Category Theory for Program Construction by Calculation

Lambert Meertens
CWI, Amsterdam and Department of Computing Science, Utrecht University

September 5, 1995

Contents

Introduction

Category theory is the theory of *structure-preserving* transformations. This theory provides us with a *language* for describing complex problems in an elegant way, and *tools* to give elegant, and above all simple, proofs. The language of category theory allows us to consider the essence of some problem, without the burden of—often numerous and complicated—non-essential aspects.

The constructive and universal character of category theory is an aid in finding these proofs. In these lecture notes we emphasise a *calculational* proof style, to which category theory lends itself well. Examples are drawn from functional programming—and to a lesser extent from lattice theory—and in fact the applicability to calculational program construction has influenced the choice of topics covered.

Those familiar with publications on category theory may be surprised that this text doesn't contain any diagrams (pictures of objects connected by labelled arrows). Diagrams can be illuminating for small problems and save some writing—but it is nevertheless advisable to avoid the use of diagrams *as a substitute for proofs* for the following reasons. First, a diagram cannot make clear *in which order* the arrows were constructed and *why* the construction is indeed correct. Furthermore, a diagram is drawn within one particular category. We frequently deal with more than one category at once, and in such more complicated situations diagrams are hardly helpful.

Exercises

The exercises included in the text are there because they are meant to be done by the reader. They sometimes introduce concepts that are used later in the main text. Further, many have been selected to help the rather abstract theory come to life by relating them to concrete familiar structures. Finally, an important part of the course is to learn how to construct elegant proofs. That is something that simply can not be learned by only reading already constructed proofs, which afterwards look deceptively simple. Only by the effort of trying to construct new proofs, as required by the exercises, can the proof technique be mastered. None of the exercises is particularly difficult, although they gradually become more advanced; but the reader who has done all previous exercises should be able to finish each next one.

Acknowledgement

For this text use was made of unpublished material produced by the Mathematics of Programming group of Roland Backhouse at Eindhoven, much of which was written by Marcel Bijsterveld. However, substantive changes were made in content and presentation, and any shortcomings of the present text are the sole responsibility of the present author. The treatment owes much to earlier work by Grant Malcolm and Maarten Fokkinga.

The document was prepared using the Math$\int$pad editing system developed by the Mathematics of Programming group at Eindhoven.

Chapter 1

Notation

Proof style

Proofs are generally given in the Feijen-Dijkstra style. To show that proposition P follows from Q, a proof may look like this:

$$
\begin{array}{ll}
 & P \\
\Leftarrow & \quad \{ \quad \text{hint why } P \Leftarrow R \quad \} \\
 & R \\
\equiv & \quad \{ \quad \text{hint why } R \equiv S \quad \} \\
 & S \\
\Leftarrow & \quad \{ \quad \text{hint why } S \Leftarrow T \quad \} \\
 & T \\
\Leftarrow & \quad \{ \quad \text{hint why } T \Leftarrow U \quad \} \\
 & U \\
\equiv & \quad \{ \quad \text{hint why } U \equiv Q \quad \} \\
 & Q \ .
\end{array}
$$

We use this proof style also to solve equations for unknowns. Then hints of the form '$x := E$' may occur. The meaning is: this proof step is valid *if* we substitute E for the unknown x. The expression E may introduce further unknowns, which must be solved for subsequently. By collecting and performing all these substitutions, an expression for the unknown results. For example,

$$
\begin{array}{ll}
 & z^2 = 2i \\
\Leftarrow & \quad \{ \quad z := x + iy \quad \} \\
 & (x + iy)^2 = 2i
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{ \quad \text{algebra; } i^2 = -1 \quad \} \\
& (x^2 - y^2) + 2ixy = 2i \\
\Leftarrow & \{ \quad y := x \quad \} \\
& 2ix^2 = 2i \\
\Leftarrow & \{ \quad \textsc{Leibniz} \quad \} \\
& x^2 = 1 \\
\Leftarrow & \{ \quad x := 1 \quad \} \\
& \mathsf{true}
\end{array}
$$

is a proof of $(1 + i)^2 = 2i$.

The hint 'Leibniz' explains proof steps of the form '$E_x = E_y \Leftarrow x = y$', in which E_y is the same expression as E_x except for replacing one or more occurrences of a sub-expression x by the expression y—of course while respecting the usual restrictions on introducing a free variable in a context with a different binding for that variable. (Rule names to be used in hints are given in small capitals.)

Typing notation

All expressions, functions and other values are assumed to have some well-defined type (or possibly a polymorphic type). Even if some type is not explicitly given, it is assumed to be known.

Instead of the conventional $f\colon\ s \to t$ we write $f \in t \leftarrow s$ to denote the typing of a function. So we give the target type first, and only then the source type. A function typed this way is *total* on s, unless we explicitly state that it is partial.

We let $f.x$ denote the application of the function f to an argument x, which for $f \in t \leftarrow s$ implies that x has type s. (Function application has a high priority: $f.x + y$ means $(f.x)+y$.)

Whenever convenient, we identify a type with the set of values of that type.

Lambda forms

Instead of $\lambda x{\in}a \bullet x^2{+}1$ we write $(x : x{\in}a : x^2{+}1)$. In general, the form is $(\mathit{dummies} : \mathit{range} : \mathit{result})$, in which *dummies* is a list of variables, *range* is a propositional expression that depends on the *dummies*, while *result* is an expression that also may depend on the *dummies*. We continue to call these lambda-less expressions *lambda forms*. The source type of a lambda form is basically the set of values for which the *range* evaluates to true, while the target type is the type of *result*.

Often pure typing information in the range part is omitted, as in $(\mathit{dummies} :: \mathit{result})$. We do this with typing information that can be inferred from the context. For example, in the context of functions on the naturals, $(n :: 3{\times}n{+}1)$ is supposed to stand for $(n : n{\in}\mathbb{N} : 3{\times}n{+}1)$ and has typing $\mathbb{N}{\leftarrow}\mathbb{N}$.

The meaning of lambda forms follows from the following characterisation rule:

$$f = (x :: E) \quad \equiv \quad \forall(x :: f.x = E) \quad , \qquad \text{Lambda-char}$$

so if $f \in t \leftarrow s$, the ranges are understood to be as in $f = (x : x{\in}s : E) \quad \equiv \quad \forall(x : x{\in}s : f.x = E)$. (Generally, rules involve an implicit universal quantification over the free variables: for all f ... .)

Although the rule Lambda-char will not be used directly, we show now, also as an illustration of the general proof style, how it implies a number of (well-known) rules that *are* used. The first rule can be used to introduce or eliminate lambda forms:

$$f = (x :: f.x) \quad . \qquad \text{Lambda-abstraction}$$

It is derived as follows:

$$\begin{array}{ll} & f = (x :: f.x) \\ \equiv & \quad \{ \quad \text{Lambda-char} \quad \} \\ & \forall(x :: f.x = f.x) \\ \equiv & \quad \{ \quad \text{reflexivity of } = \quad \} \\ & \forall(x :: \mathsf{true}) \\ \equiv & \quad \{ \quad \text{rules for } \forall \quad \} \\ & \mathsf{true} \quad . \end{array}$$

The next rule is:

$$f = g \quad \equiv \quad \forall(x :: f.x = g.x) \quad . \qquad \text{Extensionality}$$

The proof is easy:

$$\begin{array}{ll} & f = g \\ \equiv & \quad \{ \quad \text{Lambda-abstraction applied to } g \quad \} \\ & f = (x :: g.x) \\ \equiv & \quad \{ \quad \text{Lambda-char} \quad \} \\ & \forall(x :: f.x = g.x) \quad . \end{array}$$

The special computation rule for lambda-application:

$$(x :: E).x = E \qquad \text{Lambda-comp}$$

follows immediately from Lambda-char. (Recall that the rule is understood to be universally quantified over the free variable x.) The general computation rule for lambda-application:

$$(x :: E).y = E[x := y] \qquad \text{LAMBDA-COMP}$$

(in which $E[x := y]$ denotes substitution of y for all free occurrences of x in E) can then be derived using elementary substitution rules:

$$\begin{array}{ll} & (x :: E).y = E[x := y] \\ \equiv & \quad \{ \quad \text{substitution rules} \quad \} \\ & ((x :: E).x = E)[x := y] \\ \equiv & \quad \{ \quad \text{(special) LAMBDA-COMP} \quad \} \\ & \mathsf{true}[x := y] \\ \equiv & \quad \{ \quad \text{substitution rules} \quad \} \\ & \mathsf{true} \ . \end{array}$$

This gives us the rule for dummy renaming (also known as alpha-conversion):

$$(x :: E) = (y :: E[x := y]) \ . \qquad \text{DUMMY-RENAMING}$$

The proof is left to the reader as an exercise.

From the definition of function composition:

$$f \circ g = (x :: f.(g.x)) \ , \qquad \text{FUN-}\circ\text{-DEF}$$

we obtain immediately by LAMBDA-CHAR the following computation rule:

$$(f \circ g).x = f.(g.x) \ . \qquad \text{FUN-}\circ\text{-COMP}$$

(The composition operation has a low priority: $f \circ g{+}h$ means $f \circ (g{+}h)$.)

Finally, we mention, without proof, three fusion rules for lambda forms:

$$f \circ (y :: U) = (y :: f.U) \ ; \qquad \text{LAMBDA-LEFT-FUSION}$$

$$(x :: E) \circ g = (y :: E[x := g.y]) \ . \qquad \text{LAMBDA-RIGHT-FUSION}$$

$$(x :: E) \circ (y :: U) = (y :: E[x := U]) \ . \qquad \text{LAMBDA-LAMBDA-FUSION}$$

Quantifier notation

A lambda form may be preceded by a quantifier. We have used this already above with $\forall$. For the quantifiers $\forall$ and $\exists$ the result expression has to be propositional, as in $\forall(n :: \exists(m :: m > n))$. The quantifier may also be an associative operator, with a neutral element if the range may be empty. For example, the meaning of $\cap(x : x \supseteq s : \mathit{convex}.x)$ is: the intersection of all convex supersets of s. With this convention we could also have used $\wedge$ and $\vee$ instead of $\forall$ and $\exists$.

Set comprehension

The notation $\$(\mathit{dummies} : \mathit{range} : \mathit{result})$, again with a propositional result expression, stands for a set comprehension. For example, $\$(x : x \supseteq s : \mathit{convex}.x)$ is the set of all convex supersets of s. Using the section notation introduced below we can characterise the meaning by:

$$s = \$(x :: E) \;\equiv\; (\in s) = (x :: E) \quad . \qquad \text{Compr-char}$$

From this rule, which demonstrates the well-known correspondence between sets and predicates, it is easy to derive:

$$s = \$(x :: x \in s) \quad . \qquad \text{Compr-abstraction}$$

Relations

As for functions, we assume that relations are typed. We shall usually consider a relation R between two sets s and t as a subset of the Cartesian product $s \times t$, but often step silently to the corresponding binary-predicate view and use an infix notation then, so $xRy \equiv (x, y) \in R$. The Compr-abstraction rule specialised to relations is then:

$$R = \$(x, y :: xRy) \quad . \qquad \text{Rel-abstraction}$$

Relational composition is defined by:

$$R \circ S = \$(x, z :: \exists (y :: xRy \wedge ySz)) \quad , \qquad \text{Rel-}\circ\text{-def}$$

from which we easily obtain:

$$x \; (R \circ S) \; z \equiv \exists (y :: xRy \wedge ySz) \quad . \qquad \text{Rel-}\circ\text{-comp}$$

Section notation

If $\oplus$ is a binary infix operation, the *presection* $x\oplus$ denotes the function $(y :: x \oplus y)$, while the *postsection* $\oplus y$ denotes the function $(x :: x \oplus y)$. Used by itself, often between parentheses to avoid confusion, $\oplus$ stands for the function $(x, y :: x \oplus y)$ — and not for the curry'd version $(x :: (y :: x \oplus y))$. So, rather than being a function of some type $(a \leftarrow c) \leftarrow b$, it has some type $a \leftarrow b \times c$.

Exercise 1.1 Suppose that instead of the Leibniz rule as given we only have the following 'safe' rule:

$$f.x = f.y \;\Leftarrow\; x = y \qquad \text{Fun-Leibniz}$$

('safe' in the sense that the rule can be applied blindly—no care is required for variable bindings). Show that the rule $E[z := x] = E[z := y] \;\Leftarrow\; x = y$, which is very close to the original Leibniz rule, is a consequence of the rule Fun-Leibniz.

□

Exercise 1.2 Prove the rule DUMMY RENAMING.

□

Exercise 1.3 Prove the three lambda-fusion rules.

□

Exercise 1.4 Prove the rule COMPR-ABSTRACTION.

□

Exercise 1.5 Prove the rule REL-∘-COMP.

□

Chapter 2

Basic Definitions

In this chapter we give an introduction to the basic concepts of category theory. The concept of a category corresponds in lattice theory to the concept of a *pre-ordered set*, i.e., a set with a binary relation $\sqsupseteq$ which is reflexive and transitive. (Here, $x \sqsupseteq y$ means 'x contains y', i.e. the relationship between the ordering $\sqsupseteq$ and the binary infimum operator $\sqcap$ is as follows: $x \sqcap y = y \ \equiv \ x \sqsupseteq y$.) In lattice theory, we write $x \sqsupseteq y$; in category theory we write $f \in x \leftarrow y$. Here f is considered to be a *witness* of the ordering relation between x and y. In this sense, category theory is constructive: it is a theory how to *construct* such witnesses rather than a theory about the *existence* of the witnesses. So, reflexivity corresponds in category theory to the existence for all objects of an identity arrow and transitivity corresponds to the existence of a composition operator.

2.1 Categories

We start by giving the definition of a category.

Definition 2.1 (Category) A *category* comprises:
(a) A class of *objects*. We use the notation $x \in \mathcal{C}$ to denote that x is an object of the category $\mathcal{C}$.
(b) A class of *arrows*.
(c) Two mappings to the class of objects from the class of arrows, called *codomain* and *domain*, denoted by cod and dom, respectively. We say that an arrow f is *to* x and *from* y if $\mathsf{cod}.f = x$ and $\mathsf{dom}.f = y$. With $x \leftarrow y$ we denote the collection of all arrows to x from y, and so we also often write $f \in x \leftarrow y$.
(d) A *composition operator* $\circ$ which is a partial function defined for all pairs of arrows f and g with $\mathsf{dom}.f = \mathsf{cod}.g$ and assigning to such an f and g an arrow $f \circ g \in \mathsf{cod}.f \leftarrow \mathsf{dom}.g$, satisfying: for all arrows $f \in w \leftarrow x$, $g \in x \leftarrow y$ and $h \in y \leftarrow z$:

$$f \circ (g \circ h) = (f \circ g) \circ h \quad .$$

(This associativity law allows us to drop the brackets and write '$f \circ g \circ h$'.)

(e) For each object x an *identity arrow* $\mathsf{id}_x \in x \leftarrow x$ such that: for all arrows $f \in x \leftarrow y$,

$$\mathsf{id}_x \circ f = f = f \circ \mathsf{id}_y \quad .$$

□

In the literature one also finds the term *morphism* instead of *arrow*.

Arrow typing

Instead of saying $f \in x \leftarrow y$, we also say: f has the *typing* $x \leftarrow y$. To prevent confusion, we sometimes mention explicitly in the typing to which category an arrow belongs. For example, to make explicit that f is a $\mathcal{C}$-arrow, we write $f \in x \xleftarrow{\mathcal{C}} y$.

The requirement '$\mathsf{dom}.f = \mathsf{cod}.g$' in (c) above can also be expressed in the *typing rule*

$$f \circ g \in x \leftarrow z \Leftarrow f \in x \leftarrow y \wedge g \in y \leftarrow z \quad . \qquad \circ\text{-TYPING}$$

In an expression '$f \circ g$' the components are called *suitably typed* if their typing satisfies $\mathsf{dom}.f = \mathsf{cod}.g$. This is a requirement for the definedness of the expression. We call an expression *suitably typed* if its components are suitably typed. Applied to more complicated expressions, such as '$f \circ g \circ h$', it applies to all component compositions; for this case $\mathsf{dom}.f = \mathsf{cod}.g \wedge \mathsf{dom}.g = \mathsf{cod}.h$.

If two arrows are equal they have the same codomain and domain. For example, the arrow equality $\mathsf{id}_x = \mathsf{id}_y$ implies the equality of the objects x and y. An arrow equality '$f = g$' is called *suitably typed* if, first, both f and g are suitably typed, and, moreover, such that $\mathsf{cod}.f = \mathsf{cod}.g \wedge \mathsf{dom}.f = \mathsf{dom}.g$. Applied to '$f \circ g = h \circ k$', for example, suitable typing means: there exist w, x, y and z such that $f \in w \leftarrow x$, $g \in x \leftarrow z$, $h \in w \leftarrow y$ and $k \in y \leftarrow z$. Throughout the following there will always be the assumption, whether stated explicitly or not, that all arrow compositions and equalities are suitably typed.

We write '$f = g \in x \leftarrow y$' as a shorthand for: $f \in x \leftarrow y \wedge g \in x \leftarrow y \wedge f = g$.

Transformation

If we have a class of arrows α_x, one for each object x of some category $\mathcal{C}$—as for the identity arrows—we call it a *transformation* on $\mathcal{C}$. Sometimes it is useful to think of a transformation as a mapping from objects of some category to arrows of some (possibly other!) category, and to make that point of view explicit we use the notation $[\alpha] = (x :: \alpha_x)$. So, applying this to id, we have $[\mathsf{id}].x = \mathsf{id}_x$.

Subcategory

A category $\mathcal{D}$ is called a *subcategory* of a category $\mathcal{C}$ if: (a) the objects of $\mathcal{D}$ form a subclass of the objects of $\mathcal{C}$, (b) the arrows of $\mathcal{D}$ form a subclass of the arrows of $\mathcal{C}$, (c) the composition of arrows in $\mathcal{D}$ is the same as that in $\mathcal{C}$, and (d) the identity arrows of $\mathcal{D}$ form a subclass of the identity arrows of $\mathcal{C}$.

If some object $x \in \mathcal{C}$ is not an object of a subcategory $\mathcal{D}$, then clearly all arrows of $\mathcal{C}$ that have x as their codomain or domain are not arrows of $\mathcal{D}$. If that is the only restriction, we have a 'full' subcategory; more precisely, a category $\mathcal{D}$ is called a *full subcategory* of a category $\mathcal{C}$ if: $\mathcal{D}$ is a subcategory of $\mathcal{C}$ and, moreover, for all $x, y \in \mathcal{D}$, $x \overset{\mathcal{D}}{\leftarrow} y$ is the same as $x \overset{\mathcal{C}}{\leftarrow} y$.

Precategory

Sometimes a structure is 'almost' a category: it satisfies all requirements of the definition, except for the uniqueness of (co)domains. Such an almost-category is called a *precategory*. For example, there might be a 'polymorphic' identity $\mathsf{id} \in x \leftarrow x$ for all x, and in general arrows may admit several typings. The following *triple trick* is a standard construction that will turn any precategory into a category: Replace each arrow f that admits the typing $f \in x \leftarrow y$ by a triple $(x,f,y) \in x \leftarrow y$. So, for example, a polymorphic identity is turned into as many arrows as there are objects. This construction is usually performed 'virtually': we continue to write f instead of the more proper (x,f,y), so as to keep the notation light. Extra care must be taken then, though, to ensure suitable typing in compositions. Further, in a category the following is valid:

$$u = x \wedge v = y \;\Leftarrow\; f \in u \leftarrow v \wedge f \in x \leftarrow y \quad ,$$

since $f \in u \leftarrow v$ means $\mathsf{cod}.f = u \wedge \mathsf{dom}.f = v$, but if the triple trick has been applied virtually, this rule is nonsensical: the two occurrences of f are short-hand for different arrows (u,f,v) and (x,f,y).

Derived categories; base category

It is often the case in category theory that one category is defined in terms of another. More precisely, there are general constructions to create a new category from a given category. We call such a newly created category a *derived category* and the category on which it is based the *base category*. For example, given a category $\mathcal{C}$, we can define the *objects* of a category $\mathcal{D}$ to be some subclass of the *arrows* of $\mathcal{C}$. So, in the derived category $\mathcal{D}$ we have arrows to arrows of $\mathcal{C}$ from arrows of $\mathcal{C}$.To prevent confusion, we sometimes mention explicitly in the typing to which category an arrow belongs. For example, if f and g are arrows in the base category $\mathcal{C}$, then we denote the arrow φ in $\mathcal{D}$ to f from g by

$$\varphi \in f \overset{\mathcal{D}}{\longleftarrow} g \quad .$$

Witnesses

From the definition of a category it should be obvious that one way of looking at it is as a directed graph with some additional structure, in which the objects are the vertices (nodes) and the arrows the arcs. It often makes sense to interpret $f \in x \leftarrow y$ as the statement: 'f is a way to reach the destination x from the starting point y'. We can

also give a propositional interpretation to $x \leftarrow y$, namely: 'x is reachable from y'. This is true if the arrow collection $x \leftarrow y$ contains at least one arrow; otherwise it is false. This interpretation gives rise to the term 'witness': we call an arrow f a *witness* of $x \leftarrow y$ (the reachability of x from y) whenever $f \in x \leftarrow y$.

Coherence conditions

In the definition of a category the requirement

$$f \circ (g \circ h) = (f \circ g) \circ h$$

is a so-called *coherence condition*. In general, a rule is called a coherence condition if it has the form of one or more equalities between arrow-valued expressions that state that the different ways in which the basic rules of category theory can be applied to establish the reachability of some object from some other object—given a bunch of ingredients that are as generally typed as allowed by suitability—has no effect on the witnesses obtained.

For example, consider the typing of the identity arrows. Given an $f \in x \leftarrow y$ we can show the reachability of x from y in (at least) three different ways: immediately by saying we have an arrow $f \in x \leftarrow y$; by arguing that we have (i) an arrow $\mathsf{id}_x \in x \leftarrow x$, and (ii) the arrow $f \in x \leftarrow y$, so we can use composition to construct $\mathsf{id}_x \circ f \in x \leftarrow y$; or by arguing that we have (i) the arrow $f \in x \leftarrow y$, and (ii) an arrow $\mathsf{id}_y \in y \leftarrow y$, so we can use composition to construct $f \circ \mathsf{id}_y \in x \leftarrow y$. The identity axiom: the requirement that $\mathsf{id}_x \circ f = f = f \circ \mathsf{id}_y$ whenever $f \in x \leftarrow y$, states that these three witnesses are the same. So this axiom is a coherence condition.

Likewise, the associativity axiom expresses that the two ways of constructing an arrow with typing $w \leftarrow z$, given arrows with typings $w \leftarrow x$, $x \leftarrow y$ and $y \leftarrow z$, are equivalent.

Note. Coherence conditions should *never* be automatically assumed, but are *in each case* rules that are *explicitly* given as part of a definition. For example, given $f \in x \leftarrow x$ we have two ways of reaching x from x: by using f, and by using id_x. But it would be an error to conclude from this that $f = \mathsf{id}_x$.[1]

Monomorphic categories

In general there is no requirement whatsoever that the arrows between two given objects are unique. A category in which they are, so each class $x \leftarrow y$ contains at most one arrow, is called *monomorphic*. In a monomorphic category all conceivable coherence conditions are automatically fulfilled: it is generally valid to conclude $f = g$ from $f \in x \leftarrow y$ and $g \in x \leftarrow y$.

[1] For a more advanced counterexample, jumping wildly ahead: in monads, which will defined in the last chapter, $\mu_M = M\mu \in MM \dot{\leftarrow} MMM$ has the right form to be a coherence condition, but this arrow equality between the two natural transformations does in general not hold

Small categories

In the definition of category we used the term 'class' for the collections of objects and of arrows, and not the term 'set'. The reason is that in the general case these collections may be too 'large' to be sets in the sense of standard axiomatised set theory. Set theory may be modelled within category theory and *vice versa*, and with the assumption that all classes are sets it would be possible to obtain inconsistencies like Russell's paradox.

A category is said to be *small* if the class of objects and the class of arrows are both sets. A category is said to be *locally small* if for all objects x and y the collection of all arrows to x from y, i.e. $x \leftarrow y$, is a set. If a category is locally small $x \leftarrow y$ is called a *hom-set*.

Monomorphic categories are locally small.

