$$

The use of commutativity indicates that the relative ordering of elements is changed: the root of the rose tree becomes the rightmost element of the fork tree.

The five type constructors 0, 1, $+$, $\times$ and List almost form a so-called Kleene algebra, the algebraic underpinning of regular expressions. The only difference is that, in a Kleene algebra, $+$ is required to be idempotent ($a + a = a$), which is, in general, not the case for coproducts. Nonetheless, some laws of Kleene algebra carry over to our setting, for instance, $\mathsf{List}\,(A + B) \cong \mathsf{List}\,A \times \mathsf{List}\,(B \times \mathsf{List}\,A)$. Here is the proof which showcases all three fixpoint rules:

$$
\begin{array}{ll}
& \mathsf{List}\,A \times \mathsf{List}\,(B \times \mathsf{List}\,A) \\
\cong & \quad \{\ \text{definition of } \mathsf{List}\ \} \\
& \mathsf{List}\,A \times (\mu\,L\,.\,1 + B \times \mathsf{List}\,A \times L) \\
\cong & \quad \{\ \text{rolling rule (53): } \mathsf{F} := \mathsf{List}\,A\times \text{ and } \mathsf{G}\,X := 1 + B \times X\ \} \\
& \mu\,L\,.\,\mathsf{List}\,A \times (1 + B \times L) \\
\cong & \quad \{\ \text{definition of } \mathsf{List}\ \} \\
& \mu\,L\,.\,(\mu\,X\,.\,1 + A \times X) \times (1 + B \times L) \\
\cong & \quad \{\ \text{type fusion (50)}\ \} \\
& \mu\,L\,.\,\mu\,X\,.\,1 + B \times L + A \times X \\
\cong & \quad \{\ \text{diagonal rule (56)}\ \} \\
& \mu\,L\,.\,1 + B \times L + A \times L \\
\cong & \quad \{\ \text{distributivity (39) and commutativity (26)}\ \} \\
& \mu\,L\,.\,1 + (A + B) \times L \\
\cong & \quad \{\ \text{definition of } \mathsf{List}\ \} \\
& \mathsf{List}\,(A + B)\ .
\end{array}
$$

Further applications of type fusion can be found in the recent paper "Type Fusion" by the first author [13].

7. Further Reading and Related Work

Every textbook on category theory defines the notions of isomorphism and equivalence, see, for example [5, 16]. However, the calculational properties of isomorphisms are mostly ignored. For example, the all important fact that functors preserve isomorphisms, on which our isomorphic reasoning rests, is often mentioned only in passing. This omission has a profound impact on the style of proofs, which typically involve chasing arrows in a diagram or arguing informally with and about arrows. As this paper amply demonstrates, our own preference is a calculational style, showing the equality of two arrows or the isomorphism of two objects by a series of meaning-preserving steps.

Central to the latter undertaking is the Yoneda lemma, which support an indirect mode of reasoning. Because of its central importance it also appears in every textbook on category theory, [5, 16] being no exceptions. Unfortunately, often only the abstract result is presented, leaving readers unenlightened and depriving them of a powerful reasoning tool. We have hinted at another link to computing science: continuation-passing style.

Likewise, adjunctions are directly relevant to computing science and programming. An adjunction bundles a programming language construct in a single package: introduction and elimination rules are given by the adjoint transposition, β- and η-rules fall out of the isomorphism itself, fusion rules correspond to the naturality of the transposition. The standard example of an adjunction is the list functor $\mathsf{List} : \mathbf{Set} \to \mathbf{Mon}$, which is left adjoint to the forgetful functor $\mathsf{U} : \mathbf{Mon} \to \mathbf{Set}$ from the category of monoids to the category of sets [18]. Again, the calculational properties of adjunctions are usually neglected, a remarkable exception being the unpublished draft [10], which we highly recommend for further reading. (We also did not emphasise the calculational aspects as we approached adjunctions from a somewhat limited perspective.)

Along similar lines, [11] showcases a calculational approach to colimits and limits.

The use of initial algebras and fixpoints in categories to provide a semantics to datatypes originated with Lambek [15]. Following the suggestion of Lambek, Backhouse *et al.* [3] use lattice theory as a source of inspiration and generalise the lattice-theoretic fixpoint rules to category-theoretic rules, type fusion and the rolling rules among others. The development in Section 6 is based on their work. The original paper does not provide any proofs; they can be found in an unpublished draft [4]. The proof of type fusion, Section 6.3, draws on this work; the other proofs are ours. An alternative proof of type fusion, which makes the isomorphisms explicit, can be found in a recent paper [13] by the first author.

The interest in type isomorphisms is broad and is surveyed by Di Cosmo [7, 8]. Atanassow and Jeuring take up the task of inferring invertible coercions between isomorphic types using Generic Haskell [2]. They apply their technique to simplifying XML processing in Haskell. Also of note is the work of Fiore, which establishes the decidability of type isomorphisms for recursive polynomial types [9]; and of Soloviev, who gives an axiomatization of isomorphic types in symmetric monodial closed categories [19]. Finally, Mazur provides a more detailed and philosophical look at the question of isomorphism over equality [17].