Isomorphic objects

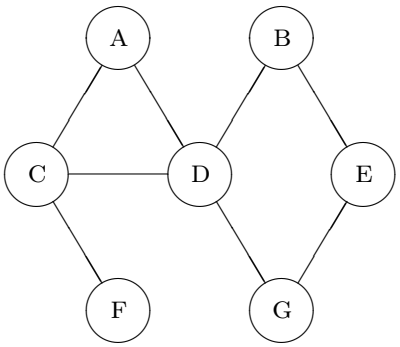
Definition 2.2 (Isomorphism) Objects x and y in the same category are said to be *isomorphic* if: there exist two arrows $f \in x \leftarrow y$ and $g \in y \leftarrow x$ such that $f \circ g = \mathsf{id}_x$ and $g \circ f = \mathsf{id}_y$ (in which case we also say that f and g are each others' inverse). If this is the case we write $x \cong y$.

The pair (f, g) is called an *isomorphism* between x and y, or a *witness* of the isomorphy $x \cong y$. Sometimes we omit the g component—which is uniquely determined by f—and say that f is an isomorphism (between x and y).

□

If $\mathcal{C}$ is some category with isomorphic objects x and y, with witness (f, g), we obtain an automorphism on $\mathcal{C}$ as follows: (a) Map x to y, y to x, and each other object $z \in \mathcal{C}$ to itself. (b) Map each arrow $h \in u \leftarrow v$ to $\delta_u \circ h \circ \varepsilon_v$, where $\delta_x = g$, $\delta_y = f$ and $\delta_z = \mathsf{id}_z$ for each other object $z \in \mathcal{C}$, while $\varepsilon_x = f$, $\varepsilon_y = g$ and $\varepsilon_z = \mathsf{id}_z$ for each other object $z \in \mathcal{C}$.

Note that this mapping respects the typing of arrows and preserves compositions and identities: if we denote it (both for objects and for arrows) by F, then $h \in u \leftarrow v$ is mapped to $Fh \in Fu \leftarrow Fv$, and for all suitably typed $\mathcal{C}$-arrows h and k and all $z \in \mathcal{C}$ we have: $F(h \circ k) = Fh \circ Fk$ and $F\mathsf{id}_z = \mathsf{id}_{Fz}$. It is also an involution: $F \circ F$ is the identity mapping. Any statement that can be made about $\mathcal{C}$ is invariant under this mapping. In other words, there is a symmetry between isomorphic objects; they are interchangeable in the sense that they cannot be distinguished by their 'categorical' properties, just as in the (undirected) graph below there is a symmetry between the vertices B and G: they are indistinguishable by their graph-theoretical properties. (In contrast, A, for example, can be distinguished from B by the property: 'has a neighbour of degree 3'.)



For another example, define in the complex numbers $j = -i$. Any property of i is true of j as well, and *vice versa*. For example, $|\ 3+4i\ | = 5$, so $|\ 3+4j\ | = 5$. Likewise, $e^{i\pi} = -1$, so $e^{j\pi} = -1$. (If we define $f.x = ix$, then $f.i = -1$ while $f.j = 1$. This is not a counterexample; the issue is that the replacement of i by j has not been systematic enough: there is a hidden dependency on i in the definition of f. It is made clear in: $(x :: ix).i = -1$; and indeed, $(x :: jx).j = -1$.) One way of looking at this is that i is—by definition—an 'implementation' of the specification $(x :: x^2+1=0)$. But for any i that satisfies that specification, $-i$ is an equally valid implementation.

Uniqueness up to isomorphism; 'the'

In category theory it is usual to define notions by giving the categorical property they have to satisfy. In general there may be many solutions, but if all solutions are necessarily isomorphic this is, nevertheless, considered a proper definitional mechanism. The reason is that different solutions, although different, are categorically indistinguishable, so any solution is precisely as good as any other solution. Take, for concreteness, the notion '*initial object*', which will be defined later. A category may have many initial objects, but if so they are all isomorphic. This situation is usually described by saying: initial objects are *unique up to isomorphism*. Further, since all are equally good, we may arbitrarily pick one (assuming there is at least one solution) and call it '*the*' initial object. When we do this, we need not mention the arbitrarily picking a solution any more, but instead may say something like: 'Let 0 denote 'the' initial object.'.

Note. The converse statement that two categorically indistinguishable objects are isomorphic is, in general, false.

Exercise 2.3 Show that $\cong$ is an equivalence relation.

□

Exercise 2.4 Show that if both $(f, g0)$ and $(f, g1)$ are isomorphisms, $g0 = g1$.

□

2.2 Examples of Concrete Categories

Definition 2.5 (Pre-ordered set) A *pre-ordered set* is: a structure $(A, \sqsupseteq)$, in which A is a set and $\sqsupseteq$ is a relation on A that is reflexive and transitive.

□

Definition 2.6 (PreOset.$\mathcal{A}$) Given a pre-ordered set $\mathcal{A} = (A, \sqsupseteq)$, the category PreOset.$\mathcal{A}$ is defined as follows:

Objects: The elements of A.

Arrows: The elements of $\sqsupseteq$ viewed as a subset of $A \times A$, i.e., (x, y) is an arrow if: $x \sqsupseteq y$, and it then has the typing $(x, y) \in x \leftarrow y$.

Composition: $(x, y) \circ (y, z) = (x, z)$.

Identities: All pairs (x, x) for $x \in A$.

□

Note that the definitions of composition and identities were forced. The existence of the identity arrows follows from the reflexivity of $\sqsupseteq$, and that the set of arrows is closed under composition follows from the transitivity of $\sqsupseteq$. (In fact, the statement that this construction applied to $(A, \sqsupseteq)$ gives a category is *equivalent* to the statement that $(A, \sqsupseteq)$ is a pre-ordered set.) Composition is associative because associativity is a coherence condition, and this category is monomorphic.

Definition 2.7 (Partially ordered set) A *partially ordered set* is: a structure $(A, \sqsupseteq)$, in which A is a set and $\sqsupseteq$ is a relation on A that is reflexive, transitive and anti-symmetric—that is, $x \sqsupseteq y \wedge y \sqsupseteq x \ \equiv \ x = y$.

□

Definition 2.8 (POset.$\mathcal{A}$) Given a partially ordered set $\mathcal{A}$, the category POset.$\mathcal{A}$ is: PreOset.$\mathcal{A}$.

□

The introduction of a different name, POSet rather than PreOSet, is strictly speaking superfluous, but serves to make the restriction of anti-symmetry on the order relation of $\mathcal{A}$ explicit.

Definition 2.9 (Fun) The category Fun is defined as follows:

Objects: Types.

Arrows: Typed total functions, where the codomain of a function is its target type, and the domain its source type.

Composition: Function composition.

Identities: The identity mappings, one for each type.

□

Remark 2.10 It is conventional to name this category Set, after its objects (recall that we identify types and sets), but it makes more sense to name it after its arrows, as it has the same objects as Par and Rel defined below.

□

Remark 2.11 Depending on the details of the typing discipline, we may only obtain a precategory. For example, if there is a 'coercion' (implicit type conversion) to reals from integers, we may as well type $\mathsf{floor} \in \mathbb{Z} \leftarrow \mathbb{R}$ as $\mathsf{floor} \in \mathbb{R} \leftarrow \mathbb{R}$. So the triple trick is supposed to have been used here. For example, to be precise, one might name the first version $\mathsf{floor}_{\mathbb{Z}}$ and the second $\mathsf{floor}_{\mathbb{R}}$.

□

Remark 2.12 A fine point is whether there is only one empty set, or many: the empty set of integers, the empty set of lists of integers, and so on. The reader may freely adopt either point of view. Categorically it hardly makes a difference, since all empty sets are isomorphic.

□

Remark 2.13 We shall sometimes give examples using a functional programming notation, in a dialect of Haskell or Gofer[2]. However, for this to be a model of Fun we must take care that all functions are total, and consider a polymorphic function definition as an abbreviation for a whole collection of monomorphic function definitions. Unfortunately, these languages have a typing discipline that is still too weak to express many interesting concepts properly, such as the polytypic map functor introduced in the chapter on Algebras.

[2]We do not adhere to the requirement that constructor functions begin with a capital letter.

□

Remark 2.14 Fun is not a small category, since there is no 'set of sets'.

□

Definition 2.15 (Par) The category Par is just like Fun, except that we also allow *partial* functions.

□

Definition 2.16 (Rel) The category Rel is defined as follows:

Objects: Sets.

Arrows: Relations, where $R \in x \leftarrow y$ if: R is a relation between x and y.

Composition: Relation composition.

Identities: The equality relations, one for each set.

□

(This is, again, actually only a precategory.)

Definition 2.17 (Monoid) A *monoid* is: a structure $(A,\oplus,e)$, in which A is a set, $\oplus \in A \leftarrow A \times A$ and $e \in A$, satisfying the *monoid laws*: $\oplus$ is an associative operation, and e is a neutral (or identity) element for $\oplus$.

□

(Note that a group is a monoid equipped with yet another operation, inverse, giving rise to additional laws.)

Definition 2.18 (Mon.$\mathcal{M}$) Given a monoid $\mathcal{M} = (A,\oplus,e)$, the category Mon.$\mathcal{M}$ is defined as follows:

Objects: Some (further irrelevant) set with one element.

Arrows: A.

Composition: $\oplus$.

Identities: $\{e\}$.

□

So a monoid corresponds to a one-object category; conversely, each one-object category corresponds to a monoid.

Definition 2.19 (Discr.S) Given a set S, the *discrete category* Discr.S is defined as follows:

Objects: S.

Arrows: S, where for each $x \in S : x \in x \leftarrow x$.

Composition: Forced by the arrows and the identity law: $x \circ x = x$.

Identities: Forced by the arrows: $\mathsf{id}_x = x$.

□

This is the categorical counterpart of a discrete graph. Note that composition and identities are forced by the definition of the arrows. Through the lack of structure this kind of category is not very exciting, but it can be a helpful tool in some categorical construction work, as well in providing counterexamples.

Definition 2.20 (Paths.$\mathcal{G}$) Given a directed graph $\mathcal{G} = (V, E)$, in which V is a set of vertices and E a set of arcs (each having a target vertex and a source vertex, both from V), the category Paths.$\mathcal{G}$ is defined as follows:

Objects: V.

Arrows: The set $x \leftarrow y$ consists of the paths on $\mathcal{G}$ ending in x and starting from y, where a path is a finite (possibly empty) sequence of arcs, such that the source of each is the target of the next.

Composition: Concatenation.

Identities: The empty paths.

□

To make this completely correct, the triple trick for turning a precategory into a category must be applied, since we need a separate identity arrow for each object, while it might be argued that there is only one empty sequence and therefore only one empty path.

Definition 2.21 (Nat) The category Nat is defined as follows:

Objects: $\mathbb{N}$.

Arrows: The set $m \overset{\mathsf{Nat}}{\leftarrow} n$ consists of $\$(k :: k < m) \overset{\mathsf{Fun}}{\leftarrow} \$(k :: k < n)$.

Composition: As in Fun.

Identities: The identity arrows of Fun.

□

Remark 2.22 But for the notation used for the objects, Nat is the full subcategory of Fun whose objects are $\$(k :: k < n)$ for $n \in \mathbb{N}$.

□

Exercise 2.23 Which category-theoretic statement about $\mathsf{PreOset}.\mathcal{A}$ corresponds to the statement that $\mathcal{A}$ is a partially ordered set?

□

Exercise 2.24 When are two objects of Fun isomorphic? Answer the same question for the other examples of concrete categories.

□

Exercise 2.25 Compare $\mathsf{Discr}.S$ with $\mathsf{POset}.(S, =)$ and $\mathsf{Paths}.(S, \emptyset)$.

□

Exercise 2.26 How many arrows are there in the set $m \overset{\mathsf{Nat}}{\leftarrow} n$?

□

2.3 Examples of Derived Categories

Definition 2.27 (Opposite category $\mathcal{C}^{op}$) Given a category $\mathcal{C}$, the so-called *opposite* category $\mathcal{C}^{op}$ is: the category obtained by switching the roles of domain and codomain, so that $\mathcal{C}^{op}$ is defined by:

Objects: The objects of $\mathcal{C}$.

Arrows: The arrows of $\mathcal{C}$, where $f \in x \overset{\mathcal{C}^{op}}{\longleftarrow} y \equiv f \in y \overset{\mathcal{C}}{\longleftarrow} x$.

Composition: To avoid confusion, we denote composition in $\mathcal{C}^{op}$ by the symbol ';', where, by definition, $f;g = g \circ f$, in which the symbol '$\circ$' is the composition of $\mathcal{C}$.

Identities: Those of $\mathcal{C}$.

□

It follows from this definition that $\_^{op}$ is a convolution, that is, $(\mathcal{C}^{op})^{op} = \mathcal{C}$.

Dual; co-

From a definition involving a category we can make another definition for that same category, called its *dual*, by applying the original definition to the opposite category. To give an imaginary simple example, suppose we define an object x of a category $\mathcal{C}$ to be a 'black hole' if the only arrow with *domain* x is id_x. Then an object x satisfies the dual property if the only arrow with *codomain* x is id_x . So we can simply define some categorical notions by saying that they are the dual of some earlier defined notion.

If a notion is dualised we can invent some new term if we wish, as in 'white hole', but we can also use the common convention of reusing the term for the original notion preceded by '*co*', as in 'coblack hole'. So the 'coblack holes' of $\mathcal{C}$ are the 'black holes' of $\mathcal{C}^{op}$. Similarly, the dual of 'domain' is 'codomain', and indeed, $\mathsf{cod}.f$ in $\mathcal{C}$ is the same object as $\mathsf{dom}.f$ in $\mathcal{C}^{op}$.

We shall sometimes directly define some co*thing* without first defining the notion *thing*; the *thing* is then its dual. Which is the original and which the dual notion (why not call a 'black hole' a 'cowhite hole'?) is a matter of tradition, but there is also some system to it that would lead too far to explain it already here.

Definition 2.28 (Product category) Given two categories $\mathcal{C}$ and $\mathcal{D}$, the *product category* $\mathcal{C}\times\mathcal{D}$ is defined by:

Objects: The class of all pairs (v, x) with $v \in \mathcal{C}$ and $x \in \mathcal{D}$.

Arrows: The class of arrows $(v, x)^{\mathcal{C}\times\mathcal{D}}\!\leftarrow (w, y)$ consists of all pairs (f, g) such that $f \in v \xleftarrow{\mathcal{C}} w$ and $g \in x \xleftarrow{\mathcal{D}} y$.

Composition: Componentwise: $(f, g)\circ(h, k) = (f\circ h, g\circ k)$, in which the composition of '$f\circ h$' is that of $\mathcal{C}$ and the composition of '$g\circ k$' is that of $\mathcal{D}$.

Identities: The identity arrow $\mathsf{id}_{(v,x)} = (\mathsf{id}_v, \mathsf{id}_x)$.

□

Exercise 2.29 What would be the co-opposite of a category?

□

Exercise 2.30 What is Rel^{op}? And what is $(\mathsf{PreOset}.\mathcal{A})^{op}$?

□

Exercise 2.31 Show that $\_^{op}$ distributes through $\times$, i.e., $(\mathcal{C}\times\mathcal{D})^{op} = \mathcal{C}^{op}\times\mathcal{D}^{op}$

□

Exercise 2.32 What is $\mathsf{Discr}.S \times \mathsf{Discr}.T$?

□

2.4 Initial and Terminal Objects

2.4.1 Initial Objects

Definition 2.33 (Initial object) An *initial object* of a category $\mathcal{C}$ is: an object a of $\mathcal{C}$ such that, for each object x in the category, there is a *unique* arrow to x from a—that is, there is precisely one arrow in $x{\leftarrow}a$.

□

We use $(\!|x \leftarrow a|\!)$ to denote the unique arrow to an arbitrary object x of $\mathcal{C}$ from an initial object a, which is characterised by:

$$(\!|x \leftarrow a|\!) = f \equiv f \in x \leftarrow a \quad . \qquad (\!|\_|\!)\text{-CHAR}$$

(Note that the $\Rightarrow$ direction is the requirement of suitable typing, while the $\Leftarrow$ direction expresses uniqueness and is a coherence condition.)

Definition 2.34 (Catamorphism; initiality) Given an initial object a, we call these unique arrows $(\!|x \leftarrow a|\!)$ *catamorphisms*, and the pair $(a, (\!|\_ \leftarrow a|\!))$—in which the second component denotes the mapping $(x :: (\!|x \leftarrow a|\!))$—an *initiality*.

□

Example 2.35 Recall that there is an arrow with typing $x \leftarrow y$ in $\mathsf{POset}.\mathcal{A}$, where $\mathcal{A} = (A, \sqsupseteq)$, if $x \sqsupseteq y$. For an initial object a of $\mathsf{POset}.\mathcal{A}$ there is an arrow with typing $x \leftarrow a$ for all $x \in A$, which means: $x \sqsupseteq a$ for all $a \in A$. A partially ordered set $\mathcal{A}$ can have at most one element with that property, which is then called the *bottom* of $\mathcal{A}$ and often denoted by $\bot$. If $\mathcal{A}$ has a bottom, it is an initial object of $\mathsf{POset}.\mathcal{A}$, since there is an arrow to each object, which by the monomorphicity of this category is unique.

□

Example 2.36 We show that Fun has an initial object. First we show that it is empty. Next we verify that any empty set is indeed initial.

Suppose that the set s is an initial object of Fun. Below $(x : x \in s : 0)$ and $(x : x \in s : 1)$ stand for two functions with the typing $\{0,1\} \leftarrow s$. Then:

$$
\begin{array}{ll}
 & \neg\exists(x :: x \in s) \\
\equiv & \quad \{ \quad \text{Quantification rules} \quad \} \\
 & \forall(x : x \in s : \mathsf{false}) \\
\equiv & \quad \{ \quad 0 \neq 1 \quad \} \\
 & \forall(x : x \in s : 0 = 1) \\
\equiv & \quad \{ \quad \textsc{Extensionality} \quad \} \\
 & (x : x \in s : 0) = (x : x \in s : 1) \\
\Leftarrow & \quad \{ \quad \text{symmetry and transitivity of } = \quad \} \\
 & (\!|\{0,1\} \leftarrow s|\!) = (x : x \in s : 0) \wedge (\!|\{0,1\} \leftarrow s|\!) = (x : x \in s : 1) \\
\Leftarrow & \quad \{ \quad (\!|\text{-}|\!)\text{-CHAR} \quad \} \\
 & (x : x \in s : 0) \in \{0,1\} \leftarrow s \wedge (x : x \in s : 1) \in \{0,1\} \leftarrow s \quad ,
\end{array}
$$

where the latter typings are those assumed.

Next,

$$
\begin{array}{ll}
 & f = g \in t \leftarrow s \\
\equiv & \quad \{ \quad \textsc{Extensionality} \quad \} \\
 & \forall(x : x \in s : f.x = g.x) \\
\Leftarrow & \quad \{ \quad f.x = g.x \Leftarrow \mathsf{false}\text{; quantification rules} \quad \} \\
 & \forall(x : x \in s : \mathsf{false}) \\
\equiv & \quad \{ \quad \text{quantification rules} \quad \} \\
 & \neg\exists(x :: x \in s) \quad ,
\end{array}
$$

which shows that arrows to any t from an empty s are unique.

□

Example 2.37 A directed graph is called a *rooted tree* if: there is a vertex, called the *root* of the tree, such that there is a unique path to any vertex of the graph from the root. This corresponds directly to the definition of initial object in $\mathsf{Paths}.\mathcal{G}$.

□

In the remainder of this section we present some elementary properties and theorems concerning catamorphisms.

It is easy to show that initial objects are unique up to isomorphism. Assume that we are working in a category $\mathcal{C}$ which has an initial object, and let 0 stand for 'the' initial object. Fixing the initial object has the advantage that we can drop the parameter a in the notation $(\!|x \leftarrow a|\!)$ and simply write $(\!|x|\!)$. So we get for the characterisation rule:

$$(\!|x|\!) = f \equiv f \in x \leftarrow 0 \quad . \qquad (\!|\text{-}|\!)\text{-CHAR}$$

By instantiating $f := (\!|x|\!)$ we obtain:

$$(\!|x|\!) \in x \leftarrow 0 \quad . \qquad (\!|\text{-}|\!)\text{-TYPING}$$

By instantiating f,$x := \mathsf{id}_0$,0 we obtain:

$$(\!|0|\!) = \mathsf{id}_0 \quad . \qquad (\!|\text{-}|\!)\text{-ID}$$

Theorem 2.38

$$f \circ (\!|y|\!) = (\!|x|\!) \;\Leftarrow\; f \in x \leftarrow y \quad . \qquad (\!|\text{-}|\!)\text{-FUSION}$$

Proof

$$\begin{array}{ll} & f \circ (\!|y|\!) = (\!|x|\!) \\ \equiv & \quad \{ \quad (\!|\text{-}|\!)\text{-CHAR} \quad \} \\ & f \circ (\!|y|\!) \in x \leftarrow 0 \\ \Leftarrow & \quad \{ \quad (\!|\text{-}|\!)\text{-TYPING, } \circ\text{-TYPING} \quad \} \\ & f \in x \leftarrow y \quad . \end{array}$$

□

Note that the condition $f \in x \leftarrow y$ is in fact nothing but the requirement that $f \circ (\!|y|\!) = (\!|x|\!)$ is suitably typed. Note, further, that the last two rules have the form of a coherence condition.

Exercise 2.39 Rewrite the $(\!|\_|\!)$ rules given above for an arbitrary, not fixed initial object, using the notation $(\!|\_ \leftarrow \_|\!)$. Then prove, *by using these rules*, that initial objects in the same category are unique up to isomorphism.

□

Exercise 2.40 Which of the further categories given as examples have initial objects, and what are they?.

□

2.4.2 Terminal Objects

Definition 2.41 (Terminal object) An *terminal object* of a category $\mathcal{C}$ is: an initial object of $\mathcal{C}^{op}$.

□

It is customary to use 1 for 'the' terminal object. We call the unique arrows to the terminal object *anamorphisms*, and use the notation $[\![\_]\!]$. By dualisation of $(\!|\_|\!)$-CHAR we obtain:

$$[\![x]\!] = f \equiv f \in 1 \leftarrow x \quad . \qquad [\![\_]\!]\text{-CHAR}$$

Example 2.42 We show that Fun has a terminal object. First we show that it is a one-element set. Next we verify that any one-element set is indeed terminal.

Suppose that the set 1 is a terminal object of Fun. Let z stand for the set $\{0\}$. Because $[\![z]\!] \in 1 \leftarrow z$ and $0 \in z$, $[\![z]\!].0 \in 1$, 1 has at least one element. We show now that $a = b \;\Leftarrow\; a \in 1 \wedge b \in 1$, so that 1 has at most one element. So assume that $a \in 1$ and $b \in 1$. Define $f \in 1 \leftarrow z$ by $f.0 = a$, and $g \in 1 \leftarrow z$ by $g.0 = b$. Then:

$$\begin{array}{ll} & a = b \\ \equiv & \quad \{ \quad \text{Definition of } f \text{ and } g \quad \} \\ & f.0 = g.0 \\ \Leftarrow & \quad \{ \quad \textsc{Leibniz} \quad \} \\ & f = g \\ \Leftarrow & \quad \{ \quad \text{symmetry and transitivity of } = \quad \} \\ & [\![z]\!] = f \wedge [\![z]\!] = g \\ \Leftarrow & \quad \{ \quad [\![\_]\!]\text{-CHAR} \quad \} \\ & f \in 1 \leftarrow z \wedge g \in 1 \leftarrow z \quad , \end{array}$$

where the latter typing is the one assumed.

Next, assume that 1 is some one-element set, say $\{e\}$. Then:

$$
\begin{array}{ll}
 & f = g \in 1 \leftarrow s \\
\equiv & \quad \{ \quad \text{EXTENSIONALITY} \quad \} \\
 & \forall(x : x{\in}s : f.x = g.x) \\
\Leftarrow & \quad \{ \quad \text{symmetry and transitivity of } = \quad \} \\
 & \forall(x : x{\in}s : f.x = e \ \wedge \ g.x = e) \\
\equiv & \quad \{ \quad \text{quantification rules} \quad \} \\
 & \forall(x : x{\in}s : f.x = e) \wedge \forall(x : x{\in}s : g.x = e) \\
\equiv & \quad \{ \quad \text{membership of one-element set} \quad \} \\
 & \forall(x : x{\in}s : f.x \in \{e\}) \wedge \forall(x : x{\in}s : g.x \in \{e\}) \\
\equiv & \quad \{ \quad 1 = \{e\}; \text{LEIBNIZ} \quad \} \\
 & \forall(x : x{\in}s : f.x \in 1) \wedge \forall(x : x{\in}s : g.x \in 1) \\
\Leftarrow & \quad \{ \quad \text{typing in } \mathsf{Fun} \quad \} \\
 & f \in 1 \leftarrow s \wedge g \in 1 \leftarrow s \quad ,
\end{array}
$$

which shows that arrows to a one-element set from any s are unique.