In spirit, Cáccamo and Winskel's work is the closest to ours [6]. Their main contribution is a nice calculus where naturality pops out. Their focus is on more mundane topics, such as the categorical concepts of *ends* and *powers* that power the underlying theory of their calculus. There is no discussion of initial algebras

Our use of the colloquial term 'high-school algebra' is due to Thorsten Altenkirch [1]. The term was originally coined by Tarski, for the algebraic theory of a semiring with exponentiation; Altenkirch introduces 'university algebra'. High-school algebra, as in this paper, is interpreted in a lambda calculus with coproducts (bicartesian closed category); Altenkirch's university algebra is interpreted in a dependently typed lambda calculus.

8. Conclusion

We have offered a perspective on why isomorphism is a preferable notion to equality. Furthermore, we have stated and proven a range of isomorphisms on both non-recursive and recursive types. All functors *preserve* isomorphisms, but a key property at the base of our approach is that functors that are fully faithful also *reflect* isomorphisms.

The Yoneda lemma builds on this: it implies the proof principle of *indirect isomorphism*. The Yoneda functor Y, and its dual, $\bar{\mathsf{Y}}$, are fully faithful due to the Yoneda lemma. Thus, we can prove that two objects, A and B, are isomorphic by showing an isomorphism between the set of arrows from A and the set of arrows from B. We used this fact as a definitional principle for the arithmetic objects 0, 1, $+$, $\times$, and moreover, to prove that 0 and $+$ form a commutative monoid under isomorphism, as do 1 and $\times$. These statements are category agnostic.

Adjunctions have provided a powerful means to concisely package up lots of structure about the constructions we have explored. We have seen, for example, how they give meaning to $+$ and $\times$ as bifunctors, and how the adjoint transposition contains computational content: case analysis in the context of coproducts, and currying and function application in the context of exponentials. When we came to model recursive datatypes as least fixpoints of functors, the property that left adjoints preserve initial objects was invaluable. It is notable trend with adjunctions that we often discover interesting functors when we start with simple ones and explore the adjoints that arise from it. The diagonal functor is very simple, but its left adjoint $+$ and right adjoint $\times$ are interesting. Again, the forgetful functor is simple, even uninteresting, but its left adjoint Free and right adjoint Cofree are far from that.

Using adjunctions and the Yoneda lemma we have rounded out the categorical interpretation of high-school algebra under isomorphism, and brought the rules of fixpoint calculus into our category theory setting — the latter with an isomorphic twist. Having built our platform to 'reason isomorphically', we have presented some parting examples of isomorphisms on recursive datatypes in a clean calculational style.

References

[1] Altenkirch, T.: From High School to University Algebra (June 2008), `http://www.cs.nott.ac.uk/~txa/publ/unialg.pdf`

[2] Atanassow, F., Jeuring, J.: Inferring Type Isomorphisms Generically. In: Mathematics of Program Construction. LNCS, vol. 3125, pp. 32–53 (2004)

[3] Backhouse, R., Bijsterveld, M., van Geldrop, R., van der Woude, J.: Categorical fixed point calculus. In: Category Theory and Computer Science. LNCS, vol. 953, pp. 159–179 (1995)

[4] Backhouse, R., Bijsterveld, M., van Geldrop, R., van der Woude, J.: Category theory as coherently constructive lattice theory (2003), working Document, available from `http://www.cs.nott.ac.uk/~rcb/MPC/CatTheory.ps.gz`

[5] Barr, M., Wells, C.: Category Theory for Computing Science. Les Publications CRM, Montréal, 3rd edn. (1999), the book is available from Centre de recherches mathématiques `http://crm.umontreal.ca/`

[6] Cáccamo, M., Winskel, G.: A Higher-Order Calculus for Categories. In: Theorem Proving in Higher Order Logics. LNCS, vol. 2152, pp. 136–153 (2001)

[7] Di Cosmo, R.: The Isomorphisms of Types, `http://www.dicosmo.org/ResearchThemes/ISOS/ISOShomepage.html`

[8] Di Cosmo, R.: A short survey of isomorphisms of types. Math. Struct. in Comp. Sci. 15(05), 825–838 (2005)

[9] Fiore, M.: Isomorphisms of Generic Recursive Polynomial Types. In: Principles of programming languages. pp. 77–88. ACM (2004)

[10] Fokkinga, M.M., Meertens, L.: Adjunctions. Tech. Rep. Memoranda Inf 94-31, University of Twente, Enschede, Netherlands (June 1994)

[11] Fokkinga, M.M.: Calculate categorically! Formal Aspects of Computing 4(2), 673–692 (1992)

[12] Gibbons, J., Paterson, R.: Parametric datatype-genericity. In: Workshop on Generic programming. pp. 85–93. ACM Press (August 2009)

[13] Hinze, R.: Type fusion. In: Pavlovic, D., Johnson, M. (eds.) Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST2010) (2010), to appear

[14] Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley Publishing Company, 2nd edn. (1998)

[15] Lambek, J.: A fixpoint theorem for complete categories. Math. Zeitschr. 103, 151–161 (1968)

[16] Mac Lane, S.: Categories for the Working Mathematician. Graduate Texts in Mathematics, Springer-Verlag, Berlin, 2nd edn. (1998)

[17] Mazur, B.: When is one thing equal to some other thing? `http://abel.math.harvard.edu/~mazur/preprints/when_is_one.pdf` (Sept 2007)

[18] Rydeheard, D.: Adjunctions. In: Pitt, D., Poigne, A., Rydeheard, D. (eds.) Category Theory and Computer Science (1987), lNCS 283

[19] Soloviev, S.: A Complete Axiom System for Isomorphism of Types in Closed Categories. In: Logic Programming and Automated Reasoning. LNCS, vol. 698, pp. 360–371 (1993)