□

Exercise 2.43 Dualise the other $(\!|\_|\!)$ rules.

□

Exercise 2.44 Which of the further categories given as examples have terminal objects, and what are they?.

□

2.5 Functors

Definition 2.45 (Functor) Given two categories $\mathcal{C}$ and $\mathcal{D}$, a *functor* to $\mathcal{C}$ from $\mathcal{D}$ is: a pair of mappings $F = (F_{obj}, F_{arr})$ such that the object mapping F_{obj} maps objects of $\mathcal{D}$ to objects of $\mathcal{C}$ and the arrow mapping F_{arr} arrows of $\mathcal{D}$ to arrows of $\mathcal{C}$, satisfying, first, the following typing requirement: for all arrows f in $\mathcal{D}$:

$$F_{arr}.f \in F_{obj}.x \xleftarrow{\mathcal{C}} F_{obj}.y \Leftarrow f \in x \xleftarrow{\mathcal{D}} y \quad ;$$

and, second, the following two coherence conditions: for all $\mathcal{D}$-arrows f and g with $\mathsf{dom}.f = \mathsf{cod}.g$:

$$F_{arr}.(f \circ g) \;=\; F_{arr}.f \circ F_{arr}.g \quad ;$$

and for each object x in $\mathcal{D}$:

$$F_{arr}.\mathsf{id}_x = \mathsf{id}_{F_{obj}.x} \quad .$$

□

It is convention not to write subscripts $\_{}_{obj}$ and $\_{}_{arr}$ but to use the same symbol for denoting the object and the arrow mapping of a functor. Moreover, we will use *juxtaposition* to denote functor application. With that convention the above functor requirements become:

$$Ff \in Fx \xleftarrow{\mathcal{C}} Fy \Leftarrow f \in x \xleftarrow{\mathcal{D}} y \quad ;$$

$$F(f \circ g) \;=\; Ff \circ Fg \quad ;$$

$$F\mathsf{id}_x = \mathsf{id}_{Fx} \quad .$$

So a functor respects typing, distributes over composition and preserves identities.

Remark 2.46 Using $F{\times}F$ to denote the mapping $(f, g :: (Ff, Fg))$, a concise way of expressing these requirements is

$$\mathsf{cod}{\circ}F = F{\circ}\mathsf{cod} \;\wedge\; \mathsf{dom}{\circ}F = F{\circ}\mathsf{dom} \quad ;$$

$$F{\circ}(\circ) \;=\; (\circ) \circ F{\times}F \quad ;$$

$$F{\circ}[\mathsf{id}] = [\mathsf{id}]{\circ}F \quad .$$

So a functor commutes with the basic ingredients of categorical structure.

□

We write, more concisely, $F \in \mathcal{C} \leftarrow \mathcal{D}$ instead of: F is a functor to $\mathcal{C}$ from $\mathcal{D}$. For functors we use capitalised identifiers or non-alphabetic symbols.

Example 2.47 What is a functor $F \in \mathsf{POset}.\mathcal{A} \leftarrow \mathsf{POset}.\mathcal{B}$? Put $\mathcal{A} = (A, \sqsupseteq)$ and $\mathcal{B} = (B, \supseteq)$. Then F is, as far as its object mapping is concerned, a function $F \in A \leftarrow B$. An arrow with typing $x{\leftarrow}y$ in $\mathsf{POset}.\mathcal{B}$, which means that x and y are elements of B with $x \supseteq y$, is mapped by F to an arrow with typing $Fx \leftarrow Fy$ in $\mathsf{POset}.\mathcal{A}$, which is only possible if there is such an arrow, i.e., if $Fx \sqsupseteq Fy$. So F satisfies $Fx \sqsupseteq Fy \;\Leftarrow\; x \supseteq y$; in other words, it is a monotonic function between the two partially ordered sets. It is easy to see that, conversely, any such monotonic function gives a functor: the typing requirements are satisfied, and because of monomorphicity the coherence conditions are trivially true.

□

Example 2.48 (Identity functor) If $\mathcal{C}$ is a category, then $\mathsf{Id}_{\mathcal{C}}$ denotes the *identity functor* on $\mathcal{C}$. It maps objects and arrows of $\mathcal{C}$ to themselves. The subscript $\mathcal{C}$ is usually omitted.

□

Example 2.49 (`map`; `square`) In Fun, let L map a type x to the type 'list of x'. Take for the arrow mapping of L the well-known higher-order function `map` from functional languages. Then L is a functor. The reader should check that the requirements are fulfilled.

Similarly, define in some functional language:

```
type Square x = (x, x)

square f (a, b) = (f a, f b)
```

This also defines an endofunctor.

□

Example 2.50 (Constant functor) As a final example, given an object a in some category $\mathcal{C}$, the *constant functor* $\mathsf{K}.a \in \mathcal{C} \leftarrow \mathcal{D}$ maps each object of some category $\mathcal{D}$ to a. For the arrow mapping we must make sure that $(\mathsf{K}.a)f \in a \leftarrow a$ for all $\mathcal{D}$-arrows f, which we accomplish by mapping all $\mathcal{D}$-arrows to id_a.

□

Extending an object mapping to a functor

It is often the case that we have an object mapping and want to extend it to a functor. Typically, we first construct a candidate for the arrow mapping that satisfies the typing requirement, and then check that it meets the coherence conditions.

The converse is never necessary: given an arrow mapping F_{arr} that preserves composition and identities, there is always a *unique* way of extending it to a full-fledged functor, namely by defining: $F_{obj} = \mathsf{cod} \circ F_{arr} \circ [\mathsf{id}]$.

Functor composition

Since functors are (pairs of) mappings, we can compose these mappings, and the result is again a functor. We denote functor composition also by juxtaposition. This introduces a harmless ambiguity in a case like FGh, since the two interpretations, $(F \circ G).h$ and $F.(G.h)$, denote the same arrow.

We have, apparently, the ingredients for introducing yet another category.

Definition 2.51 (Cat) The category of small categories, denoted by Cat, is defined as follows:

Objects: The small categories.

Arrows: The functors between small categories.

Composition: Functor composition.

Identities: The identity functors.

Strictly speaking this is a precategory, so the triple trick is assumed to have been applied. The restriction to small categories serves to avoid Russell's paradox. (Note that Cat itself is, like Fun, not a small category.)

□

Theorem 2.52 A functor between two categories is also a functor between their opposites; more precisely, $F \in \mathcal{C} \leftarrow \mathcal{D} \equiv F \in \mathcal{C}^{op} \leftarrow \mathcal{D}^{op}$.

□

The proof is left as an exercise to the reader.

Definition 2.53 (Contravariant) In a context in which we are discussing categories $\mathcal{C}$ and $\mathcal{D}$, a functor F is called *contravariant* if it has the typing: $F \in \mathcal{C} \leftarrow \mathcal{D}^{op}$ (or, equivalently, $F \in \mathcal{C}^{op} \leftarrow \mathcal{D}$) .

□

(Note that this is not an intrinsic property of the functor, but depends on the context.)

Example 2.54 Given a relation $R \in x \leftarrow y$, its converse $R^{\cup} \in y \leftarrow x$ is defined by: $R^{\cup} = \$(v, u :: uRv)$. Then $(R \circ S)^{\cup} = S^{\cup} \circ R^{\cup}$, and identity relations are unchanged, so the arrow mapping $\_^{\cup}$ determines a contravariant functor (to Rel^{op} from Rel). For the object mapping we pattern-match $R^{\cup} \in x^{\cup} \overset{\mathsf{Rel}^{op}}{\longleftarrow} y^{\cup}$ against $R^{\cup} \in y \overset{\mathsf{Rel}}{\longleftarrow} x$, giving $x^{\cup} = x$.

□

To emphasize that some functor is 'normal', that is, not contravariant (in the context of two given categories), it may be called *covariant*. Endofunctors are necessarily covariant. In a composition FG, the result is contravariant if one of the two components is contravariant and the other covariant. The result is covariant if both components are covariant, or if both are contravariant. Note in particular that for $F \in \mathcal{C} \leftarrow \mathcal{C}^{op}$ the composition FF is an endofunctor.

Definition 2.55 (Endofunctor) A functor F is called an *endofunctor* on $\mathcal{C}$ if: $F \in \mathcal{C} \leftarrow \mathcal{C}$.

□

Definition 2.56 (Bifunctor) A functor $\oplus$ is called a *bifunctor* (or *binary functor)* if: its source category is a product category, i.e., it has typing $\oplus \in \mathcal{C} \leftarrow \mathcal{D}\times\mathcal{E}$ for some $\mathcal{C}$, $\mathcal{D}$ and $\mathcal{E}$.

□

As suggested by the symbol, we will often use an infix notation for bifunctors, as in $x\oplus y$. The functorial properties, worked out for a bifunctor, give us:

$$f\oplus g \in u\oplus x \xleftarrow{\mathcal{C}} v\oplus y \Leftarrow f \in u \xleftarrow{\mathcal{D}} v \wedge g \in x \xleftarrow{\mathcal{E}} y \ ;$$

$$(f\circ h)\oplus(g\circ k) \ = \ f\oplus g \circ h\oplus k \ ;$$

$$\mathsf{id}_x\oplus\mathsf{id}_y = \mathsf{id}_{x\oplus y} \ .$$

Given a bifunctor, we can take a section by freezing one of its arguments to an object, as in '$x\oplus$' for $x\in\mathcal{D}$. For the object mapping of $\oplus$ the meaning is clear. For the arrow mapping, this section should be understood to stand for $\mathsf{id}_x\oplus$. We shall often appeal implicitly to the following theorem:

Theorem 2.57 Given a bifunctor $\oplus \in \mathcal{C} \leftarrow \mathcal{D}\times\mathcal{E}$, both of the sections $x\oplus$ and $\oplus y$ are functors for all $x\in\mathcal{D}$ and $y\in\mathcal{E}$, with typing: $x\oplus \in \mathcal{C} \leftarrow \mathcal{E}$ and $\oplus y \in \mathcal{C} \leftarrow \mathcal{D}$.

□

The functoriality requirements give us this commutation rule for suitably typed arrows f and g to which sectioned bifunctors have been applied:

$$f\oplus\mathsf{id} \circ \mathsf{id}\oplus g \ = \ \mathsf{id}\oplus g \circ f\oplus\mathsf{id} \ . \qquad \text{SECTION-COMMUTE}$$

Here we have omitted the subscripts to id, which can be reconstructed from a typing for f and g.

A counterpart to the last rule is given by the following theorem, which as it were allows us to 'reconstruct' a bifunctor from its left and right sections.

Theorem 2.58 Given an object mapping $\oplus$ to the objects of $\mathcal{C}$ from the objects of $\mathcal{D}\times\mathcal{E}$, and two collections of functors, $F_x \in \mathcal{C} \leftarrow \mathcal{D}$ for each $x\in\mathcal{E}$, and $G_u \in \mathcal{C} \leftarrow \mathcal{E}$ for each $u\in\mathcal{D}$, if together they satisfy, first, for all $x\in\mathcal{E}$ and $u\in\mathcal{D}$, the typing conditions:

$$F_x u = u\oplus x = G_u x \quad ,$$

and, moreover, for all $f \in u \xleftarrow{\mathcal{D}} v$ and $g \in x \xleftarrow{\mathcal{E}} y$, the commutation property:

$$F_x f \circ G_v g \;=\; G_u g \circ F_y f \quad ,$$

then: $\oplus$ can be extended to a bifunctor $\oplus \in \mathcal{C} \leftarrow \mathcal{D}\times\mathcal{E}$ by defining its arrow mapping for f and g typed as above by: $f\oplus g \;=\; F_x f \circ G_v g$.

Proof For the typing conditions on $\oplus$ we have:

$$\begin{array}{ll}
 & f\oplus g \in u\oplus x \leftarrow v\oplus y \\
\equiv & \quad \{ \quad \text{definition of arrow mapping } \oplus \quad \} \\
 & F_x f \circ G_v g \in u\oplus x \leftarrow v\oplus y \\
\Leftarrow & \quad \{ \quad \circ\text{-TYPING} \quad \} \\
 & F_x f \in u\oplus x \leftarrow v\oplus x \;\wedge\; G_v g \in v\oplus x \leftarrow v\oplus y \\
\equiv & \quad \{ \quad \text{typing conditions on } F \text{ and } G \quad \} \\
 & F_x f \in F_x u \leftarrow F_x v \;\wedge\; G_v g \in G_v x \leftarrow G_v y \\
\Leftarrow & \quad \{ \quad F_x \text{ and } G_v \text{ are functors} \quad \} \\
 & f \in u \leftarrow v \;\wedge\; g \in x \leftarrow y \quad .
\end{array}$$

In verifying the coherence conditions we omit the subscripts to F and G. For all suitably typed f, g, h and k:

$$\begin{array}{ll}
 & (f\circ h)\oplus(g\circ k) \\
= & \quad \{ \quad \text{definition of arrow mapping } \oplus \quad \} \\
 & F(f\circ h) \circ G(g\circ k) \\
= & \quad \{ \quad F \text{ and } G \text{ are functors} \quad \} \\
 & F f \circ F h \circ G g \circ G k \\
= & \quad \{ \quad \text{given commutation property} \quad \}
\end{array}$$

$$
\begin{array}{ll}
 & F f \circ G g \circ F h \circ G k \\
= & \quad \{ \quad \text{definition of arrow mapping } \oplus \quad \} \\
 & f \oplus g \circ h \oplus k \quad .
\end{array}
$$

Further:

$$
\begin{array}{ll}
 & \mathsf{id} \oplus \mathsf{id} \\
= & \quad \{ \quad \text{definition of arrow mapping } \oplus \quad \} \\
 & F\mathsf{id} \circ G\mathsf{id} \\
= & \quad \{ \quad F \text{ and } G \text{ are functors} \quad \} \\
 & \mathsf{id} \quad .
\end{array}
$$

□

Exercise 2.59 Define the arrow mapping $\ll$ by: $x \ll \ = \mathsf{K}.x$ for all $x \in \mathcal{C}$. Show that $\ll$ can be extended to a bifunctor $\ll \in \mathcal{C} \leftarrow \mathcal{C} \times \mathcal{D}$.

□

Exercise 2.60 Give a simple example of an object mapping that can be extended in two *different* ways to a functor. (Hint. Take the monoid $\mathcal{B}_{\equiv} = (\mathbb{B}, \equiv, \mathsf{true})$, where $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$, and consider the endofunctors of $\mathsf{Mon}.\mathcal{B}_{\equiv}$.)

□

Exercise 2.61 Prove Theorem 2.52.

□

Exercise 2.62 Prove Theorem 2.57.

□

Exercise 2.63 Prove the rule SECTION-COMMUTE.

□

2.6 Natural Transformations

Definition 2.64 (Natural transformation) Given two categories $\mathcal{C}$ and $\mathcal{D}$ and two functors F,$G \in \mathcal{C} \leftarrow \mathcal{D}$, a *natural transformation* to F from G is: a transformation η on $\mathcal{D}$ such that for all $x \in \mathcal{D}$:

$$\eta_x \in Fx \xleftarrow{\mathcal{C}} Gx$$

and for each arrow $f \in x \leftarrow y$ in $\mathcal{D}$

$$Ff \circ \eta_y \;=\; \eta_x \circ Gf \quad .$$

We abbreviate this naturality condition on η to: $\eta \in F \dot{\leftarrow} G$.

□

The coherence condition above corresponds to the fact that an arrow to Fx from Gy can be constructed in two different ways. The simplest example of a natural transformation is $\mathsf{id} \in \mathsf{Id} \dot{\leftarrow} \mathsf{Id}$.

Definition 2.65 (Composition of natural transformations) Given two natural transformations $\eta \in F \dot{\leftarrow} G$ and $\tau \in G \dot{\leftarrow} H$, where F,G,$H \in \mathcal{C} \leftarrow \mathcal{D}$, we define their composition $\eta \circ \tau \in F \dot{\leftarrow} H$ by:

(2.66) $\quad (\eta \circ \tau)_x = \eta_x \circ \tau_x$ for all $x \in \mathcal{D}$.

□

The reader should check that $\eta \circ \tau$, thus defined, is indeed a natural transformation and has typing $\eta \circ \tau \in F \dot{\leftarrow} H$.

We define next two ways to combine a functor with a natural transformation.

Definition 2.67 ($F\eta$; η_F) Given a functor F and a natural transformation $\eta \in G \dot{\leftarrow} H$, where $F \in \mathcal{C} \leftarrow \mathcal{D}$ and G,$H \in \mathcal{D} \leftarrow \mathcal{E}$, we define $F\eta \in FG \dot{\leftarrow} FH$ by:

(2.68) $\quad [F\eta] = F \circ [\eta] \quad .$

Further, given a natural transformation $\eta \in G \dot{\leftarrow} H$ and a functor F, where G,$H \in \mathcal{C} \leftarrow \mathcal{D}$ and $F \in \mathcal{D} \leftarrow \mathcal{E}$, we define $\eta_F \in GF \dot{\leftarrow} HF$ by:

(2.69) $\quad [\eta_F] = [\eta] \circ F \quad .$

□

So $(F\eta)_x = F(\eta_x)$ and $(\eta_F)_x = \eta_{(Fx)}$. This shows we can omit the brackets and write $F\eta_x$ and η_{Fx} without ambiguity.

For an arrow $\eta_x \in Fx \leftarrow Gx$ we shall often drop the subscript x and rely on available type information for its reconstruction. In this way we mimic in a sense—typographically—what is standard in some polymorphic-typing contexts. The dot over the arrow in $\eta \in F \dot{\leftarrow} G$ may serve to stress that we mean the transformation, and not just some suitably typed member of the family.

Exercise 2.70 Another notation for $(\!|x|\!)$ that one finds in the literature is ¡$_x$. Show that ¡ is a natural transformation.

□

Exercise 2.71 Given categories $\mathcal{C}$ and $\mathcal{D}$, define their so-called *functor category*, whose objects are the functors to $\mathcal{C}$ from $\mathcal{D}$ and whose arrows are natural transformations.

□

Exercise 2.72 A *natural isomorphism* is: an isomorphism between two functors. Work out the consequences of this definition, using the result of the previous exercise.

□

Chapter 3

Sum and Product

3.1 Lattices

Definition 3.1 (Join) In a partially ordered set $(A, \sqsupseteq)$ an element j of A is called the *join* of two elements x and y of A if: $\forall(z :: z \sqsupseteq j \ \equiv \ z \sqsupseteq x \wedge z \sqsupseteq y)$.

□

By the substitution $z \ := \ j$ we obtain $j \sqsupseteq x \wedge j \sqsupseteq y$, so the join is a common upper bound of x and y. Further, if z is any upper bound of x and y, we see that $z \sqsupseteq j$, so j is the *least* upper bound. The antisymmetry of $\sqsupseteq$ tells us now that the join of x and y, if it exists, is unique, and we denote it by $x \sqcup y$. Using this notation we have as the characterisation of join:

$$z \sqsupseteq x \sqcup y \ \equiv \ z \sqsupseteq x \wedge z \sqsupseteq y \ .$$

Definition 3.2 (Join semilattice) A partially ordered set is called a *join semilattice* if: any pair of elements has a join.

□

It is easy to see from the characterisation that $x \sqsupseteq y \ \equiv \ x = x \sqcup y$, and that the binary operation $\sqcup$ is symmetric: $x \sqcup y = y \sqcup x$, associative: $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$, and idempotent: $x \sqcup x = x$. (Conversely, given any symmetric, associative and idempotent operation on a set, the set can be partially ordered by *defining* $x \sqsupseteq y \ \equiv \ x = x \sqcup y$.)

Definition 3.3 (Meet, meet semilattice) Dually, the *meet* of x and y is characterised by: $\forall(z :: x \sqcap y \sqsupseteq z \ \equiv \ x \sqsupseteq z \wedge y \sqsupseteq z)$. (The meet in $(A, \sqsupseteq)$ is the join in $(A, \sqsupseteq^{\cup})$.) A *meet semilattice* is: a partially ordered set in which each pair of elements has a meet.

□

Definition 3.4 (Lattice) A *lattice* is: a partially ordered set that is both a *join* and a *meet* semilattice.

□

Join and meet interact in the *absorption laws*: $x \sqcup (x \sqcap y) = x = x \sqcap (x \sqcup y)$.

The most familiar example of a lattice is probably that of the subsets of a set, ordered by $\supseteq$ ('contains'), in which join is set union, and meet set intersection. Another example is given by the naturals partially ordered by the divisibility relation; join is then lcm (least common multiple), and meet is gcd (greatest common divisor). (Note that in this lattice 0 is on top, since it is divisible by any natural—including itself since we need to have reflexivity of the relation.)

The concepts of join and meet in lattice theory are captured in category theory by the concepts of *sum* and *product*. We present a detailed treatment of the notion of sum, and then introduce products as the dual notion.

3.2 Sum

3.2.1 From suprema to sums

Recall that in lattice theory the join $x \sqcup y$ of x and y (if it exists) is characterised by:

$$\forall (z :: z \sqsupseteq x \sqcup y \;\equiv\; z \sqsupseteq x \wedge z \sqsupseteq y) \quad .$$

A first attempt towards a categorical generalisation looks like this:

$$h \in z \leftarrow x + y \equiv f \in z \leftarrow x \wedge g \in z \leftarrow y \quad .$$

We have replaced $x \sqcup y$ by $x + y$, and introduced 'witnessing' arrows. A question is how to universally quantify them. Quantifying over all h, f and g is clearly too strong in general; there has to be some 'organic' relationship between these three. Here we only sketch, informally, the nature of this relationship. Below we give a precise definition—which, however, is highly abstract. Apart from the elegance of this abstract definition, it is also paradigmatic for the general style of categorical definitions, and thus paves the way for the introduction of more advanced concepts.

A major ingredient of the relationship between h, f and g is that there is a *one-to-one* correspondence between—on the lhs of the equivalence—the arrows h whose typing is given by $h \in z \leftarrow x + y$, and—on the rhs of the equivalence—the pairs of arrows (f, g) whose typing is $f \in z \leftarrow x$ and $g \in z \leftarrow y$. If h and (f, g) are in that correspondence, each is determined by the other. This can be expressed by introducing names for the translations in both directions, thus:

$$h = f \triangledown g \quad \text{and} \quad (f, g) = (h \circ \mathsf{inl}, h \circ \mathsf{inr}) \quad ,$$

in which the fact that the translation from h to (f, g) has this particular form is suggested by typing considerations. Furthermore, these two translations have to be each others' inverse, which can be expressed by:

$$h = f \triangledown g \;\equiv\; (f, g) = (h{\circ}\mathsf{inl}, h{\circ}\mathsf{inr}) \quad .$$

So the categorical concept of *sum* consists of three ingredients: a binary object mapping $+$, a pair of arrows $(\mathsf{inl}, \mathsf{inr})$, and a binary operation $\triangledown$ on certain pairs of arrows, namely: those having the same codomain (called z above). In the categorical definition in the next chapter we transform the lattice-theoretical notion that a supremum is a *least* upper bound into a statement involving *initiality*.

3.2.2 Definition of sum

Assume that we have some category $\mathcal{C}$. On top of this base category we construct new categories, so-called *cocone categories*.

Definition 3.5 (Cocone) Given $x, y{\in}\mathcal{C}$, a *cocone* for x and y is: a triple (z,f,g) in which $z{\in}\mathcal{C}$, $f \in z \leftarrow x$ and $g \in z \leftarrow y$ (so f and g have the same codomain).

□

Definition 3.6 (Cocone category) Given $x, y{\in}\mathcal{C}$, the *cocone category* for x and y, denoted $x \Uparrow y$, is defined as follows:

Objects: The cocones for x and y.

Arrows: The arrows in $(z,f,g) \overset{x\Uparrow y}{\longleftarrow} (s,p,q)$ are arrows of $\mathcal{C}$ as restricted by the following condition:

$$h \in (z,f,g) \overset{x\Uparrow y}{\longleftarrow} (s,p,q) \quad\equiv\quad h \in z \overset{\mathcal{C}}{\longleftarrow} s \;\wedge\; f = h{\circ}p \;\wedge\; g = h{\circ}q \quad .$$

Composition: That of $\mathcal{C}$.

Identities: Those of $\mathcal{C}$, where $\mathsf{id}_{(z,f,g)}$ in $x \Uparrow y$ is id_z in $\mathcal{C}$.

(To be precise, this is a precategory, which can be turned into a category with the triple trick.)

□

Associativity of composition and neutrality of identities are met because they are inherited from the base category. It must be checked, though, that arrow composition preserves the restriction on arrows (in which the arrow equalities are the coherence condition for the two ways of constructing arrows with typings $z{\leftarrow}x$ and $z{\leftarrow}y$). The verification is as follows:

$$
\begin{array}{ll}
 & h{\circ}k \in (z,f,g) \overset{x\Uparrow y}{\longleftarrow} (t,u,v) \\
\equiv & \quad \{ \quad \text{definition of } \overset{x\Uparrow y}{\longleftarrow} \quad \} \\
 & h{\circ}k \in z \overset{\mathcal{C}}{\longleftarrow} t \ \wedge \ f = h{\circ}k{\circ}u \ \wedge \ g = h{\circ}k{\circ}v \\
\Leftarrow & \quad \{ \quad \text{properties of } \circ \quad \} \\
 & h \in z \overset{\mathcal{C}}{\longleftarrow} s \ \wedge \ f = h{\circ}p \ \wedge \ g = h{\circ}q \ \wedge \ k \in s \overset{\mathcal{C}}{\longleftarrow} t \ \wedge \ p = k{\circ}u \ \wedge \ q = k{\circ}v \\
\equiv & \quad \{ \quad \text{definition of } \overset{x\Uparrow y}{\longleftarrow} \quad \} \\
 & h \in (z,f,g) \overset{x\Uparrow y}{\longleftarrow} (s,p,q) \wedge k \in (s,p,q) \overset{x\Uparrow y}{\longleftarrow} (t,u,v) \ .
\end{array}
$$

Now we can define the notion of sum:

Definition 3.7 (Sum) A *sum* of two objects x and y is: an initiality of $x \Uparrow y$.

□

As in all categorical definitions based on initiality, sums are unique 'up to isomorphism'. Another name for 'sum' that occurs in the literature is: *coproduct*.

Definition 3.8 (Has sums) A category *has sums* if: all pairs of objects have a sum.

□

An example of a category that has no sums is the discrete category with at least two objects. For recall that in a discrete category all arrows are identities. So all cocones are of the form $(z,\mathsf{id}_z,\mathsf{id}_z)$, which is a cocone of z and z. So if $x \neq y$, there are no cocones of x and y; in other word, $x \Uparrow y$ has no objects and *a fortiori* no initial objects.

3.2.3 Properties of sum

Throughout the following we assume that $\mathcal{C}$ has sums. Moreover, we assume that for each pair of objects (x, y), from among the —possibly many— sums of x and y , one is chosen to be 'the' sum. Which one is chosen is entirely irrelevant, since all are isomorphic and so have the same categorical properties. As defined, the sum of x and y is a pair, consisting of an initial object a of $x \Uparrow y$ (by itself a triple) and the $(\!|\_|\!)$ mapping from the objects of $x \Uparrow y$ to arrows of $x \Uparrow y$. Let us first give new names to the components of a, so that a equals

$$(x{+}y \ , \ \mathsf{inl}_{x,y} \ , \ \mathsf{inr}_{x,y}) \ .$$

The fact that a is an object of $x \Uparrow y$ gives us:

$$x{+}y \in \mathcal{C} \ ,$$

$$\mathsf{inl}_{x,y} \in x{+}y \leftarrow x \ , \qquad \mathsf{inl}\text{-TYPING}$$

$$\mathsf{inr}_{x,y} \in x{+}y \leftarrow y \quad . \qquad \mathsf{inr}\text{-TYPING}$$

Remark 3.9 It is general usage also to say 'the sum of x and y' when referring to the *object* $x{+}y$; in practice this does not lead to confusion.

□

Next, from $(\!|\_|\!)$-CHAR, we have, for $f \in z \leftarrow x$ and $g \in z \leftarrow y$:

$$(\!|(z,f,g)|\!) = h \;\equiv\; h \in (z,f,g) \overset{x\Uparrow y}{\longleftarrow} (x{+}y \;,\; \mathsf{inl}_{x,y} \;,\; \mathsf{inr}_{x,y}) \quad .$$

Before we look into the detailed consequences of this proposition, we introduce a more pleasant notation for $(\!|(z,f,g)|\!)$. Note that z is in fact redundant, since it is equal to $\mathsf{cod}.f$ (and also to $\mathsf{cod}.g$). Thus we can convey the same information using only f and g as arguments, which means that we can use a binary operator, and we define:

$$f \triangledown g = (\!|(z,f,g)|\!) \text{ where } z = \mathsf{cod}.f = \mathsf{cod}.g \quad .$$

Using this notation and unfolding the definition of $\overset{x\Uparrow y}{\longleftarrow}$, we obtain (still under the typing assumption $f \in z \leftarrow x \;\wedge\; g \in z \leftarrow y$):

$$f \triangledown g = h \quad\equiv\quad h \in z \leftarrow x{+}y \;\wedge\; f = h{\circ}\mathsf{inl}_{x,y} \;\wedge\; g = h{\circ}\mathsf{inr}_{x,y} \quad .$$

By the substitution $h := f \triangledown g$, and making the typing assumption explicit, we find this typing rule for $\triangledown$:

$$f \triangledown g \in z \leftarrow x{+}y \Leftarrow f \in z \leftarrow x \;\wedge\; g \in z \leftarrow y \quad . \qquad \triangledown\text{-TYPING}$$

Recall that, when writing an arrow equality like $f \triangledown g = h$, we imply that both sides can be typed and have the same type. The conjunct '$h \in z \leftarrow x{+}y$' in the unfolded definition of $\overset{x\Uparrow y}{\longleftarrow}$ may therefore be omitted, giving the characterisation rule:

$$f \triangledown g = h \quad\equiv\quad f = h{\circ}\mathsf{inl}_{x,y} \;\wedge\; g = h{\circ}\mathsf{inr}_{x,y} \quad . \qquad \triangledown\text{-CHAR}$$

Up to now, all we have done is to rephrase the abstract definition of sum in a concrete but *equivalent* way. Thus, we can give the following alternative definition of sum:

Theorem 3.10 $((x{+}y \;,\; \mathsf{inl}_{x,y} \;,\; \mathsf{inr}_{x,y}), \triangledown)$ is a sum of x and y if inl-TYPING, inr-TYPING, $\triangledown$-TYPING and $\triangledown$-CHAR are satisfied.

□

Just as for initiality in general, we obtain a number of useful rules from the characterisation rule by simple substitutions. The substitution $h := f \triangledown g$ gives us the computation rule:

$$f \;=\; f \triangledown g \circ \mathsf{inl}_{x,y} \quad\wedge\quad g \;=\; f \triangledown g \circ \mathsf{inr}_{x,y} \quad . \qquad \triangledown\text{-COMP}$$

The substitution $f,g,h := \mathsf{inl}_{x,y},\mathsf{inr}_{x,y},\mathsf{id}_{x+y}$ gives us the identity rule:

$$\mathsf{inl}_{x,y} \triangledown \mathsf{inr}_{x,y} = \mathsf{id}_{x+y} \quad . \qquad \triangledown\text{-ID}$$

Finally, we derive the fusion rule:

$$\begin{array}{ll} & p \triangledown q \;=\; h \circ f \triangledown g \\ \equiv & \quad \{ \quad \triangledown\text{-CHAR with } f,g,h := p \;,\; q \;,\; h \circ f \triangledown g \quad \} \\ & p \;=\; h \circ f \triangledown g \circ \mathsf{inl}_{x,y} \;\;\wedge\;\; q \;=\; h \circ f \triangledown g \circ \mathsf{inr}_{x,y} \\ \equiv & \quad \{ \quad \triangledown\text{-COMP} \quad \} \\ & p = h{\circ}f \;\wedge\; q = h{\circ}g \\ \Leftarrow & \quad \{ \quad p,q := h{\circ}f \;,\; h{\circ}g \quad \} \\ & \mathsf{true} \quad , \end{array}$$

which proves:

$$(h{\circ}f) \triangledown (h{\circ}g) \;=\; h \circ f \triangledown g \quad . \qquad \triangledown\text{-FUSION}$$

By the substitution $f,g := \mathsf{inl}_{x,y},\mathsf{inr}_{x,y}$, and using $\triangledown$-ID, we get an additional rule

$$h = (h{\circ}\mathsf{inl}_{x,y}) \triangledown (h{\circ}\mathsf{inr}_{x,y}) \quad . \qquad \triangledown\text{-FORM}$$

Note that the implicit typing requirement here is that $h \in z \leftarrow x{+}y$ for some z: not all arrows can be brought into $\triangledown$-form, but only those having a sum object as their domain.

The following lemma is often useful, either by itself or in combination with $\triangledown$-FORM, for rewriting an arrow equality involving $\triangledown$ to a pair of simpler equations:

Lemma 3.11 $\triangledown$ is injective, that is, for all suitably typed f, g, h and k:

$$f \triangledown g = h \triangledown k \;\equiv\; f = h \wedge g = k \quad .$$

Proof By cyclic implication.

$$\begin{array}{ll} & f \triangledown g = h \triangledown k \\ \Leftarrow & \quad \{ \quad \text{LEIBNIZ} \quad \} \\ & f = h \wedge g = k \\ \Leftarrow & \quad \{ \quad \triangledown\text{-COMP} \quad \} \\ & f \triangledown g \circ \mathsf{inl} \;=\; h \triangledown k \circ \mathsf{inl} \;\;\wedge\;\; f \triangledown g \circ \mathsf{inr} \;=\; h \triangledown k \circ \mathsf{inr} \\ \Leftarrow & \quad \{ \quad \text{LEIBNIZ} \quad \} \\ & f \triangledown g = h \triangledown k \quad . \end{array}$$

□ There is more to sums. The operation + takes pairs of $\mathcal{C}$-objects to $\mathcal{C}$-objects, and so behaves like the object part of a (bi)functor. Can we extend it to a full-fledged functor? The typing requirement is:

$$f{+}g \in x{+}y \leftarrow u{+}v \Leftarrow f \in x \leftarrow u \wedge g \in y \leftarrow v \quad .$$

Can we construct an arrow of this type? Let us calculate:

$$\begin{array}{ll} & f{+}g \in x{+}y \leftarrow u{+}v \\ \Leftarrow & \quad \{ \quad \text{arrow has a sum domain, so try } f{+}g := p \triangledown q \quad \} \\ & p \triangledown q \in x{+}y \leftarrow u{+}v \\ \Leftarrow & \quad \{ \quad \triangledown\text{-TYPING} \quad \} \\ & p \in x{+}y \leftarrow u \wedge q \in x{+}y \leftarrow v \\ \Leftarrow & \quad \{ \quad p{,}q := \mathsf{inl}{\circ}f \text{ , } \mathsf{inr}{\circ}g\text{; } \mathsf{inl}\text{-TYPING, } \mathsf{inr}\text{-TYPING; } \circ\text{-TYPING} \quad \} \\ & f \in x \leftarrow u \wedge g \in y \leftarrow v \quad . \end{array}$$

So we find as a candidate for the arrow part:

$$f{+}g = (\mathsf{inl}{\circ}f) \triangledown (\mathsf{inr}{\circ}g) \quad . \qquad\qquad +\text{-DEF}$$

(In the last calculation we dropped the subscripts to inl and inr, and we shall generally omit them when they can be inferred from the context.)

Is +, thus defined, a functor? The typing requirement is met, since that is how we constructed the candidate definition. The preservation of identities is immediate from $\triangledown$-ID. For the distribution over composition, we verify:

$$\begin{array}{ll} & f{+}g \circ h{+}k \\ = & \quad \{ \quad +\text{-DEF} \quad \} \\ & (\mathsf{inl}{\circ}f) \triangledown (\mathsf{inr}{\circ}g) \circ (\mathsf{inl}{\circ}h) \triangledown (\mathsf{inr}{\circ}k) \\ = & \quad \{ \quad \triangledown\text{-FUSION} \quad \} \\ & ((\mathsf{inl}{\circ}f) \triangledown (\mathsf{inr}{\circ}g) \circ \mathsf{inl} \circ h) \triangledown ((\mathsf{inl}{\circ}f) \triangledown (\mathsf{inr}{\circ}g) \circ \mathsf{inr} \circ k) \\ = & \quad \{ \quad \triangledown\text{-COMP} \quad \} \\ & (\mathsf{inl}{\circ}f{\circ}h) \triangledown (\mathsf{inr}{\circ}g{\circ}k) \\ = & \quad \{ \quad +\text{-DEF} \quad \} \\ & (f{\circ}h){+}(g{\circ}k) \quad . \end{array}$$

Further properties of the functor + are the following isomorphisms, for all objects x, y and z of $\mathcal{C}$:

$$x{+}y \cong y{+}x \quad ,$$

$$(x{+}y){+}z \cong x{+}(y{+}z) \quad .$$

For the first one, we may argue that this is obvious in view of the fact that, due to symmetries in the definition, $y{+}x$ is *a* (although perhaps not *the*) sum of x and y. We can also give the isomorphism explicitly, which in both directions is $\mathsf{inr} \triangledown \mathsf{inl}$—with different subscripts. The proof that this is an isomorphism, as well as the proof of the 'associativity' isomorphism, are left to the reader as an exercise. Finally, if $\mathcal{C}$ has some initial object 0,

$$x{+}0 \cong x \quad .$$

Implicit in the proof above that $+$ distributes over composition is another fusion rule, namely:

$$(f{\circ}h) \triangledown (g{\circ}k) \;=\; f \triangledown g \circ h{+}k \quad . \qquad \triangledown\text{-}{+}\text{-FUSION}$$

The following two rules tell us that inl and inr are natural transformations:

$$f{+}g \circ \mathsf{inl} \;=\; \mathsf{inl}{\circ}f \quad . \qquad \mathsf{inl}\text{-LEAP}$$

$$f{+}g \circ \mathsf{inr} \;=\; \mathsf{inr}{\circ}g \quad . \qquad \mathsf{inr}\text{-LEAP}$$

The (easy) proofs are left to the reader.

To conclude this section, we summarise all the rules derived. In using these rules, remember that they are only valid when the arrow equalities are suitably typed. This warning applies in particular to $\triangledown$-ID and $\triangledown$-FORM.

$$\mathsf{inl} \in x{+}y \leftarrow x \quad . \qquad \mathsf{inl}\text{-TYPING}$$

$$\mathsf{inr} \in x{+}y \leftarrow y \quad . \qquad \mathsf{inr}\text{-TYPING}$$

$$f \triangledown g \in z \leftarrow x{+}y \Leftarrow f \in z \leftarrow x \wedge g \in z \leftarrow y \quad . \qquad \triangledown\text{-TYPING}$$

$$f \triangledown g = h \;\equiv\; f = h{\circ}\mathsf{inl} \;\wedge\; g = h{\circ}\mathsf{inr} \quad . \qquad \triangledown\text{-CHAR}$$

$$f \;=\; f \triangledown g \circ \mathsf{inl} \quad . \qquad \triangledown\text{-COMP}$$

$$g \;=\; f \triangledown g \circ \mathsf{inr} \quad . \qquad \triangledown\text{-COMP}$$

$$\mathsf{inl} \triangledown \mathsf{inr} = \mathsf{id} \quad . \qquad \triangledown\text{-ID}$$

$$(h{\circ}f) \triangledown (h{\circ}g) \;=\; h \circ f \triangledown g \quad . \qquad \triangledown\text{-FUSION}$$

$$h = (h{\circ}\mathsf{inl}) \triangledown (h{\circ}\mathsf{inr}) \quad . \qquad \triangledown\text{-FORM}$$

$$f{+}g = (\mathsf{inl}{\circ}f) \triangledown (\mathsf{inr}{\circ}g) \quad . \qquad {+}\text{-DEF}$$

$$(f{\circ}h) \triangledown (g{\circ}k) \;=\; f \triangledown g \circ h{+}k \quad . \qquad \triangledown\text{-}{+}\text{-FUSION}$$

$$f{+}g \circ \mathsf{inl} \;=\; \mathsf{inl}{\circ}f \quad . \qquad \mathsf{inl}\text{-LEAP}$$

$$f{+}g \circ \mathsf{inr} \;=\; \mathsf{inr}{\circ}g \quad . \qquad \mathsf{inr}\text{-LEAP}$$

3.3 Product

Products are the dual construction of sums (and conversely; hence the name *coproduct* for sum). The components of the product of x and y are named thus: $(x \times y\ ,\ \mathsf{exl}_{x,y}\ ,\ \mathsf{exr}_{x,y})$, and for $[\![(z,f,g)]\!]$ we write $f \vartriangle g$.

Here we confine ourselves to listing the rules, which can be derived mechanically from those for sum by substituting the new names, and changing the order of all arrows and compositions:

$$\mathsf{exl} \in x \leftarrow x \times y \quad . \qquad \textsf{exl-TYPING}$$

$$\mathsf{exr} \in y \leftarrow x \times y \quad . \qquad \textsf{exr-TYPING}$$

$$f \vartriangle g \in x \times y \leftarrow z \Leftarrow f \in x \leftarrow z \ \wedge\ g \in y \leftarrow z \quad . \qquad \vartriangle\textsf{-TYPING}$$

$$f \vartriangle g = h \quad \equiv \quad f = \mathsf{exl} \circ h \ \wedge\ g = \mathsf{exr} \circ h \quad . \qquad \vartriangle\textsf{-CHAR}$$

$$f \ = \ \mathsf{exl} \circ f \vartriangle g \quad . \qquad \vartriangle\textsf{-COMP}$$

$$g \ = \ \mathsf{exr} \circ f \vartriangle g \quad . \qquad \vartriangle\textsf{-COMP}$$

$$\mathsf{exl} \vartriangle \mathsf{exr} = \mathsf{id} \quad . \qquad \vartriangle\textsf{-ID}$$

$$(f \circ h) \vartriangle (g \circ h) \ = \ f \vartriangle g \circ h \quad . \qquad \vartriangle\textsf{-FUSION}$$

$$h = (\mathsf{exl} \circ h) \vartriangle (\mathsf{exr} \circ h) \quad . \qquad \vartriangle\textsf{-FORM}$$

$$f \times g = (f \circ \mathsf{exl}) \vartriangle (g \circ \mathsf{exr}) \quad . \qquad \times\textsf{-DEF}$$

$$(h \circ f) \vartriangle (k \circ g) \ = \ h \times k \circ f \vartriangle g \quad . \qquad \times\textsf{-}\vartriangle\textsf{-FUSION}$$

$$\mathsf{exl} \circ f \times g \ = \ f \circ \mathsf{exl} \quad . \qquad \textsf{exl-LEAP}$$

$$\mathsf{exr} \circ f \times g \ = \ g \circ \mathsf{exr} \quad . \qquad \textsf{exr-LEAP}$$

3.4 Examples of sums and products

3.4.1 POset.$\mathcal{A}$

Assume that $\mathcal{A}$ is a partially ordered set. We have already seen that the sum corresponds to $\sqcup$; the product corresponds of course, dually, to $\sqcap$. Since this category is monomorphic, the other ingredients of categorical sum and product are thereby fixed.

3.4.2 Fun

Sum in Fun corresponds to *disjoint union* $\uplus$ (and not to normal set union $\cup$!).

A possible implementation of disjoint union in terms of sets is to add a 'tag bit' to the values involved, to keep track of the component from which they originated:

$$x \uplus y = \$(c, t : t \in \{0,1\} : c \in x \text{ if } t=0 \text{ else } c \in y) \quad ,$$

$$\mathsf{inl} = (a :: (a, 0)) \quad ,$$

$$\mathsf{inr} = (b :: (b, 1)) \quad ,$$

$$f \triangledown g = (c, t : t \in \{0,1\} : f.c \text{ if } t=0 \text{ else } g.c) \quad .$$

We verify $\triangledown$-CHAR, using the (implicit) typing, and thus establish the condition for Theorem 3.10:

$$\begin{array}{ll}
 & f \triangledown g = h \\
\equiv & \quad \{ \quad \text{EXTENSIONALITY} \quad \} \\
 & \forall(c, t : t \in \{0,1\} : (f \triangledown g).(c,t) = h.(c,t)) \\
\equiv & \quad \{ \quad \text{range split: } t \in \{0,1\} \quad \} \\
 & \forall(c :: (f \triangledown g).(c,0) = h.(c,0)) \wedge \forall(c :: (f \triangledown g).(c,1) = h.(c,1)) \\
\equiv & \quad \{ \quad \text{definition of } f \triangledown g \quad \} \\
 & \forall(c :: f.c = h.(c,0)) \wedge \forall(c :: g.c = h.(c,1)) \\
\equiv & \quad \{ \quad \text{definition of } \mathsf{inl} \text{ and } \mathsf{inr} \quad \} \\
 & \forall(c :: f.c = h.(\mathsf{inl}.c)) \wedge \forall(c :: g.c = h.(\mathsf{inr}.c)) \\
\equiv & \quad \{ \quad \text{FUN-}\circ\text{-COMP} \quad \} \\
 & \forall(c :: f.c = (h \circ \mathsf{inl}).c) \wedge \forall(c :: g.c = (h \circ \mathsf{inr}).c) \\
\equiv & \quad \{ \quad \text{EXTENSIONALITY} \quad \} \\
 & f = h \circ \mathsf{inl} \ \wedge \ g = h \circ \mathsf{inr} \quad .
\end{array}$$

The sum type can be implemented in modern functional languages by using a variant that is used in pattern matches, as in:

```
data Sum x y = inl x | inr y
```

In this context it is easy to understand why sums are only unique 'up to isomorphism', for surely

```
data Choice x y = one x | other y
```

is an equally valid implementation. The operation $\triangledown$ can be implemented by:

```
case f g = h where

   h (inl a) = f a

   h (inr b) = g b
```

Why is normal set union not a sum? A direct answer is that in Fun the objects $x \uplus y$ and $x \cup y$ are in general not isomorphic, and since we have proved one to be the sum, the other is not. But here is a concrete counterexample.

Take for x and y both the same singleton set, say $\{0\}$, take for z the two-element set $\{1,2\}$, and choose for f the constant function $\mathsf{K}.1 \in z \leftarrow x$ and for g the constant function $\mathsf{K}.2 \in z \leftarrow y$. Now assume that sum is implemented as set union, which in this case implies that $x+y = \{0\}$. Then we obtain the following contradiction:

$$
\begin{array}{ll}
 & 1 = 2 \\
\equiv & \quad \{ \quad \text{definition of } \mathsf{K} \quad \} \\
 & (\mathsf{K}.1).0 = (\mathsf{K}.2).0 \\
\equiv & \quad \{ \quad \text{definition of } f \text{ and } g \quad \} \\
 & f.0 = g.0 \\
\equiv & \quad \{ \quad \triangledown\text{-COMP} \quad \} \\
 & (f \triangledown g \circ \mathsf{inl}).0 = (f \triangledown g \circ \mathsf{inr}).0 \\
\equiv & \quad \{ \quad \text{FUN-}\circ\text{-COMP} \quad \} \\
 & (f \triangledown g).(\mathsf{inl}.0) = (f \triangledown g).(\mathsf{inr}.0) \\
\Leftarrow & \quad \{ \quad \text{LEIBNIZ} \quad \} \\
 & \mathsf{inl}.0 = \mathsf{inr}.0 \\
\Leftarrow & \quad \{ \quad \text{symmetry and transitivity of } = \quad \} \\
 & \mathsf{inl}.0 = 0 \ \wedge \ \mathsf{inr}.0 = 0 \\
\equiv & \quad \{ \quad \text{membership of singleton set} \quad \}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{inl}.0 \in \{0\} \ \wedge \ \mathsf{inr}.0 \in \{0\} \\
\equiv & \quad \{ \quad \text{assumption about } x{+}y \quad \} \\
 & \mathsf{inl}.0 \in x{+}y \ \wedge \ \mathsf{inr}.0 \in x{+}y \\
\Leftarrow & \quad \{ \quad \mathsf{inl}\text{-TYPING and } \mathsf{inr}\text{-TYPING} \quad \} \\
 & 0{\in}x \wedge 0{\in}y \\
\equiv & \quad \{ \quad x = y = \{0\} \quad \} \\
 & \mathsf{true} \ .
\end{array}
$$

Product in Fun corresponds to the well-known Cartesian product:

$$x{\times}y = \$(a, b :: a{\in}x \wedge b{\in}y) \ ,$$

$$\mathsf{exl} = (a, b :: a) \ ,$$

$$\mathsf{exr} = (a, b :: b) \ ,$$

$$f \vartriangle g = (c :: (f.c, g.c)) \ .$$

The verification of $\vartriangle$-CHAR is left to the reader.

Products are also simple to implement in a functional language:

```
data Product x y = pair x y

exl (pair a b) = a

exr (pair a b) = b

split f g = h where

   h c = pair (f c) (g c)
```

3.4.3 Rel

In the category of relations sum is as in Fun, where inl and inr are the relations obtained by the standard embedding from functions to relations. The operation $\triangledown$ has to be extended to relations, which, expressed at the point level, can be done as follows:

$$d \ (R \triangledown S) \ (c, t) \quad \equiv \quad (d \ R \ c) \ \mathsf{if} \ t{=}0 \ \mathsf{else} \ (d \ S \ c) \ .$$

The verification that this defines a sum is essentially the same as for Fun.

When it comes to products, there is a surprise: Rel has products, but . . . they are the *same* as the sums. In fact, it is easy to see why this must need be so. The category Rel is self-dual, and so any categorical 'thing' coincides with the co-'thing', as was also the case for initiality and terminality in Rel.

So why does the obvious extension to relations of the product construction in Fun not yield a product here? The problem is, concisely, that this extension gives $R \vartriangle \emptyset = \emptyset$, which is incompatible with $\vartriangle$-COMP, according to which $\mathsf{exl} \circ R \vartriangle \emptyset \ = \ R$.

Exercise 3.12 Show that Cat has products. (Hint: consider the object mapping $(\mathcal{C},\mathcal{D}::C\times\mathcal{D})$, taking two categories to a product category.)

□

Chapter 4

Adjunctions

4.1 Galois connections

Recall the characterisation of join in lattice theory: $z \sqsupseteq x \sqcup y \ \equiv \ z \sqsupseteq x \wedge z \sqsupseteq y$. We can reformulate this as follows. Given a partially ordered set $(A, \sqsupseteq)$, we define the relation $\sqsupseteq^2$ on $A^2 = A \times A$ by: $(u, x) \sqsupseteq^2 (v, y) \ \equiv \ u \sqsupseteq v \wedge x \sqsupseteq y$. Then $(A^2, \sqsupseteq^2)$ is also a partially ordered set. We have now: $z \sqsupseteq x \sqcup y \ \equiv \ (z, z) \sqsupseteq^2 (x, y)$. Writing further $\sqcup.(x, y)$ for $x \sqcup y$ and $\Delta.z$ for (z, z), we obtain:

$$z \sqsupseteq \sqcup.(x, y) \ \equiv \ \Delta.z \sqsupseteq^2 (x, y) \ .$$

This is an example of a so-called *Galois connection*, in this case between $(A, \sqsupseteq)$ and $(A^2, \sqsupseteq^2)$.

Definition 4.1 (Galois connection) A pair of monotonic functions $F \in C \leftarrow D$ and $G \in D \leftarrow C$ forms a *Galois connection* between the partially ordered sets $(C, \sqsupseteq_C)$ and $(D, \sqsupseteq_D)$ if: the following equivalence holds for all $x \in C$ and $y \in D$:

$$x \sqsupseteq_C F.y \ \equiv \ G.x \sqsupseteq_D y \ . \tag{4.2}$$

□

The function F is called the lower *adjoint* and the function G is called the upper *adjoint* of the Galois connection.

Using extensionality, we can express (4.2) in a slightly different way, thereby making it more similar to the definition that will be given below for adjunctions:

$$\begin{array}{ll} & \forall(x, y : x \in C \wedge y \in D : x \sqsupseteq_C F.y \ \equiv \ G.x \sqsupseteq_D y) \\ \equiv & \quad \{ \quad \text{LAMBDA-COMP} \quad \} \\ & \forall(x, y : x \in C \wedge y \in D : (x, y :: \ x \sqsupseteq_C F.y).(x, y) = (x, y :: \ G.x \sqsupseteq_D y).(x, y)) \\ \equiv & \quad \{ \quad \text{EXTENSIONALITY} \quad \} \\ & (x, y :: \ x \sqsupseteq_C F.y) = (x, y :: \ G.x \sqsupseteq_D y) \ . \end{array}$$

So F and G form a Galois connection if the function (in this case, a predicate) mapping the pair x,y to $x \sqsupseteq_C F.y$ is equal to the function mapping the pair x,y to $G.x \sqsupseteq_D y$. In other words, an alternative definition of Galois connection is:

(F, G) forms a Galois connection if: the following functional equality holds:

$$(x, y :: x \sqsupseteq_C F.y) = (x, y :: G.x \sqsupseteq_D y) \quad . \tag{4.3}$$

Galois connections are interesting for a number of reasons. First, we saw that $\sqcup$ was characterised by a Galois connection. More generally, if one of F and G is given, the other is characterised by a Galois connection. Galois connections are further interesting because, as soon as we recognise one, we can immediately deduce a number of useful properties of the adjoints. In particular, if we instantiate (4.2) in such a way that one side becomes true, we obtain two *cancellation properties*. We can express these properties point-free as follows:

$$G \circ F \mathrel{\dot{\sqsupseteq}}_D \mathsf{Id} \quad , \tag{4.4}$$

$$\mathsf{Id} \mathrel{\dot{\sqsupseteq}}_C F \circ G \quad , \tag{4.5}$$

where $\dot{\sqsupseteq}_A$ denotes the partial ordering relation obtained by lifting $\sqsupseteq_A$ on A to a relation on A-valued functions by: $F \mathrel{\dot{\sqsupseteq}}_A G \equiv \forall(x :: F.x \sqsupseteq_A G.x)$, and Id denotes the identity function. Furthermore, from these cancellation properties it is straightforward to prove that both adjoints of a Galois connection are monotonic.

These properties of a Galois connection can be used to give an alternative definition of a Galois connection, equivalent to our previous formulation:

(F, G) forms a Galois connection if the following two clauses hold:

$$G \circ F \mathrel{\dot{\sqsupseteq}}_D I \quad \text{and} \quad I \mathrel{\dot{\sqsupseteq}}_C F \circ G \quad ,$$

F and G are monotonic.

Example 4.6 Using the total (and therefore partial) order $\geq$ on numbers, both the integers $\mathbb{Z}$ and the reals $\mathbb{R}$ are partially ordered sets. Using $\mathsf{float} \in \mathbb{R} \leftarrow \mathbb{Z}$ for the standard embedding of the integers into the reals, we have:

$$r \geq \mathsf{float}.i \;\equiv\; \mathsf{floor}.r \geq i \quad .$$

This characterises floor.

□

Example 4.7 For another example, take propositional logic, in which $\Leftarrow$ (follows from) is a partial order. Denoting the logical connective 'if' (the converse of 'implies') by $\leftarrow$, we have:

$$p \;\Leftarrow\; q \wedge r \quad\equiv\quad p \leftarrow r \;\Leftarrow\; q \quad .$$

(Here $F = (\wedge\, r)$ and $G = (\leftarrow r)$.) All properties of $\leftarrow$ follow from this, given the properties of $\wedge$, and *vice versa*. For example, $p \;\Leftarrow\; (p \leftarrow r) \wedge r$ is one of the two cancellation properties, expressed at the point level.

□

The concept of a Galois connection in lattice theory is captured in category theory by the concept of an *adjunction*. We saw several alternative, but equivalent, definitions of a Galois connection. We can likewise give several alternative equivalent definitions of an adjunction.

4.2 The Hom bifunctor

Before turning to the definition of the notion of adjunction, we introduce a concept that has many applications, and that we will use to give an elegant definition of adjunctions.

Given some category $\mathcal{C}$, consider the function mapping a pair of objects $x{\in}\mathcal{C}$ and $y{\in}\mathcal{C}$ to the set[1] $x \overset{\mathcal{C}}{\leftarrow} y$ consisting of the arrows in the category $\mathcal{C}$ to x from y. (The reason why we denote the category explicitly over the arrow here, is that soon we are going to consider several categories simultaneously.) Now a set is an object in the category Fun (which has sets for objects, functions for arrows, and function composition as the composition operator), and so the function $\overset{\mathcal{C}}{\leftarrow}$ maps pairs of objects from $\mathcal{C}$ to objects of Fun. This function can be extended to a binary functor. Sets of arrows between a pair of objects from a category are called hom-sets, and accordingly this bifunctor is known as the *Hom (bi)functor*. It is covariant in its first argument and contravariant in its second argument. Its typing will thus be:

$$(\overset{\mathcal{C}}{\leftarrow}) \in \mathsf{Fun} \leftarrow \mathcal{C}{\times}\mathcal{C}^{op} \quad .$$

To make $\overset{\mathcal{C}}{\leftarrow}$ indeed into a functor we have to define, for an arrow $(f,g) \in (x,y) \overset{\mathcal{C}\times\mathcal{C}^{op}}{\longleftarrow} (u,v)$ (so that $f \in x \overset{\mathcal{C}}{\longleftarrow} u$ and $g \in v \overset{\mathcal{C}}{\longleftarrow} y$), a function result $f \overset{\mathcal{C}}{\leftarrow} g \in x \overset{\mathcal{C}}{\leftarrow} y$. Let us construct a candidate:

$$\begin{array}{ll} & f \overset{\mathcal{C}}{\leftarrow} g \in (x \overset{\mathcal{C}}{\leftarrow} y) \longleftarrow (u \overset{\mathcal{C}}{\leftarrow} v) \\ \Leftarrow & \quad \{ \quad f \overset{\mathcal{C}}{\leftarrow} g \quad := \quad (h:\ h \in u \overset{\mathcal{C}}{\longleftarrow} v:\ k);\ \text{typing in } \mathsf{Fun} \quad \} \\ & k \ \in \ x \overset{\mathcal{C}}{\leftarrow} y \ \Leftarrow \ h \in u \overset{\mathcal{C}}{\longleftarrow} v \\ \Leftarrow & \quad \{ \quad k \ := \ f \circ h \circ g;\ \circ\text{-TYPING} \quad \} \\ & f \in x \overset{\mathcal{C}}{\longleftarrow} u \wedge g \in v \overset{\mathcal{C}}{\longleftarrow} y \quad , \end{array}$$

which gives us:

$$f \overset{\mathcal{C}}{\leftarrow} g \ = \ (h:\ h \in u \overset{\mathcal{C}}{\longleftarrow} v:\ f \circ h \circ g) \quad .$$

It is easily verified that:

$$\mathsf{id}_x \overset{\mathcal{C}}{\leftarrow} \mathsf{id}_y \ = \ \mathsf{id}_{(x \overset{\mathcal{C}}{\leftarrow} y)} \quad ,$$

and for all suitably typed f, g, h and k we have:

[1] For convenience we assume that the category is locally small. That is, the arrows between a pair of objects form a set.

$$
\begin{array}{ll}
 & (f \circ h) \overset{\mathcal{C}}{\leftarrow} (g;k) \\
= & \quad \{ \quad \text{composition of } \mathcal{C}^{op} \quad \} \\
 & (f \circ h) \overset{\mathcal{C}}{\leftarrow} (k \circ g) \\
= & \quad \{ \quad \text{candidate for } \overset{\mathcal{C}}{\leftarrow} \quad \} \\
 & (q :: \ f \circ h \circ q \circ k \circ g) \\
= & \quad \{ \quad \text{Lambda-lambda-fusion} \quad \} \\
 & (p :: \ f \circ p \circ g) \circ (q :: \ h \circ q \circ k) \\
= & \quad \{ \quad \text{candidate for } \overset{\mathcal{C}}{\leftarrow} \quad \} \\
 & (f \overset{\mathcal{C}}{\leftarrow} g) \circ (h \overset{\mathcal{C}}{\leftarrow} k) \ .
\end{array}
$$

Thus we have indeed defined a functor.

We will now use the Hom bifunctor to show that a slight generalisation is also a functor. Let $F \in \mathcal{C} \leftarrow \mathcal{D}$ and $G \in \mathcal{C} \leftarrow \mathcal{E}$ be two functors to the same category $\mathcal{C}$. Consider the function $(x, y :: \ Fx \overset{\mathcal{C}}{\leftarrow} Gy \ )$, which can be applied both to an object (x, y) from the category $\mathcal{D} \times \mathcal{E}^{op}$ and to a suitably typed arrow (f, g) from that category. This is again a bifunctor, now with typing $\mathsf{Fun} \leftarrow \mathcal{D} \times \mathcal{E}^{op}$, since it can be rewritten as the composition of two functors, $(\overset{\mathcal{C}}{\leftarrow}) \in \mathsf{Fun} \leftarrow \mathcal{C} \times \mathcal{C}^{op}$ and $F \times G \in \mathcal{C} \times \mathcal{C}^{op} \leftarrow \mathcal{D} \times \mathcal{E}^{op}$, as follows:

$$(x, y :: \ Fx \overset{\mathcal{C}}{\leftarrow} Gy \ ) = (\overset{\mathcal{C}}{\leftarrow})(F \times G) \ .$$

If F or G are identity functors, we can specialise the notation for this bifunctor to, respectively, $(x, y :: \ x \overset{\mathcal{C}}{\leftarrow} Gy \ )$ or $(x, y :: \ Fx \overset{\mathcal{C}}{\leftarrow} y \ )$.

4.3 Definition of adjunction

The category-theoretic concept corresponding to the notion of a Galois connection is that of an *adjunction*. There is a large number of equivalent definitions of an adjunction. The one we give here is chosen as being the simplest and easiest to remember.

Recall that the categorical generalisation of a monotonic function is a functor. Suppose, as at the end of the previous section, that F and G are functors, but this time with typing $\mathcal{C} \leftarrow \mathcal{D}$ and $\mathcal{D} \leftarrow \mathcal{C}$, respectively, so they go in each of the two directions between two categories $\mathcal{C}$ and $\mathcal{D}$. Then the two functors $(x, y :: \ x \overset{\mathcal{C}}{\leftarrow} Fy)$ and $(x, y :: \ Gx \overset{\mathcal{D}}{\leftarrow} y)$ have the *same* typing, namely $\mathsf{Fun} \longleftarrow \mathcal{C} \times \mathcal{D}^{op}$. If they are isomorphic then F and G are said to be *adjoint functors*. So we have the following definition.

Definition 4.8 (Adjunction) A pair of functors $F \in \mathcal{C} \leftarrow \mathcal{D}$ and $G \in \mathcal{D} \leftarrow \mathcal{C}$ forms an adjunction between the categories $\mathcal{C}$ and $\mathcal{D}$ if: the following functor isomorphy holds:

$$(x, y :: \ x \overset{\mathcal{C}}{\leftarrow} Fy) \cong (x, y :: \ Gx \overset{\mathcal{D}}{\leftarrow} y) \ .$$

The functor F is called the *lower adjoint* and functor G is called the *upper adjoint*.

□

(Compare this to Definition 4.3.)

Remark 4.9 In the literature one also finds the term *left adjoint* for F and *right adjoint* for G.

□

Remark 4.10 It is immediate from the definition that, whenever (F, G) forms an adjunction between $\mathcal{C}$ and $\mathcal{D}$, dually (G, F) forms an adjunction between $\mathcal{D}^{op}$ and $\mathcal{C}^{op}$.

□

4.4 Properties of adjunctions

Let us spell out some of the consequences of Definition 4.8. Throughout the remainder of this section we assume that F and G are functors with typing $F \in \mathcal{C} \leftarrow \mathcal{D}$ and $G \in \mathcal{D} \leftarrow \mathcal{C}$. Expanding the definition of a natural isomorphism, we get that there are two natural transformations $\lceil\_\rceil$ and $\lfloor\_\rfloor$ such that

$$\lceil\_\rceil \in (x, y :: \ x \overset{\mathcal{C}}{\leftarrow} Fy) \dot{\leftarrow} (x, y :: \ Gx \overset{\mathcal{D}}{\leftarrow} y) \quad , \tag{4.11}$$

$$\lfloor\_\rfloor \in (x, y :: \ Gx \overset{\mathcal{D}}{\leftarrow} y) \dot{\leftarrow} (x, y :: \ x \overset{\mathcal{C}}{\leftarrow} Fy) \quad , \tag{4.12}$$

which are each others' inverse, i.e.,

$$\lceil \lfloor\_\rfloor \rceil = \mathsf{id} \wedge \lfloor \lceil\_\rceil \rfloor = \mathsf{id} \quad . \tag{4.13}$$

The natural transformations $\lceil\_\rceil$ and $\lfloor\_\rfloor$ are called the *upper adjungate* and the *lower adjungate*, respectively. Expanding next the definition of a natural transformation we get from (4.11): for each $x \in \mathcal{C}$ and $y \in \mathcal{D}$:

$$\lceil\_\rceil_{x,y} \in x \overset{\mathcal{C}}{\leftarrow} Fy \leftarrow Gx \overset{\mathcal{D}}{\leftarrow} y$$

and for each$(f, g) \in (x, y) \leftarrow (u, v)$ (an arrow in $\mathcal{C} \times \mathcal{D}^{op}$):

$$(f \overset{\mathcal{C}}{\leftarrow} Fg) \circ \lceil\_\rceil_{u,v} = \lceil\_\rceil_{x,y} \circ (Gf \overset{\mathcal{D}}{\leftarrow} g) \quad .$$

A similar result can be derived from (4.12).

We will usually omit the subscripts of $\lceil\_\rceil$ and $\lfloor\_\rfloor$ when they can be inferred from the typings of the context.

Now, we can use extensionality and the definition of the functor $(x, y :: \ Fx \overset{\mathcal{C}}{\leftarrow} Gy)$ from the previous section—by which $(Ff \overset{\mathcal{C}}{\leftarrow} Gg)h \ = \ Ff \circ h \circ Gg$ (for suitably typed f, g and h)—to expand this as well as (4.13) further, thus obtaining an alternative presentation of the definition of an adjunction.

Theorem 4.14 F and G form an adjunction if: there exist two transformations $\lceil\text{-}\rceil$ and $\lfloor\text{-}\rfloor$ satisfying for all $x{\in}\mathcal{C}$ and $y{\in}\mathcal{D}$, first, the typing requirements

(a) $$\lceil g \rceil \in x \xleftarrow{\mathcal{C}} Fy \Leftarrow g \in Gx \xleftarrow{\mathcal{D}} y \quad , \qquad \lceil\text{-}\rceil\text{-TYPING}$$

(b) $$\lfloor f \rfloor \in Gx \xleftarrow{\mathcal{D}} y \Leftarrow f \in x \xleftarrow{\mathcal{C}} Fy \quad ; \qquad \lfloor\text{-}\rfloor\text{-TYPING}$$

furthermore, for each $f \in x \xleftarrow{\mathcal{C}} u$ and $g \in v \xleftarrow{\mathcal{D}} y$, the following naturality equalities:
for each $h \in Gu \xleftarrow{\mathcal{D}} v$:

(c) $$f \circ \lceil h \rceil \circ Fg \;=\; \lceil Gf \circ h \circ g \rceil \quad ; \qquad \lceil\text{-}\rceil\text{-FUSION}$$

for each $h \in u \xleftarrow{\mathcal{C}} Fv$:

(d) $$Gf \circ \lfloor h \rfloor \circ g \;=\; \lfloor f \circ h \circ Fg \rfloor \quad ; \qquad \lfloor\text{-}\rfloor\text{-FUSION}$$

and, finally, the inverse property, expressed as the equivalence:
for each $f \in x \xleftarrow{\mathcal{C}} Fy$ and $g \in Gx \xleftarrow{\mathcal{D}} y$:

(e) $$\lceil g \rceil = f \;\equiv\; g = \lfloor f \rfloor \quad . \qquad \text{INVERSE}$$

□

Actually, either one of $\lceil\text{-}\rceil$-FUSION and $\lfloor\text{-}\rfloor$-FUSION suffices, since the other one follows from it by INVERSE. Furthermore, the FUSION laws above are two-sided, but each can be decomposed into an equivalent pair of simpler one-sided FUSION laws (where f,g and h are supposed to range over all suitably typed arrows):

$$f{\circ}\lceil h \rceil = \lceil Gf \circ h \rceil \quad ; \qquad \lceil\text{-}\rceil\text{-LEFT-FUSION}$$

$$\lceil h \rceil \circ Fg \;=\; \lceil h{\circ}g \rceil \quad ; \qquad \lceil\text{-}\rceil\text{-RIGHT-FUSION}$$

$$Gf \circ \lfloor h \rfloor \;=\; \lfloor f{\circ}h \rfloor \quad ; \qquad \lfloor\text{-}\rfloor\text{-LEFT-FUSION}$$

$$\lfloor h \rfloor{\circ}g = \lfloor h \circ Fg \rfloor \quad . \qquad \lfloor\text{-}\rfloor\text{-RIGHT-FUSION}$$

Under the assumption of INVERSE, the two LEFT-FUSION laws are equivalent, and so are the two RIGHT-FUSION laws. Thus we have as yet another definition that is slightly easier to verify:

Theorem 4.15 F and G form an adjunction if: we have two transformations $\lceil\text{-}\rceil$ and $\lfloor\text{-}\rfloor$ satisfying the requirements $\lceil\text{-}\rceil$-TYPING, $\lfloor\text{-}\rfloor$-TYPING and INVERSE as in Theorem 4.14, and moreover one of the two LEFT-FUSION laws above, as well as one of the two RIGHT-FUSION laws.

□

The following result shows that the arrow part of the functor F can be defined in terms of the adjungates, thereby preparing the way for Theorem 4.17:

Corollary 4.16 $Fg = \lceil \lfloor \mathsf{id} \rfloor \circ g \rceil$.

Proof

$$
\begin{array}{ll}
 & Fg \\
= & \quad \{ \quad \lceil\text{-}\rceil \text{ and } \lfloor\text{-}\rfloor \text{ are each other's inverse} \quad \} \\
 & \lceil \lfloor \mathsf{id} \rfloor \rceil \circ Fg \\
= & \quad \{ \quad \lceil\text{-}\rceil\text{-RIGHT-FUSION} \quad \} \\
 & \lceil \lfloor \mathsf{id} \rfloor \circ g \rceil \quad .
\end{array}
$$

□

The interesting thing is now that if we define F on arrows this way, we get functoriality and RIGHT-FUSION for free:

Theorem 4.17 Given an object mapping F to $\mathcal{C}$ from $\mathcal{D}$, a functor G in the other direction, and two transformations $\lceil\text{-}\rceil$ and $\lfloor\text{-}\rfloor$ satisfying the requirements $\lceil\text{-}\rceil$-TYPING, $\lfloor\text{-}\rfloor$-TYPING and INVERSE, and moreover one of the two LEFT-FUSION laws (as in Theorem 4.15), the object mapping F can (by Corollary 4.16 in a unique way) be extended to a functor $F \in \mathcal{C} \leftarrow \mathcal{D}$ such that (F, G) forms an adjunction.

Proof On arrows of $\mathcal{D}$ we define F by $Fg = \lceil \lfloor \mathsf{id} \rfloor \circ g \rceil$. Then, for the functorial typing requirements, we argue:

$$
\begin{array}{ll}
 & Fg \in Fx \xleftarrow{\mathcal{C}} Fy \\
\equiv & \quad \{ \quad \text{definition of } F \text{ on arrows} \quad \} \\
 & \lceil \lfloor \mathsf{id} \rfloor \circ g \rceil \in Fx \xleftarrow{\mathcal{C}} Fy \\
\Leftarrow & \quad \{ \quad \lceil\text{-}\rceil\text{-TYPING} \quad \} \\
 & \lfloor \mathsf{id} \rfloor \circ g \in GFx \xleftarrow{\mathcal{D}} y \\
\equiv & \quad \{ \quad \lfloor \mathsf{id} \rfloor \in GFx \xleftarrow{\mathcal{D}} x \text{ (see below)} \quad \} \\
 & g \in x \xleftarrow{\mathcal{D}} y \quad .
\end{array}
$$

In the last step above we used:

$$
\begin{array}{ll}
 & \lfloor \mathsf{id} \rfloor \in GFx \xleftarrow{\mathcal{D}} x \\
\Leftarrow & \quad \{ \quad \lfloor \_ \rfloor\text{-TYPING} \quad \} \\
 & \mathsf{id} \in Fx \xleftarrow{\mathcal{C}} Fx \quad .
\end{array}
$$

For the distribution over composition, we calculate:

$$
\begin{array}{ll}
 & Ff \circ Fg \;=\; F(f \circ g) \\
\equiv & \quad \{ \quad \text{definition of } F \text{ on arrows} \quad \} \\
 & \lceil \lfloor \mathsf{id} \rfloor \circ f \rceil \circ \lceil \lfloor \mathsf{id} \rfloor \circ g \rceil \;=\; \lceil \lfloor \mathsf{id} \rfloor \circ f \circ g \rceil \\
\equiv & \quad \{ \quad \lceil \_ \rceil\text{-LEFT-FUSION} \quad \} \\
 & \lceil G \lceil \lfloor \mathsf{id} \rfloor \circ f \rceil \circ \lfloor \mathsf{id} \rfloor \circ g \rceil = \lceil \lfloor \mathsf{id} \rfloor \circ f \circ g \rceil \\
\equiv & \quad \{ \quad \text{INVERSE} \quad \} \\
 & G \lceil \lfloor \mathsf{id} \rfloor \circ f \rceil \circ \lfloor \mathsf{id} \rfloor \circ g \;=\; \lfloor \mathsf{id} \rfloor \circ f \circ g \\
\Leftarrow & \quad \{ \quad \text{LEIBNIZ} \quad \} \\
 & G \lceil \lfloor \mathsf{id} \rfloor \circ f \rceil \circ \lfloor \mathsf{id} \rfloor \;=\; \lfloor \mathsf{id} \rfloor \circ f \\
\equiv & \quad \{ \quad \lfloor \_ \rfloor\text{-LEFT-FUSION} \quad \} \\
 & \lfloor \lceil \lfloor \mathsf{id} \rfloor \circ f \rceil \rfloor \;=\; \lfloor \mathsf{id} \rfloor \circ f \\
\equiv & \quad \{ \quad \text{INVERSE} \quad \} \\
 & \lceil \lfloor \mathsf{id} \rfloor \circ f \rceil = \lceil \lfloor \mathsf{id} \rfloor \circ f \rceil \\
\equiv & \quad \{ \quad \text{reflexivity of} = \quad \} \\
 & \mathsf{true} \quad ,
\end{array}
$$

and similarly $F\mathsf{id} = \mathsf{id}$.

Finally, for $\lceil \_ \rceil$-RIGHT-FUSION, we calculate:

$$
\begin{array}{ll}
 & \lceil h \rceil \circ Fg \\
= & \quad \{ \quad \text{definition of } F \text{ on arrows} \quad \} \\
 & \lceil h \rceil \circ \lceil \lfloor \mathsf{id} \rfloor \circ g \rceil \\
= & \quad \{ \quad \lceil \_ \rceil\text{-LEFT-FUSION} \quad \} \\
 & \lceil G \lceil h \rceil \circ \lfloor \mathsf{id} \rfloor \circ g \rceil \\
= & \quad \{ \quad \lfloor \_ \rfloor\text{-LEFT-FUSION} \quad \} \\
 & \lceil \lfloor \lceil h \rceil \rfloor \circ g \rceil
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \quad \lceil\text{-}\rceil \text{ and } \lfloor\text{-}\rfloor \text{ are each other's inverse} \quad \} \\
& \lceil h \circ g \rceil
\end{array}
$$

Now all conditions for applying Theorem 4.15 are satisfied.
□

Elementary consequences of a Galois connection are the cancellation properties. For an adjunction a similar result can be derived. One of the two cancellation properties for Galois connections is (4.4): $G \circ F \mathrel{\dot{\sqsupseteq}}_D I$. The categorical generalisation is: there exists a natural transformation

$$\mathsf{unit} \in GF \dot{\leftarrow} \mathsf{Id}_{\mathcal{D}} \quad .$$

We construct such a natural transformation by calculation as follows:

$$
\begin{array}{ll}
& \mathsf{unit} \in GF \dot{\leftarrow} \mathsf{Id}_{\mathcal{D}} \\
\equiv & \quad \{ \quad \text{definition natural transformation} \quad \} \\
& \forall (x :: \mathsf{unit}_x \in GFx \leftarrow x) \wedge \forall (f : f \in x \leftarrow y : GFf \circ \mathsf{unit}_y \;=\; \mathsf{unit}_x \circ f) \\
\Leftarrow & \quad \{ \quad \mathsf{unit}_x \;:=\; \lfloor \mathsf{id}_{Fx} \rfloor \quad \} \\
& \forall (x :: \lfloor \mathsf{id}_{Fx} \rfloor \in GFx \leftarrow x) \wedge \forall (f : f \in x \leftarrow y : GFf \circ \lfloor \mathsf{id}_{Fy} \rfloor \;=\; \lfloor \mathsf{id}_{Fx} \rfloor \circ f) \\
\equiv & \quad \{ \quad \lfloor\text{-}\rfloor\text{-TYPING}, \lfloor\text{-}\rfloor\text{-FUSION} \quad \} \\
& \mathsf{true} \quad .
\end{array}
$$

This natural transformation unit is commonly known as the *unit* of the adjunction. So, an immediate consequence of an adjunction is that we have a natural transformation unit, such that

$$\mathsf{unit} \in GF \dot{\leftarrow} \mathsf{Id}_{\mathcal{D}} \text{ , where } \mathsf{unit}_x = \lfloor \mathsf{id}_{Fx} \rfloor \text{ for all } x. \qquad \mathsf{unit}\text{-DEF}$$

Dually, we also have a natural transformation counit, known as the *co-unit* of the adjunction, such that

$$\mathsf{counit} \in \mathsf{Id}_{\mathcal{C}} \dot{\leftarrow} FG \text{ , where } \mathsf{counit}_x = \lceil \mathsf{id}_{Gx} \rceil \text{ for all } x. \qquad \mathsf{counit}\text{-DEF}$$

These two natural transformations correspond to the cancellation properties for Galois connections, and give likewise rise to an alternative definition of adjunctions.

Theorem 4.18 F and G form an adjunction if we have two natural transformations unit and counit such that

$$\mathsf{unit} \in GF \dot{\leftarrow} \mathsf{Id}_{\mathcal{D}} \quad , \qquad \mathsf{unit}\text{-TYPING}$$

$$\mathsf{counit} \in \mathsf{Id}_{\mathcal{C}} \dot{\leftarrow} FG \quad , \qquad \mathsf{counit}\text{-TYPING}$$

and satisfying the following two coherence conditions:

$$\mathsf{counit}_F \circ F\mathsf{unit} \;=\; \mathsf{id}_F \quad , \qquad \text{counit-INVERSE}$$

$$G\mathsf{counit} \circ \mathsf{unit}_G \;=\; \mathsf{id}_G \quad . \qquad \text{unit-INVERSE}$$

Proof We use Theorem 4.14, and have to construct a lower and upper adjungate. For $f \in x \xleftarrow{\mathcal{C}} Fy$ we construct a candidate for the lower adjungate (aiming at $\lfloor\text{-}\rfloor$-TYPING) as follows:

$$\begin{array}{ll} & \lfloor f \rfloor \in Gx \xleftarrow{\mathcal{D}} y \\ \Leftarrow & \qquad \{ \quad \lfloor f \rfloor \;:=\; h \circ \mathsf{unit}_y;\ \circ\text{-TYPING} \quad \} \\ & h \in Gx \xleftarrow{\mathcal{D}} GFy \\ \Leftarrow & \qquad \{ \quad h \;:=\; Gf;\ G \text{ is a functor} \quad \} \\ & f \in x \xleftarrow{\mathcal{C}} Fy \quad . \end{array}$$

Thus, we put

$$\lfloor f \rfloor \;=\; Gf \circ \mathsf{unit} \quad ,$$

and, dually, for $g \in Gx \xleftarrow{\mathcal{D}} y$ we put

$$\lceil g \rceil \;=\; \mathsf{counit} \circ Fg \quad .$$

We only verify $\lfloor\text{-}\rfloor$-FUSION, the other fusion requirement being dual.

$$\begin{array}{ll} & Gg \circ \lfloor f \rfloor \circ h \\ = & \qquad \{ \quad \text{definition of candidate for } \lfloor\text{-}\rfloor \quad \} \\ & Gg \circ (Gf \circ \mathsf{unit}) \circ h \\ = & \qquad \{ \quad \mathsf{unit} \in GF \dot{\leftarrow} \mathsf{Id} \quad \} \\ & Gg \circ Gf \circ GFh \circ \mathsf{unit} \\ = & \qquad \{ \quad G \text{ is a functor} \quad \} \\ & G(g \circ f \circ Fh) \circ \mathsf{unit} \\ = & \qquad \{ \quad \text{our candidate for } \lfloor\text{-}\rfloor \quad \} \\ & \lfloor g \circ f \circ Fh \rfloor \quad . \end{array}$$

For INVERSE we only verify the $\Leftarrow$-direction, the other being dual.

$$
\begin{array}{ll}
 & \lceil g \rceil = f \\
\equiv & \quad \{ \quad \text{candidate for } \lceil \text{-} \rceil \quad \} \\
 & \mathsf{counit} \circ Fg \;=\; f \\
\equiv & \quad \{ \quad \mathsf{counit}_F \circ F\mathsf{unit} \;=\; \mathsf{id}_F \quad \} \\
 & \mathsf{counit} \circ Fg \;=\; f \circ \mathsf{counit} \circ F\mathsf{unit} \\
\equiv & \quad \{ \quad \mathsf{counit} \in \mathsf{Id} \dot{\leftarrow} FG \quad \} \\
 & \mathsf{counit} \circ Fg \;=\; \mathsf{counit} \circ FGf \circ F\mathsf{unit} \\
\equiv & \quad \{ \quad F \text{ is a functor} \quad \} \\
 & \mathsf{counit} \circ Fg \;=\; \mathsf{counit} \circ F(Gf \circ \mathsf{unit}) \\
\Leftarrow & \quad \{ \quad \textsc{Leibniz} \quad \} \\
 & g \;=\; Gf \circ \mathsf{unit} \\
\equiv & \quad \{ \quad \text{candidate for } \lfloor \text{-} \rfloor \quad \} \\
 & g = \lfloor f \rfloor
\end{array}
$$

□

We have shown that, given a lower and upper adjungate, we can construct a unit and co-unit, and vice versa. An adjunction does not determine the lower and upper adjungate uniquely, nor the unit and co-unit. But the above constructions show that we can always choose the lower and upper adjungate and the unit and co-unit together in such a way that the following properties hold for all suitably typed arrows f and g:

$$\lfloor f \rfloor \;=\; Gf \circ \mathsf{unit} \text{ for } f \in x \xleftarrow{\mathcal{C}} Fy \;\;, \qquad \lfloor\text{-}\rfloor\text{-DEF}$$

$$\lceil g \rceil \;=\; \mathsf{counit} \circ Fg \text{ for } g \in Gx \xleftarrow{\mathcal{D}} y \;\;. \qquad \lceil\text{-}\rceil\text{-DEF}$$

We'll do this in the rest of this chapter so that we can always use this property.

In the above proof a valuable property is proved. Namely, for all suitably typed arrows f and g:

$$Gf \circ \mathsf{unit} \;=\; g \;\equiv\; f \;=\; \mathsf{counit} \circ Fg \;\;. \qquad \mathsf{unit\text{-}counit\text{-}INVERSE}$$

This property is an alternative for the two coherence conditions as stated in the above theorem. That is, the two coherence conditions give rise to this property as shown in the last proof, while conversely, by instantiating f,g := counit,id and f,g := id,unit in unit-counit-INVERSE we obtain counit-INVERSE and unit-INVERSE. So we have as a last alternative definition of an adjunction:

Theorem 4.19 F and G form an adjunction if: there exist two natural transformations unit and counit such that unit-TYPING and counit-TYPING as in Theorem 4.18 hold, as well as unit-counit-INVERSE.

□

We rephrase Corollary 4.16 using $\mathsf{unit}{=}\lfloor\mathsf{id}\rfloor$ and add its dual:

Theorem 4.20 If (F, G) forms an adjunction,

$$Fg = \lceil \mathsf{unit} \circ g \rceil \quad , \qquad \text{LOWER-ADJOINT-DEF}$$

$$Gf = \lfloor f \circ \mathsf{counit} \rfloor \quad , \qquad \text{UPPER-ADJOINT-DEF}$$

□

To conclude this section, we show how adjunctions can be composed:

Theorem 4.21 If (F, G) forms an adjunction between categories $\mathcal{C}$ and $\mathcal{D}$, and (H, K) forms an adjunction between categories $\mathcal{D}$ and $\mathcal{E}$, then: (FH, KG) forms an adjunction between $\mathcal{C}$ and $\mathcal{E}$.

□

Exercise 4.22 Show that for two isomorphic categories the witnesses of the isomorphy form an adjunction. Compare the witnesses of the transitivity of $\cong$ (see Exercise 2.3) to the statement of Theorem 4.21.

□

Exercise 4.23 Prove Theorem 4.21. Construct the unit and co-unit of the composite adjunction.

□

4.5 Examples of adjunctions

4.5.1 Rel and Fun

We construct an adjunction between the categories Rel and Fun. Note that they have the same objects. There is a well-known one-to-one correspondence between set-valued functions and relations, and the translations forth and back will be the adjungates.

Let, for a set x, $\exists x$ denote the powerset of x, so

$$\exists x = \$(s :: s \subseteq x) \quad .$$

To turn a relation into a set-valued function, we can use:

$$\lfloor R \rfloor = (v :: \$(u :: u\ R\ v)) \in \exists x \xleftarrow{\mathsf{Fun}} y \Leftarrow R \in x \xleftarrow{\mathsf{Rel}} y$$

(so, at the point level, $u \in \lfloor R \rfloor.v \ \equiv \ u\ R\ v$). To go the other way we construct its inverse:

$$\begin{array}{ll} & \lceil f \rceil \\ = & \quad \{ \quad \textsc{Rel-abstraction} \quad \} \\ & \$(u, v :: u\ \lceil f \rceil\ v) \\ = & \quad \{ \quad \text{definition of } \lfloor R \rfloor \text{ at the point level} \quad \} \\ & \$(u, v :: u \in \lfloor \lceil f \rceil \rfloor.v) \\ = & \quad \{ \quad \text{inverse} \quad \} \\ & \$(u, v :: u \in f.v) \end{array}$$

(so $u\ \lceil f \rceil\ v \ \equiv \ u \in f.v$). It is easily checked that the typing is as required:

$$\lceil f \rceil = \$(u, v :: u \in f.v) \in x \xleftarrow{\mathsf{Rel}} y \Leftarrow f \in \exists x \xleftarrow{\mathsf{Fun}} y \quad .$$

The unexciting verification that these candidate adjungates are each others' inverse also in the other direction is omitted.

To make this fit the pattern of adjunctions, we have to extend $\exists$ to a functor to Fun from Rel and to find some functor $\Re \in \mathsf{Rel} \leftarrow \mathsf{Fun}$ that on objects is (semantically) the identity function. We can use Theorem 4.20 for finding candidates. To apply that theorem we first compute candidates for the unit and counit:

$$\begin{array}{ll} & \mathsf{unit} \\ = & \quad \{ \quad \mathsf{unit}\text{-}\textsc{def} \quad \} \\ & \lfloor \mathsf{id} \rfloor \\ = & \quad \{ \quad \text{candidate for } \lfloor \_ \rfloor \quad \} \end{array}$$

$$
\begin{array}{ll}
 & (v :: \$(u :: u\ (\mathsf{id})\ v)) \\
= & \quad \{ \quad \text{definition of } \mathsf{id} \text{ in } \mathsf{Rel} \quad \} \\
 & (v :: \$(u :: u = v)) \\
= & \quad \{ \quad \text{membership of singleton set} \quad \} \\
 & (v :: \$(u :: u{\in}\{v\})) \\
= & \quad \{ \quad \textsc{Compr-abstraction} \quad \} \\
 & (v :: \{v\}) \\
= & \quad \{ \quad \textsc{Lambda-abstraction} \quad \} \\
 & \{\text{-}\} \quad .
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{counit} \\
= & \quad \{ \quad \mathsf{counit}\text{-}\textsc{def} \quad \} \\
 & \lceil \mathsf{id} \rceil \\
= & \quad \{ \quad \text{candidate for } \lceil \text{-} \rceil \quad \} \\
 & \$(u, v :: u \in v) \\
= & \quad \{ \quad \textsc{Rel-abstraction} \quad \} \\
 & (\in) \quad .
\end{array}
$$

To obtain a candidate for the arrow part of $\Re$ we compute:

$$
\begin{array}{ll}
 & \Re f \\
= & \quad \{ \quad \textsc{Lower-adjoint-def} \quad \} \\
 & \lceil \mathsf{unit} \circ f \rceil \\
= & \quad \{ \quad \text{candidates for } \lceil \text{-} \rceil \text{ and } \mathsf{unit} \quad \} \\
 & \$(u, v :: u \in (\{\text{-}\} \circ f).v) \\
= & \quad \{ \quad \textsc{Fun-}\circ\textsc{-comp} \quad \} \\
 & \$(u, v :: u{\in}\{f.v\}) \\
= & \quad \{ \quad \text{membership of singleton set} \quad \} \\
 & \$(u, v :: u = f.v) \quad .
\end{array}
$$

Expressed at the point level this amounts to

$$u\ (\Re f)\ v \ \equiv \ u = f.v \quad .$$

This is the standard embedding of functions into relations, which preserves identities and distributes over composition, so we have a functor, which is known as the *graph* functor.

As to $\exists$:

$$
\begin{array}{ll}
 & \exists R \\
= & \quad \{ \quad \textsc{Upper-adjoint-def} \quad \} \\
 & \lfloor R \circ \mathsf{counit} \rfloor \\
= & \quad \{ \quad \text{candidates for } \lfloor \_ \rfloor \text{ and } \mathsf{counit} \quad \} \\
 & (v :: \$(u :: u \ (R \circ (\in)) \ v)) \\
= & \quad \{ \quad \textsc{Rel-}\circ\textsc{-comp} \quad \} \\
 & (v :: \$(u :: \exists(w :: u \ R \ w \wedge w \in v))) \quad .
\end{array}
$$

This amounts to:

$$u \in (\exists R).v \ \equiv \ \exists(w :: u \ R \ w \wedge w \in v) \quad .$$

We can save ourselves the task of verifying that $\exists$ is a functor (known as the *existential image* functor) by applying Theorem 4.17. We already have all ingredients needed for that except for one of the LEFT-FUSION laws. We show that $\lfloor \_ \rfloor$-LEFT-FUSION holds:

$$
\begin{array}{ll}
 & \exists R \circ \lfloor S \rfloor \\
= & \quad \{ \quad \text{candidate for } \exists R \quad \} \\
 & (v :: \$(u :: \exists(w :: u \ R \ w \wedge w \in v))) \circ \lfloor S \rfloor \\
= & \quad \{ \quad \textsc{Lambda-right-fusion} \quad \} \\
 & (v :: \$(u :: \exists(w :: u \ R \ w \ \wedge \ w \in \lfloor S \rfloor .v))) \\
= & \quad \{ \quad \text{candidate for } \lfloor \_ \rfloor \text{ (recall that } u \in \lfloor R \rfloor .v \ \equiv \ u \ R \ v) \quad \} \\
 & (v :: \$(u :: \exists(w :: u \ R \ w \wedge w \ S \ v))) \\
= & \quad \{ \quad \textsc{Rel-}\circ\textsc{-comp} \quad \} \\
 & (v :: \$(u :: u \ (R \circ S) \ v)) \\
= & \quad \{ \quad \text{candidate for } \lfloor \_ \rfloor \quad \} \\
 & \lfloor R \circ S \rfloor \quad .
\end{array}
$$

4.5.2 Sum as adjoint functor

We saw before that sum is the categorical generalisation of the lattice-theoretic concept of join. Join is characterised by a Galois connection. It will then, perhaps, not be a surprise

that sum can be likewise characterised by an adjunction. In categories with sums, the bifunctor $+$ is the lower adjoint of the doubling functor Δ, where $\Delta \in \mathcal{C}\times\mathcal{C} \leftarrow \mathcal{C}$ is defined by:

$$\Delta x = (x, x) \quad \text{for } x \in \mathcal{C} \quad ,$$

$$\Delta f = (f, f) \quad \text{for } f \in x \xleftarrow{\mathcal{C}} y \quad .$$

The trivial verification that Δ is a functor is omitted.

Let us pretend that we do not know the definition of categorical sum, but instead work out the claim that Δ has *some* lower adjoint, which is required to have the typing $\mathcal{C} \leftarrow \mathcal{C}\times\mathcal{C}$ and so is some bifunctor $\oplus$, and verify the conditions needed for applying Theorem 4.17.

The typing requirements of $\lceil\_\rceil$, after unfolding the definition of Δ, amount to:

$$\lceil(f, g)\rceil \in z \xleftarrow{\mathcal{C}} x \oplus y \Leftarrow (f, g) \in (z, z) \xleftarrow{\mathcal{C}\times\mathcal{C}} (x, y) \quad .$$

We use the more pleasant notation $f \heartsuit g$ for $\lceil(f, g)\rceil$. Further, we express the arrows in $\mathcal{C}\times\mathcal{C}$ as two arrows in $\mathcal{C}$, and get:

$$f \heartsuit g \in z \leftarrow x \oplus y \;\Leftarrow\; f \in z \leftarrow x \wedge g \in z \leftarrow y \quad .$$

As all arrows here are arrows of $\mathcal{C}$, the superscript has been omitted.

Instead of looking directly at $\lfloor\_\rfloor$, we take a look at unit, which, by virtue of unit-DEF and $\lfloor\_\rfloor$-DEF, is interexpressible with $\lfloor\_\rfloor$:

$$\mathsf{unit} \in (x \oplus y, x \oplus y) \xleftarrow{\mathcal{C}\times\mathcal{C}} (x, y) \quad .$$

We see that unit is some pair of arrows $(\mathsf{unl}, \mathsf{unr})$, where

$$\mathsf{unl} \in x \oplus y \leftarrow x \quad \text{and} \quad \mathsf{unr} \in x \oplus y \leftarrow y \quad .$$

Using $\lfloor\_\rfloor$-DEF we then obtain, for $h \in z \leftarrow x \oplus y$:

$$\lfloor h \rfloor = (h \circ \mathsf{unl}, h \circ \mathsf{unr}) \in \Delta x \xleftarrow{\mathcal{C}\times\mathcal{C}} (x, y) \quad .$$

We can now formulate the further requirements.

For INVERSE we obtain:

$$f \heartsuit g = h \;\equiv\; f = h \circ \mathsf{unl} \;\wedge\; g = h \circ \mathsf{unr} \quad .$$

This is (but for using the symbols $\heartsuit$, unl and unr instead of $\triangledown$, inl and inr) precisely $\triangledown$-CHAR, the characterisation for categorical sum, and so $\oplus$ has to be isomorphic to $+$!

We still have to check $\lceil\_\rceil$-LEFT-FUSION, though, which amounts to:

$$h \circ f \heartsuit g \;=\; (h \circ f) \heartsuit (h \circ g) \quad .$$

We already know this to be valid, given $\triangledown$-CHAR: it is $\triangledown$-FUSION.

Concluding: given the notion of adjunction, categorical sum can be defined rather concisely by the following statement:

$+$ is the lower adjoint of Δ, with lower adjungate $\triangledown$ and unit $(\mathsf{inl}, \mathsf{inr})$.

Categorical product can dually be defined as the upper adjoint of Δ.

4.6 Exponents

Definition 4.24 (Has exponents) Assume that $\mathcal{C}$ is a category with products. Then we say that $\mathcal{C}$ *has exponents* if: for each $z{\in}\mathcal{C}$ the functor $\times z$ has an upper adjoint.

□

We work out the consequences of this definition. Assume that $\mathcal{C}$ has exponents. Since $\times z$ is an endofunctor, the adjunction is apparently between $\mathcal{C}$ and itself, and so the upper adjoint of $\times z$ is also an endofunctor. It will in general depend on z. Denote the application of 'the' upper adjoint of $\times z$ to x by $x{\leftharpoonup}z$, so that $(\times z, {\leftharpoonup}z)$ forms an adjunction. Anticipating the result of Section 5.1, we denote the application of ${\leftharpoonup}z$ to an arrow f as $f{\leftharpoonup}\mathsf{id}_z$. We give names to the adjungates, calling the lower adjungate curry and the upper adjungate uncurry. Further, we call the unit pair and the counit eval. The following are all consequences, obtained by substituting these names in the rules for adjunctions (where, again, we drop most of the subscripts):

$$\mathsf{uncurry}.g \in x \leftarrow y{\times}z \Leftarrow g \in (x{\leftharpoonup}z) \leftarrow y \quad ;$$

$$\mathsf{curry}.f \in (x{\leftharpoonup}z) \leftarrow y \Leftarrow f \in x \leftarrow y{\times}z \quad ;$$

$$\mathsf{uncurry}.g = f \;\equiv\; \mathsf{curry}.f = g \quad ;$$

$$\mathsf{pair} \in (x :: (x{\times}z){\leftharpoonup}z){\dot{\leftarrow}}\mathsf{Id} \quad ;$$

$$\mathsf{pair}_{x,z} = \mathsf{curry}.\mathsf{id}_{x\times z} \quad ;$$

$$\mathsf{eval} \in \mathsf{Id}{\dot{\leftarrow}}(x :: (x{\leftharpoonup}z){\times}z) \quad ;$$

$$\mathsf{eval}_{x,z} = \mathsf{uncurry}.\mathsf{id}_{x\leftharpoonup z} \quad ;$$

$$\mathsf{uncurry}.g \;=\; \mathsf{eval} \circ g{\times}\mathsf{id} \quad ;$$

$$\mathsf{curry}.f \;=\; f{\leftharpoonup}\mathsf{id} \circ \mathsf{pair} \quad ;$$

$$f \circ \mathsf{uncurry}.h \;=\; \mathsf{uncurry}.(f{\leftharpoonup}\mathsf{id} \circ h) \quad ;$$

$$\mathsf{uncurry}.h \circ g{\times}\mathsf{id} \;=\; \mathsf{uncurry}.(h{\circ}g) \quad ;$$

$$f{\leftharpoonup}\mathsf{id} \circ \mathsf{curry}.h \;=\; \mathsf{curry}.(f{\circ}h) \quad ;$$

$$\mathsf{curry}.h \circ g \;=\; \mathsf{curry}.(h \circ g{\times}\mathsf{id}) \quad ;$$

$$g{\times}\mathsf{id} = \mathsf{uncurry}.(\mathsf{pair} \circ g) \quad ;$$

$$f{\leftharpoonup}\mathsf{id} = \mathsf{curry}.(f \circ \mathsf{eval}) \quad .$$

An example of a category with exponents is Fun. Here is an implementation:

$$\mathsf{curry} = (f :: (u :: (v :: f.(u, v)))) \ ;$$

$$\mathsf{uncurry} = (g :: (u, v :: (g.u).v)) \ ;$$

$$\mathsf{pair} = (u :: (v :: (u, v))) \ ;$$

$$\mathsf{eval} = (f, u :: f.u) \ .$$

The names were chosen to be familiar, except that the function pair is not well-known enough to have a common name. It is the curry'd 'comma' and pairs its argument. Another common name for eval is 'apply'.

Exercise 4.25 Show that Nat has exponents. What is $m \leftharpoonup n$?

□

Exercise 4.26 Consider the category POset.(subsets.$S, \subseteq$) for the subsets of a set S ordered by the is-subset-of relation $\subseteq$, which is the converse of the contains-ordering. So the product in this category is set union $\cup$—and not intersection!—for it is the sum in POset.(subsets.$S, \supseteq$). Show that this category has exponents.

□

Chapter 5

Cartesian Closed Categories

5.1 The exponent bifunctor

We show that in a category $\mathcal{C}$ with exponents $\leftharpoonup$ can be extended to a bifunctor $(\leftharpoonup) \in \mathcal{C} \leftarrow \mathcal{C}\times\mathcal{C}^{op}$, called the *exponent bifunctor*.

As always, we first construct a candidate for the arrow mapping from the typing requirements. We aim at the application of Theorem 2.58 and construct mappings for the sections $x\leftharpoonup$, having already—by definition—that $\leftharpoonup z$ is a functor. So we need a collection of mappings G_u such that $G_u g \in (u\leftharpoonup x) \xleftarrow{\mathcal{C}} (u\leftharpoonup y) \Leftarrow g \in x \xleftarrow{\mathcal{C}^{op}} y$. Here we go:

$$
\begin{array}{ll}
 & G_u g \in (u\leftharpoonup x) \xleftarrow{\mathcal{C}} (u\leftharpoonup y) \\
\Leftarrow & \quad \{ \quad G_u g \;:=\; \mathsf{curry}.h \quad \} \\
 & \mathsf{curry}.h \in (u\leftharpoonup x) \xleftarrow{\mathcal{C}} (u\leftharpoonup y) \\
\Leftarrow & \quad \{ \quad \text{typing rule for } \mathsf{curry} \quad \} \\
 & h \in u \xleftarrow{\mathcal{C}} (u\leftharpoonup y)\times x \\
\Leftarrow & \quad \{ \quad h \;:=\; \mathsf{eval} \circ k;\ \circ\text{-TYPING} \quad \} \\
 & \mathsf{eval}_{u,y} \circ k \in u \xleftarrow{\mathcal{C}} (u\leftharpoonup y)\times x \\
\Leftarrow & \quad \{ \quad \text{typing of } \mathsf{eval} \quad \} \\
 & k \in (u\leftharpoonup y)\times y \xleftarrow{\mathcal{C}} (u\leftharpoonup y)\times x \\
\Leftarrow & \quad \{ \quad k \;:=\; \mathsf{id}_{u\leftharpoonup y}\times g;\ \times \text{ is a functor} \quad \} \\
 & g \in y \xleftarrow{\mathcal{C}} x \\
\equiv & \quad \{ \quad \text{definition of } \mathcal{C}^{op} \quad \} \\
 & g \in x \xleftarrow{\mathcal{C}^{op}} y \quad ,
\end{array}
$$

giving $G_u g = \mathsf{curry}.(\mathsf{eval}_{u,y} \circ \mathsf{id}_{u\leftharpoonup y}\times g)$.

From here on we drop the subscripts. Before proceeding we derive an auxiliary result, namely: $\mathsf{eval} \circ Gg \times \mathsf{id} \;=\; \mathsf{eval} \circ \mathsf{id}\times g$:

$$
\begin{array}{ll}
 & \mathsf{eval} \circ Gg \times \mathsf{id} \;=\; \mathsf{eval} \circ \mathsf{id}\times g \\
\equiv & \quad \{ \quad \text{definition of } \mathsf{uncurry} \quad \} \\
 & \mathsf{uncurry} \,.\, Gg \;=\; \mathsf{eval} \circ \mathsf{id}\times g \\
\equiv & \quad \{ \quad \mathsf{curry} \text{ and } \mathsf{uncurry} \text{ are each others' inverse} \quad \} \\
 & Gg = \mathsf{curry}.(\mathsf{eval} \circ \mathsf{id}\times g) \\
\equiv & \quad \{ \quad \text{definition of } G \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

To verify that each G is a functor, we check:

$$
\begin{array}{ll}
 & G(g;h) \\
= & \quad \{ \quad \text{definition of ; in } \mathcal{C}^{op} \quad \} \\
 & G(h\circ g) \\
= & \quad \{ \quad \text{definition of } G \quad \} \\
 & \mathsf{curry}.(\mathsf{eval} \circ \mathsf{id}\times(h\circ g)) \\
= & \quad \{ \quad \times \text{ is bifunctor} \quad \} \\
 & \mathsf{curry}.(\mathsf{eval} \circ \mathsf{id}\times h \circ \mathsf{id}\times g) \\
= & \quad \{ \quad \text{auxiliary result above} \quad \} \\
 & \mathsf{curry}.(\mathsf{eval} \circ Gh \times \mathsf{id} \circ \mathsf{id}\times g) \\
= & \quad \{ \quad \textsc{Sectioned commute} \text{ (for } \times) \quad \} \\
 & \mathsf{curry}.(\mathsf{eval} \circ \mathsf{id}\times g \circ Gh \times \mathsf{id}) \\
= & \quad \{ \quad \mathsf{curry} \text{ fusion} \quad \} \\
 & \mathsf{curry}.(\mathsf{eval} \circ \mathsf{id}\times g) \circ Gh \\
= & \quad \{ \quad \text{candidate definition for } G \quad \} \\
 & Gg \circ Gh \quad .
\end{array}
$$

Further,

$$
\begin{array}{ll}
 & G\mathsf{id} \\
= & \quad \{ \quad \text{definition of } G \quad \} \\
 & \mathsf{curry}.(\mathsf{eval} \circ \mathsf{id}\times\mathsf{id})
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \quad \times \text{ is bifunctor} \quad \} \\
& \mathsf{curry}.\mathsf{eval} \\
= & \{ \quad \text{definition of } \mathsf{eval} \quad \} \\
& \mathsf{curry}.(\mathsf{uncurry}.\mathsf{id}) \\
= & \{ \quad \mathsf{curry} \text{ and } \mathsf{uncurry} \text{ are each others' inverse} \quad \} \\
& \mathsf{id} \quad .
\end{array}
$$

We may appeal now to Theorem 2.58 if we can show that $Ff \circ Gg = Gg \circ Ff$, where $Ff = f \leftharpoonup \mathsf{id} = \mathsf{curry}.(f \circ \mathsf{eval})$, so that $f \circ \mathsf{eval} = \mathsf{uncurry}.(Ff)$:

$$
\begin{array}{ll}
& Ff \circ Gg \\
= & \{ \quad \text{definition of } F \quad \} \\
& \mathsf{curry}.(f \circ \mathsf{eval}) \circ Gg \\
= & \{ \quad \mathsf{curry} \text{ fusion} \quad \} \\
& \mathsf{curry}.(f \circ \mathsf{eval} \circ Gg \times \mathsf{id}) \\
= & \{ \quad \text{auxiliary result above} \quad \} \\
& \mathsf{curry}.(f \circ \mathsf{eval} \circ \mathsf{id} \times g) \\
= & \{ \quad f \circ \mathsf{eval} = \mathsf{uncurry}.(Ff) \quad \} \\
& \mathsf{curry}.(\mathsf{uncurry}.(Ff) \circ \mathsf{id} \times g) \\
= & \{ \quad \text{definition of } \mathsf{uncurry} \quad \} \\
& \mathsf{curry}.(\mathsf{eval} \circ Ff \times \mathsf{id} \circ \mathsf{id} \times g) \\
= & \{ \quad \text{Sectioned commute (for } \times) \quad \} \\
& \mathsf{curry}.(\mathsf{eval} \circ \mathsf{id} \times g \circ Ff \times \mathsf{id}) \\
= & \{ \quad \mathsf{curry} \text{ fusion} \quad \} \\
& \mathsf{curry}.(\mathsf{eval} \circ \mathsf{id} \times g) \circ Ff \\
= & \{ \quad \text{definition of } G \quad \} \\
& Gg \circ Ff \quad .
\end{array}
$$

So by Theorem 2.58 we have a bifunctor indeed, where:

$$f \leftharpoonup g = \mathsf{curry}.(f \circ \mathsf{eval} \circ \mathsf{id} \times g) \quad , \qquad \leftharpoonup\text{-DEF}$$

in which the rhs is the fourth expression obtained in the preceding calculation.

Exercise 5.1 Work out what the bifunctor $\leftharpoonup$ is in Fun. Compare this to the Hom bifunctor $\overset{\mathsf{Fun}}{\leftarrow}$.

□

Exercise 5.2 Show that for a category $\mathcal{C}$ with exponents for each x the pair $(x\leftharpoonup -, x\leftharpoonup -)$ forms an adjunction between $\mathcal{C}^{op}$ and $\mathcal{C}$. Investigate the various rules for Fun. Note that this adjunction is self-dual. (Warning. Be careful not to lose track because of the contravariance, and carefully keep track of which category the various arrows belong to. Note in particular that the counit of the adjunction is a family of arrows in $\mathcal{C}^{op}$.)

□

5.2 Cartesian closed categories

Definition 5.3 (Cartesian closed category) A category is called a *cartesian closed category*, or, for short, a CCC, if: it has products, a terminal object and exponents.

□

Remark 5.4 The presence of products is already implied by the presence of exponents. Products, terminal object and exponents are all defined by so-called limit constructions (which are performed by taking the terminal object in some category). In the literature one sometimes encounters a different, non-equivalent definition for CCC, namely: the category is required to have *all* finite limits.

□

The relevance of CCCs is largely due to the fact that Fun is a CCC. We write $\times$, 1 and $\leftharpoonup$ for 'the' product, terminal object and exponent. A collection of isomorphies that hold then in all CCCs is:

$$x\times y \cong y\times x \ ;$$

$$x\times(y\times z) \cong (x\times y)\times z \ ;$$

$$1\times x \cong x \ ;$$

$$x\leftharpoonup(y\times z) \cong (x\leftharpoonup y)\leftharpoonup z \ ;$$

$$(x\times y)\leftharpoonup z \cong (x\leftharpoonup z)\times(y\leftharpoonup z) \ ;$$

$$x\leftharpoonup 1 \cong x \ ;$$

$$1\leftharpoonup x \cong 1 \ .$$

Note that writing x^y for $x\leftharpoonup y$ gives these isomorphies a more familiar appearance; for example $x^{y\times z} \cong (x^y)^z$ and $(x\times y)^z \cong x^z\times y^z$.

We construct witnesses for $1\times x \cong x$:

$$
\begin{array}{ll}
 & f \in 1\times x \leftarrow x \\
\Leftarrow & \quad \{ \quad f \;:=\; h \vartriangle k;\ \vartriangle\text{-TYPING} \quad \} \\
 & h \in 1 \leftarrow x \wedge k \in x \leftarrow x \\
\Leftarrow & \quad \{ \quad h \;:=\; [\![x]\!];\ [\![\_]\!]\text{-TYPING};\ k \;:=\; \mathsf{id} \quad \} \\
 & \mathsf{true} \ ,
\end{array}
$$

and

$$
\begin{array}{ll}
 & g \in x \leftarrow 1\times x \\
\Leftarrow & \quad \{ \quad g \;:=\; \mathsf{exr};\ \mathsf{exr}\text{-TYPING} \quad \} \\
 & \mathsf{true} \ ,
\end{array}
$$

giving $f = [\![x]\!] \vartriangle \mathsf{id}$ and $g = \mathsf{exr}$. The verification of the coherence conditions is then:

$$
\begin{array}{ll}
 & [\![x]\!] \vartriangle \mathsf{id} \circ \mathsf{exr} \;=\; \mathsf{id} \\
\equiv & \quad \{ \quad \text{lhs: } \vartriangle\text{-FUSION; rhs: } \vartriangle\text{-ID} \quad \} \\
 & ([\![x]\!] \circ \mathsf{exr}) \vartriangle \mathsf{exr} = \mathsf{exl} \vartriangle \mathsf{exr} \\
\Leftarrow & \quad \{ \quad \textsc{Leibniz} \quad \} \\
 & [\![x]\!] \circ \mathsf{exr} \;=\; \mathsf{exl} \\
\Leftarrow & \quad \{ \quad \text{transitivity of } = \quad \} \\
 & [\![x]\!] \circ \mathsf{exr} = [\![1\times x]\!] \;\wedge\; [\![1\times x]\!] = \mathsf{exl} \\
\Leftarrow & \quad \{ \quad \text{left conjunct: } [\![\_]\!]\text{-FUSION; right conjunct: } [\![\_]\!]\text{-CHAR} \quad \} \\
 & \mathsf{exr} \in x \leftarrow 1\times x \wedge \mathsf{exl} \in 1 \leftarrow 1\times x \ ,
\end{array}
$$

which is the correct typing in this context, and

$$
\begin{array}{ll}
 & \mathsf{exr} \circ [\![x]\!] \vartriangle \mathsf{id} \;=\; \mathsf{id} \\
\equiv & \quad \{ \quad \mathsf{exr}\text{-COMP} \quad \} \\
 & \mathsf{id} \ .
\end{array}
$$

Writing $!_x$ for $[\![x]\!]$, we even have a natural isomorphism $(! \vartriangle \mathsf{id}, \mathsf{exr})$ witnessing the functor isomorphy $1\times \cong \mathsf{Id}$.

Remark 5.5 The first two isomorphies above hold in any category with products, and the third in any category with products and terminal object.

□

Exercise 5.6 Show that a category with exponents and an *initial* object is a CCC.

□

Exercise 5.7 Show that Cat is a CCC.

□

Elementals, name

Definition 5.8 (Elemental) Given an object x of a CCC, an *elemental* of x is: an arrow with typing $x \leftarrow 1$, and we abbreviate $e \in x \leftarrow 1$ to $e \in x$.

□

There is a one-to-one correspondence between the elements of some type x and the elementals of the object x in Fun. The extensionality property of functions can therefore be expressed in terms of arrows: for two arrows $f, g \in x \leftarrow y$, we have $f = g$ whenever $f \circ e = g \circ e$ for all elementals $e \in y$. (This is true in Fun, but not in all CCCs!)

The concept of 'elemental' makes it possible to handle functions—thus far exclusively modelled as arrows— as elements of some 'function space', that is, as elementals of some object $x \leftarrow y$. We introduce the abbreviation $e!_y$ for $e \circ !_y$ $(= e \circ [\![y]\!])$, so $e!_y \in x \leftarrow y \Leftarrow e \in x$, and—as usual—omit the subscript when convenient.

Definition 5.9 (Name) Given a CCC-arrow $f \in x \leftarrow y$, an elemental $n \in x \leftarrow y$ is called a *name* for f if: $\mathsf{eval} \circ n! \vartriangle \mathsf{id} = f$.

□

In Fun we can express this at the point level as: $\mathsf{eval}.(n, v) = f.v$ for all values v in the domain of f[1]. There is a one-to-one correspondence between the arrow class $x \leftarrow y$ and the elementals of $x \leftarrow y$. We construct the correspondence in two steps. First, note that the functor isomorphy $1\times \cong \mathsf{Id}$ gives rise to an adjunction $(1\times, \mathsf{Id})$ between a CCC and itself. For it implies, clearly: $(x, y :: x \longleftarrow 1\times y) \cong (x, y :: \mathsf{Id}x \longleftarrow y)$. The construction of the adjungates is straightforward and gives:

$$\lceil g \rceil = g \circ \mathsf{exr} \ ;$$

$$\lfloor f \rfloor = f \circ ! \vartriangle \mathsf{id} \ .$$

[1] In Section 4.6 we saw an implementation of eval in Fun where $\mathsf{eval} = (f, u :: f.u)$. Indeed, if functions are values, we can use them as their own names. However, we might also restrict our arrows to functions definable in some formal language (e.g. typed lambda-calculus, or Haskell), and take the *string* defining the function as its name. In that case eval is less trivial to implement.

Further, the adjunction $(\times y, \leftharpoonup y)$ establishes $(x, y :: x \longleftarrow 1\times y) \cong (x, y :: (x \leftharpoonup y) \longleftarrow 1)$, with witnesses $(\mathsf{uncurry}, \mathsf{curry})$. Combining the two into $(\lfloor \_ \rfloor \circ \mathsf{uncurry}, \mathsf{curry} \circ \lceil \_ \rceil)$ gives a natural isomorphism for $(x, y :: x \longleftarrow y) \cong (x, y :: (x \leftharpoonup y) \longleftarrow 1)$. Defining $\mathsf{unname} = \lfloor \_ \rfloor \circ \mathsf{uncurry}$ and $\mathsf{name} = \mathsf{curry} \circ \lceil \_ \rceil$, we obtain:

$$\mathsf{unname} \in (x, y :: x \longleftarrow y) \dot{\leftarrow} (x, y :: (x \leftharpoonup y) \longleftarrow 1) \ ;$$

$$\mathsf{unname}.n \ = \ \mathsf{uncurry}.n \circ ! \vartriangle \mathsf{id} \ ;$$

$$\mathsf{name} \in (x, y :: (x \leftharpoonup y) \longleftarrow 1) \dot{\leftarrow} (x, y :: x \longleftarrow y) \ ;$$

$$\mathsf{name}.f \ = \ \mathsf{curry}.(f \circ \mathsf{exr}) \ .$$

Next to the fact that name and unname are each others' inverse, we have from the naturality properties of the components:

$$f \leftharpoonup g \circ \mathsf{name}.h \ = \ \mathsf{name}.(f \circ h \circ g) \qquad \text{name-FUSION}$$

for suitably typed f, g and h.

To conclude, we verify that $\mathsf{name}.f$ is indeed a name for f:

$$\begin{array}{ll} & \mathsf{eval} \circ (\mathsf{name}.f)! \vartriangle \mathsf{id} \\ = & \quad \{ \quad \text{definition of postfix } ! \quad \} \\ & \mathsf{eval} \circ (\mathsf{name}.f \circ !) \vartriangle \mathsf{id} \\ = & \quad \{ \quad \times\text{-}\vartriangle\text{-FUSION} \quad \} \\ & \mathsf{eval} \circ \mathsf{name}.f \times \mathsf{id} \circ ! \vartriangle \mathsf{id} \\ = & \quad \{ \quad \text{definition of } \mathsf{uncurry} \quad \} \\ & \mathsf{uncurry}.(\mathsf{name}.f) \circ ! \vartriangle \mathsf{id} \\ = & \quad \{ \quad \text{definition of } \mathsf{name} \quad \} \\ & \mathsf{uncurry}.(\mathsf{curry}.(f \circ \mathsf{exr})) \circ ! \vartriangle \mathsf{id} \\ = & \quad \{ \quad \mathsf{curry} \text{ and } \mathsf{uncurry} \text{ are each others' inverse} \quad \} \\ & f \circ \mathsf{exr} \circ ! \vartriangle \mathsf{id} \\ = & \quad \{ \quad \mathsf{exr}\text{-COMP} \quad \} \\ & f \ . \end{array}$$

5.3 Bicartesian closed categories

Definition 5.10 (Bicartesian closed category) A category is called a *bicartesian closed category*, or, for short, a BCC, if: it is a CCC, and in addition it has sums and an initial object.

□

Remark 5.11 In view of the result of Exercise 5.6., it is sufficient to require sums, an initial object and exponents.

□

The category Fun is also a BCC. We write $+$ and 0 for 'the' sum and initial object. A collection of isomorphies that hold then in all BCCs—next to those mentioned already for CCCs—is:

$$x+y \cong y+x \ ;$$

$$x+(y+z) \cong (x+y)+z \ ;$$

$$x+0 \cong x \ ;$$

$$x\times(y+z) \cong (x\times y)+(x\times z) \ ;$$

$$x^{\leftarrow}(y+z) \cong (x^{\leftarrow}y)\times(x^{\leftarrow}z) \ ;$$

$$x^{\leftarrow}0 \cong 1 \ .$$

Remark 5.12 The first two hold in any category with sums, and the third in any category with sums and initial object. The fourth isomorphy is stated using only $+$ and $\times$, and thus it may seem that it holds in any category with sums and products. However, that is not the case. This is in fact easy to see: because $+$ and $\times$ are dual, the isomorphy—if universally true—would also hold when dualised. But, for example, in Nat (see Exercise 5.15) it is definitely not the case that $x+(y\times z) \cong (x+y)\times(x+z)$. So the presence of exponents is essential.

□

Exercise 5.13 Prove that $x\times(y+z) \cong (x\times y)+(x\times z)$.

□

Exercise 5.14 Show that $0^{\leftarrow}x \cong 0$ if there exists an elemental $e\in x$.

□

Exercise 5.15 Show that Nat is a BCC.

□

Chapter 6

Algebras

6.1 Algebras

Recall that a *monoid* is a structure $(A,\oplus,e)$,in which A is a set (or type), $\oplus \in A \leftarrow A\times A$ and $e\in A$, satisfying the *monoid laws*: $\oplus$ is an associative operation, and e is a neutral (or identity) element for $\oplus$. Actually, we are at the moment not interested in these laws. Some concrete examples of monoids are:

$(\mathbb{N}, + ,0)$, the naturals with addition ;

$(\mathbb{B},\equiv,\mathsf{true})$, the booleans with equivalence ;

$(\mathbb{B},\vee,\mathsf{false})$, the booleans with disjunction ;

$(\mathbb{B},\wedge,\mathsf{true})$, the booleans with conjunction ;

$(\mathbb{R},\times,1)$, the reals with multiplication ;

$(\mathbb{N}, \uparrow ,0)$, the naturals with a maximum operation ;

$(A^\star , +\!\!+ , \varepsilon)$, the words over some alphabet with concatenation ;

and so on and so on.

Monoid are algebras in the traditional sense of algebraic theory: a set (the *carrier*), together with a bunch of functions to that set. A monoid $(A,\oplus,e)$ can be modelled in category theory as follows. First, we use the bijection between elements and elementals of A in the CCC Fun to treat e as an arrow, so that $e \in A \leftarrow 1$. Now $\oplus$ and e have the same codomain, so we are allowed to combine them into a single arrow $\oplus \triangledown e \in A \leftarrow A\times A+1$. This arrow has the same information as the pair of $\oplus$ and e, since $\triangledown$ is injective, so we take the liberty to rewrite $(A,\oplus,e)$ as (A, f), where $f = \oplus \triangledown e$.

Define M as the functor $(x :: (x\times x)+1)$, that is,

$Mx = (x\times x)+1$ on objects ;

$$Mf = (f \times f) + \mathsf{id}_1 \text{ on arrows} \quad .$$

Then each monoid is of the form (A, f), with $f \in A \leftarrow MA$.

Generalizing this to other algebras in the traditional sense, in general they have some form $(A, f_0 \in A \leftarrow F_0A, f_1 \in A \leftarrow F_1A, \ldots, f_n \in A \leftarrow F_nA)$, and putting $f = f_0 \triangledown f_1 \triangledown \ldots \triangledown f_n$ and $F = F_0 + F_1 + \ldots + F_n$ this can be combined into (A, f), with $f \in A \leftarrow FA$. This leads to the following definition.

Definition 6.1 (Algebra) Given an endofunctor F on a category $\mathcal{C}$, an *F-algebra* is: a pair (x, f) where $x \in \mathcal{C}$ and $f \in x \xleftarrow{\mathcal{C}} Fx$.

□

We will often use Greek letters like φ, χ and ψ as variables that stand for algebras. For an algebra $\varphi = (x, f)$ the object x is called its *carrier*, and we denote it by $\mathsf{car}.\varphi$. Note that actually the carrier component is redundant, since $x = \mathsf{cod}.f$. In some interesting generalisations of the notion of algebra, however, the carrier component is not redundant, reason to keep it here as well. However, we shall freely apply the following convention: In a context where a $\mathcal{C}$-arrow is required, we can use the algebra $\varphi = (x, f)$, meaning its arrow component f; conversely, where an algebra is required we can use a $\mathcal{C}$-arrow f, provided it has typing $x \longleftarrow Fx$ for some $x \in \mathcal{C}$, and we mean then (x, f).

6.2 Homomorphisms

Returning to algebras in the traditional sense, given two monoids $\varphi = (A, \oplus, e)$ and $\psi = (B, \otimes, u)$, a function $h \in A \leftarrow B$ is called a monoid homomorphism to φ from ψ if it preserves the monoid structure; more precisely, if the following two rules hold:

$$(h.x) \oplus (h.y) = h.(x \otimes y) \text{ for all } x, y \in B;$$

$$e = h.u \quad .$$

Concrete examples of monoid homomorphisms are:

$$(n :: n > 0) \in (\mathbb{B}, \vee, \mathsf{false}) \leftarrow (\mathbb{N}, +, 0) \quad ;$$

$$\neg \in (\mathbb{B}, \wedge, \mathsf{true}) \leftarrow (\mathbb{B}, \vee, \mathsf{false}) \quad ;$$

$$\neg \circ (n :: n > 0) \in (\mathbb{B}, \wedge, \mathsf{true}) \leftarrow (\mathbb{N}, +, 0) \quad ;$$

$$(n :: n > 0) \in (\mathbb{B}, \vee, \mathsf{false}) \leftarrow (\mathbb{N}, \uparrow, 0) \quad ;$$

$$exp \in (\mathbb{R}, \times, 1) \leftarrow (\mathbb{N}, +, 0) \quad ;$$

$$length \in (\mathbb{N}, +, 0) \leftarrow (A^\star, +\!\!+, \varepsilon) \quad .$$

For example, *length* is a homomorphism because $(length.x)+(length.y)=length.(x+\!\!+y)$ and $0=length.\varepsilon$.

In order to move to category theory, let us work out a point-free formulation of the statement that $h \in (A, \oplus \triangledown e) \leftarrow (B, \otimes \triangledown u)$. First, h has to be a function to A from B. Furthermore,

$$
\begin{array}{ll}
 & \forall(x,y :: (h.x)\oplus(h.y) = h.(x\otimes y)) \ \wedge \ e = h.u \\
\equiv & \quad \{ \quad \text{FUN-}\circ\text{-COMP; definition of } \times \quad \} \\
 & \forall(x,y :: ((\oplus \circ h{\times}h).(x,y) = (h{\circ}\otimes).(x,y))) \ \wedge \ e = h.u \\
\equiv & \quad \{ \quad \text{EXTENSIONALITY; interpreting } e \text{ and } u \text{ as elementals of } A \text{ and } B \quad \} \\
 & \oplus \circ h{\times}h \ = \ h{\circ}\otimes \ \wedge \ e = h{\circ}u \\
\equiv & \quad \{ \quad \text{neutrality of } \mathsf{id}\text{; injectivity of } \triangledown \quad \} \\
 & (\oplus \circ h{\times}h) \triangledown (e{\circ}\mathsf{id}_1) \ = \ (h{\circ}\otimes) \triangledown (h{\circ}u) \\
\equiv & \quad \{ \quad \text{lhs: } \triangledown\text{-+-FUSION; rhs: } \triangledown\text{-FUSION} \quad \} \\
 & \oplus \triangledown e \circ (h{\times}h){+}\mathsf{id}_1 \ = \ h \circ \otimes \triangledown u \\
\equiv & \quad \{ \quad \text{definition of } M \quad \} \\
 & \oplus \triangledown e \circ Mh \ = \ h \circ \otimes \triangledown u \ .
\end{array}
$$

It is clear now how we can give a categorical definition of the notion of homomorphism.

Definition 6.2 (Homomorphism) Given an endofunctor F on a category $\mathcal{C}$, an *F-homomorphism* to an F-algebra φ from an F-algebra ψ is: a $\mathcal{C}$-arrow h such that:

$$h \in \mathsf{car}.\varphi \xleftarrow{\mathcal{C}} \mathsf{car}.\psi \quad \wedge \quad \varphi \circ Fh \ = \ h{\circ}\psi \ .$$

Anticipating the definition of the category of F-algebras, we abbreviate this condition on h to: $h \in \varphi \leftarrow \psi$.

□

(The arrow equality $\varphi \circ Fh \ = \ h{\circ}\psi$ is the coherence condition for the two ways available for constructing an arrow with typing $\mathsf{car}.\varphi \xleftarrow{\mathcal{C}} F(\mathsf{car}.\psi)$.)

Example 6.3 Assume that $\mathcal{A} = (A, \sqsupseteq)$ is a partially ordered set. An endofunctor F on $\mathsf{POset}.\mathcal{A}$ is a monotonic function on A. For this category an F-algebra is a pair (x, f) where $x{\in}A$ and $f \in x \xleftarrow{\mathsf{POset}.\mathcal{A}} Fx$. From the definition of $\mathsf{POset}.\mathcal{A}$ we know that the latter is true if: $f = (x, F.x)$ and $x \sqsupseteq F.x$. So f is completely determined by x, and the F-algebras correspond precisely to the elements $x{\in}A$ for which $x \sqsupseteq F.x$. In lattice theory these are known as pre-fixpoints.

□

6.3 The category of F-algebras

Definition 6.4 ($\mathsf{Alg}.F$) Given an endofunctor F on a category $\mathcal{C}$, the category $\mathsf{Alg}.F$ is defined as follows:

Objects: The F-algebras.

Arrows: The F-homomorphisms.

Composition: That of $\mathcal{C}$.

Identities: Those of $\mathcal{C}$, where id_φ in $\mathsf{Alg}.F$ is $\mathsf{id}_{\mathsf{car}.\varphi}$ in $\mathcal{C}$.

(To be precise, this is a precategory, which can be turned into a category with the triple trick.)

□

Associativity of composition and neutrality of identities are met because they are inherited from the base category. It must be checked, though, that arrow composition preserves the restriction on arrows. The verification is as follows:

$$
\begin{array}{ll}
 & h{\circ}k \in \varphi \overset{\mathsf{Alg}.F}{\longleftarrow} \psi \\
\equiv & \quad \{ \quad \text{definition of } \overset{\mathsf{Alg}.F}{\longleftarrow} \quad \} \\
 & h{\circ}k \in \mathsf{car}.\varphi \overset{\mathcal{C}}{\longleftarrow} \mathsf{car}.\psi \ \ \wedge \ \ \varphi \circ F(h{\circ}k) \ = \ h{\circ}k{\circ}\psi \\
\equiv & \quad \{ \quad F \text{ is a functor} \quad \} \\
 & h{\circ}k \in \mathsf{car}.\varphi \overset{\mathcal{C}}{\longleftarrow} \mathsf{car}.\psi \ \ \wedge \ \ \varphi \circ Fh \circ Fk \ = \ h{\circ}k{\circ}\psi \\
\Leftarrow & \quad \{ \quad \text{properties of } \circ \quad \} \\
 & h \in \mathsf{car}.\varphi \overset{\mathcal{C}}{\longleftarrow} \mathsf{car}.\psi \ \wedge \ \varphi{\circ}Fh = h{\circ}\chi \ \wedge \ k \in \mathsf{car}.\psi \overset{\mathcal{C}}{\longleftarrow} \mathsf{car}.\psi \ \wedge \ \chi{\circ}Fk = k{\circ}\psi \\
\equiv & \quad \{ \quad \text{definition of } \overset{\mathsf{Alg}.F}{\longleftarrow} \quad \} \\
 & h \in \varphi \overset{\mathsf{Alg}.F}{\longleftarrow} \chi \wedge k \in \chi \overset{\mathsf{Alg}.F}{\longleftarrow} \psi \ \ .
\end{array}
$$

A consequence of the convention to let an algebra $\varphi = (x, f)$ also stand for its arrow component f is the following typing rule:

$$\varphi \in \mathsf{car}.\varphi \overset{\mathcal{C}}{\longleftarrow} F(\mathsf{car}.\varphi) \equiv \varphi{\in}\mathsf{Alg}.F \ \ . \qquad \text{ALGEBRA-TYPING}$$

6.4 Initial algebras

In lattice theory we have the notion of a least pre-fixpoint of a monotonic endofunction F, which, if it exists, coincides with the least fixpoint. 'Translating' all the concepts to the corresponding concepts to category theory, we end up with the notion of an *initial algebra for an endofunctor* F.

6.4.1 Definition and properties of initial algebras

Definition 6.5 (Initial F-algebra) Given an endofunctor F on a category $\mathcal{C}$, an *initial algebra for* F, or *initial* F*-algebra*, is: an initial object of $\mathsf{Alg}.F$.

□

Let us work out the consequences of this definition in detail. Assume that F has an initial algebra, and fix one which we will call 'the' initial F-algebra. In lattice theory we denote the least prefix point of a function f by μf. We will denote the initial F-algebra by in_F, and its carrier will be denoted by μF. For the arrow typings in the base category $\mathcal{C}$, furthermore, the superscript $\mathcal{C}$ will be omitted. So, by ALGEBRA-TYPING, the fact that in_F is an object of $\mathsf{Alg}.F$ gives us:

$$\mu F \in \mathcal{C} \quad ,$$

$$\mathsf{in}_F \in \mu F \leftarrow F\mu F \quad . \qquad \mathsf{in}\text{-TYPING}$$

The catamorphism mapping $(\!|\_|\!)$ of the initiality will be written as $(\!|\_|\!)_F$ here. Assume that $\varphi \in \mathsf{Alg}.F$, with $\mathsf{car}.\varphi = x$ (which, by ALGEBRA-TYPING, equivales $\varphi \in x \leftarrow Fx$). The $(\!|\_|\!)$-CHAR rule for initialities becomes:

$$\begin{array}{ll} & (\!|\varphi|\!)_F = h \\ \equiv & \quad \{ \quad (\!|\_|\!)\text{-CHAR} \quad \} \\ & h \in \varphi \overset{\mathsf{Alg}.F}{\longleftarrow} \mathsf{in}_F \\ \equiv & \quad \{ \quad \text{definition of } \overset{\mathsf{Alg}.F}{\longleftarrow} \quad \} \\ & h \in x \leftarrow \mu F \ \ \wedge \ \ \varphi \circ Fh \ = \ h \circ \mathsf{in}_F \quad . \end{array}$$

By the substitution $h := (\!|\varphi|\!)_F$, and making the typing assumption on φ explicit, we find this typing rule for $(\!|\_|\!)_F$:

$$(\!|\varphi|\!)_F \in x \leftarrow \mu F \Leftarrow \varphi \in x \leftarrow Fx \quad . \qquad (\!|\_|\!)_F\text{-TYPING}$$

Recall that, when writing an arrow equality like $(\!|\varphi|\!)_F = h$, we imply that both sides can be typed and have the same typing. The conjunct '$h \in x \leftarrow \mu F$' in the unfolded definition of $\overset{\mathsf{Alg}.F}{\longleftarrow}$ may therefore be omitted, giving the characterisation rule for initial algebras:

$$(\!|\varphi|\!)_F = h \ \ \equiv \ \ \varphi \circ Fh \ = \ h \circ \mathsf{in}_F \quad . \qquad (\!|\_|\!)_F\text{-CHAR}$$

Up to now, all we have done is to rephrase the abstract definition of initial algebra in a concrete but *equivalent* way. Thus, we can give the following alternative definition of initial algebra:

Theorem 6.6 $((\mu F, \mathsf{in}_F), (\!|\_|\!)_F)$ is an initial F-algebra if in-TYPING, $(\!|\_|\!)_F$-TYPING and $(\!|\_|\!)_F$-CHAR are satisfied.

□

Just as for initiality in general, we obtain a number of useful rules from the characterisation rule by simple substitutions. The substitution $h := (\!|\varphi|\!)_F$ gives us the computation rule:

$$\varphi \circ F(\!|\varphi|\!)_F \;=\; (\!|\varphi|\!)_F \circ \mathsf{in}_F \quad . \tag{$(\!|\text{-}|\!)_F$-COMP}$$

The substitution φ,$h := \mathsf{in}_F$,$\mathsf{id}_{\mu F}$ gives us the identity rule:

$$(\!|\mathsf{in}_F|\!)_F = \mathsf{id}_{\mu F} \quad . \tag{$(\!|\text{-}|\!)_F$-ID}$$

Finally, we derive the fusion rule:

$$\begin{array}{ll}
 & (\!|\varphi|\!)_F = h \circ (\!|\psi|\!)_F \\
\equiv & \quad \{ \quad (\!|\text{-}|\!)_F\text{-CHAR with } h := h \circ (\!|\psi|\!)_F \quad \} \\
 & \varphi \circ F(h \circ (\!|\psi|\!)_F) \;=\; h \circ (\!|\psi|\!)_F \circ \mathsf{in}_F \\
\equiv & \quad \{ \quad F \text{ is a functor} \quad \} \\
 & \varphi \circ Fh \circ F(\!|\psi|\!)_F \;=\; h \circ (\!|\psi|\!)_F \circ \mathsf{in}_F \\
\equiv & \quad \{ \quad (\!|\text{-}|\!)_F\text{-COMP} \quad \} \\
 & \varphi \circ Fh \circ F(\!|\psi|\!)_F \;=\; h \circ \psi \circ F(\!|\psi|\!)_F \\
\Leftarrow & \quad \{ \quad \text{LEIBNIZ} \quad \} \\
 & \varphi \circ Fh \;=\; h \circ \psi \quad ,
\end{array}$$

which proves:

$$(\!|\varphi|\!)_F = h \circ (\!|\psi|\!)_F \quad \Leftarrow \quad \varphi \circ Fh \;=\; h \circ \psi \quad . \tag{$(\!|\text{-}|\!)_F$-FUSION}$$

(These rules could, of course, have been directly obtained from the general $(\!|\text{-}|\!)$-rules; for example, the last rule is the concrete translation of $(\!|\varphi|\!)_F = h \circ (\!|\psi|\!)_F \;\Leftarrow\; h \in \varphi \overset{\mathsf{Alg}.F}{\longleftarrow} \psi$.)

If if it is clear from the context, such as typing considerations, for which functor F we have an initial algebra, the subscripts F are usually omitted. We repeat the concrete rules here, omitting the subscripts:

$$\mathsf{in} \in \mu F \leftarrow F\mu F \quad . \tag{in-TYPING}$$

$$(\!|\varphi|\!) \in x \leftarrow \mu F \Leftarrow \varphi \in x \leftarrow Fx \quad . \tag{$(\!|\text{-}|\!)$-TYPING}$$

$$(\!|\varphi|\!) = h \quad \equiv \quad \varphi \circ Fh \;=\; h \circ \mathsf{in} \quad . \tag{$(\!|\text{-}|\!)$-CHAR}$$

$$\varphi \circ F(\!|\varphi|\!) \;=\; (\!|\varphi|\!) \circ \mathsf{in} \quad . \qquad (\!|\_|\!)\text{-COMP}$$

$$(\!|\mathsf{in}|\!) = \mathsf{id} \quad . \qquad (\!|\_|\!)\text{-ID}$$

$$(\!|\varphi|\!) = h \circ (\!|\psi|\!) \;\Leftarrow\; \varphi \circ Fh \;=\; h \circ \psi \quad . \qquad (\!|\_|\!)\text{-FUSION}$$

The removal of the subscripts F has created a certain ambiguity, since the rule $(\!|\_|\!)$-CHAR may also stand for $(\!|\varphi|\!) = h \;\equiv\; h \in \varphi \leftarrow \mathsf{in}$. However, this is a harmless ambiguity, and in fact these rules are equivalent under the assumption of suitable typing.

6.4.2 Data types as initial algebras

Consider the following functional program:

```
data Nattree = join Nattree Nattree | leaf Nat

maxval (join t0 t1) = max (maxval t0) (maxval t1)

maxval (leaf i) = i
```

which defines a data type `Nattree` of binary trees with naturals at the leaf nodes, and a function `maxval` to compute the maximum value occurring in the leaves of the tree. This is a common way to define a function on a recursively defined data type.

Let us view this in algebraic terms, renaming for the sake of brevity `Nat` to $\mathbb{N}$, `Nattree` to t, `join` and `leaf` to, respectively, $\pm$ and λ, `maxval` to h, and `max` to $\uparrow$. Then we have:

$$\pm \in t \leftarrow t \times t \quad ,$$

$$\lambda \in t \leftarrow \mathbb{N} \quad ,$$

$$h \in \mathbb{N} \leftarrow t \quad ,$$

$$h \circ (\pm) \;=\; (\uparrow) \circ h \times h \quad ,$$

$$h \circ \lambda = \mathsf{id} \quad .$$

Putting $T = (x :: (x \times x) + \mathbb{N})$, we see that

$$(\pm) \triangledown \lambda \in t \leftarrow Tt \quad ,$$

$$(\uparrow) \triangledown \mathsf{id} \in \mathbb{N} \leftarrow T\mathbb{N} \quad ,$$

so both are T-algebras. The defining equations for h can be combined into the *equivalent* form

$$h \in (\mathbb{N}, (\uparrow) \triangledown \mathsf{id}) \leftarrow (t, (\pm) \triangledown \lambda) \quad ,$$

so h is a T-homomorphism. The interesting thing is that this typing of h is apparently sufficient to define h.

This is so *independent* of the codomain algebra. For consider the general definitional pattern

```
h (join t0 t1) = op (h t0) (h t1)

h (leaf i) = f i
```

for suitably typed functions `f` and `op`. Whatever the choice for these two functions, these equations will define some function on our nattrees. And yet, the equations are equivalent (with some renaming) to:

$$h \in \varphi \leftarrow (t, (\pm) \triangledown \lambda) \text{ where } \varphi = (x, (\oplus) \triangledown f) \text{ for some type } x.$$

So we see that, by virtue of $\triangledown$-FORM, for *any* φ this defines a function h, or, in other words, there is a unique arrow to φ from $(t, (\pm) \triangledown \lambda)$. That now was the definition of an initiality, in this case for an initial algebra. Conclusion: $(t, (\pm) \triangledown \lambda)$ is an initial T-algebra (where the base category is Fun). If we agree that it is 'the' initial T-algebra, the rule $(\!|\_|\!)$-CHAR tells us now how to solve for h; for the `maxval` example we obtain $h = (\!|(\uparrow) \triangledown \mathsf{id}|\!)$.

The final conclusion is that a data type definition like that for `Nattree` apparently *defines an initial algebra*. The notation implicitly determines the endofunctor, and gives names to both the carrier and (the components of) the arrow part of the initial algebra. Of course, there are many other isomorphic initial algebras, like the one defined by

```
data TreeNat = fork TreeNat TreeNat | tip Nat .
```

Here are the naturals as an initial algebra:

```
data Nat = succ Nat | zero .
```

The corresponding endofunctor is $(x :: x{+}1)$. There are many isomorphic models of 'the' naturals, and yet we speak of 'the naturals', since no 'naturally' expressible property could possibly distinguish between these models. Doing mathematics would become awkward indeed if, instead of just referring to '$\mathbb{N}$', 'succ' and 'zero' as if we knew and agreed which particular algebra this stands for, we always had to say: 'Let $(\mathbb{N}, \mathsf{succ} \triangledown \mathsf{zero})$ be some initial $(x :: x{+}1)$-algebra.' The use of 'the' as a useful fiction is thus by no means a category-theoretical innovation, but general mathematical practice.

Exercise 6.7 Express the catamorphism $(\!|f \triangledown e|\!)$ on the naturals in conventional mathematical notation. What do we get for $(\!|\_|\!)$-ID then?

□

Exercise 6.8 Define the data type boolean (with constructors true and false) as an initial algebra. What are the catamorphisms? Express the function $\neg$ as a catamorphism.

□

Exercise 6.9 What is the initial $\mathsf{K}.a$-algebra?

□

Exercise 6.10 What is the initial $\mathsf{Id}_{\mathcal{C}}$-algebra, if it exists?

□

6.4.3 Applications

Cons lists

The following application is based on `Set`, expressed in a functional language. The cons-list-of-naturals data type can be defined by:

```
data Natlist = cons Nat Natlist | empty
```

where we usually write `n:x` instead of `cons n x` and `[]` instead of `empty`. The endofunctor is $L = (x :: (\mathbb{N}\times x)+1)$, and we put $Natlist = \mu L$. Let us go the reverse way, from the categorical treatment to functional programming language notation, to interpret the catamorphisms for these cons-lists. The $(\!|\_|\!)$-COMP rule for this type is:

$$
\begin{array}{ll}
& \varphi \circ (x :: (\mathbb{N}\times x)+1)(\!|\varphi|\!) \;=\; (\!|\varphi|\!)\circ\mathsf{in} \\
\equiv & \quad \{ \quad \text{working out the functor application} \quad \} \\
& \varphi \circ (\mathsf{id}\times(\!|\varphi|\!))+\mathsf{id} \;=\; (\!|\varphi|\!)\circ\mathsf{in} \\
\equiv & \quad \{ \quad \text{using } \triangledown\text{-FORM to put } \varphi = (\oplus) \triangledown e \text{ and } \mathsf{in} = (:) \triangledown [\,] \quad \} \\
& (\oplus) \triangledown e \circ (\mathsf{id}\times(\!|(\oplus) \triangledown e|\!))+\mathsf{id} \;=\; (\!|(\oplus) \triangledown e|\!) \circ (:) \triangledown [\,] \\
\equiv & \quad \{ \quad \text{abbreviate } (\!|(\oplus) \triangledown e|\!) \text{ to } f \quad \} \\
& (\oplus) \triangledown e \circ (\mathsf{id}\times f)+\mathsf{id} \;=\; f \circ (:) \triangledown [\,] \\
\equiv & \quad \{ \quad \text{lhs: } \triangledown\text{-+-FUSION, neutrality of id; rhs: } \triangledown\text{-FUSION} \quad \} \\
& ((\oplus) \circ \mathsf{id}\times f) \triangledown e = (f\circ(:)) \triangledown (f\circ[\,]) \\
\equiv & \quad \{ \quad \triangledown \text{ is injective} \quad \} \\
& (\oplus) \circ \mathsf{id}\times f \;=\; f\circ(:) \;\wedge\; e = f\circ[\,] \\
\equiv & \quad \{ \quad \text{EXTENSIONALITY, swapping the arguments to } = \quad \} \\
& \forall(n, x :: (f\circ(:)).(n, x) = ((\oplus) \circ \mathsf{id}\times f).(n, x)) \;\wedge\; f\circ[\,] = e
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{ \quad \text{FUN-}\circ\text{-COMP} \quad \} \\
& \forall(n, x :: f.(n : x) = (\oplus).((\mathsf{id}\times f).(n, x))) \ \wedge \ f\circ[\,] = e \\
\equiv & \{ \quad \text{product for } \texttt{Set} \quad \} \\
& \forall(n, x :: f.(n : x) \ = \ n \oplus f.x) \ \wedge \ f\circ[\,] = e
\end{array}
$$

Writing `foldr op e` for $f = (\!|(\oplus) \triangledown e|\!)$, and interpreting the functions $[\,] \in \mathit{Natlist} \leftarrow 1$ and $e \in z \leftarrow 1$ (for some z) as data values $[\,] \in \mathit{Natlist}$ and $e \in z$, we see that the defining equations are:

```
foldr op e (n:x) = op n (foldr op e x)

foldr op e [] = e
```

This is, to functional programmers, a well-known function. For example, `foldr (+) 0` sums the elements of a list, and if we define `tri n y = (3×n):y`, the function `foldr tri []` triples all elements. Note that the equations, written point-free, amount to these specialisations of $(\!|\_|\!)$-COMP for *Natlist*:

$$
\begin{array}{l}
(\!|(\oplus) \triangledown e|\!)\circ(:) \ = \ (\oplus) \circ \mathsf{id}\times(\!|(\oplus) \triangledown e|\!) \ , \\
(\!|(\oplus) \triangledown e|\!)\circ[\,] = e \ ,
\end{array}
$$

which we shall use in a moment.

In a similar way we can work out the $(\!|\_|\!)$-FUSION rule for this type, and obtain:

$$
(\!|(\oplus) \triangledown e|\!) = h\circ(\!|(\otimes) \triangledown u|\!) \quad \Leftarrow \quad (\oplus) \circ \mathsf{id}\times h \ = \ h\circ(\otimes) \ \wedge \ e = h\circ u \ .
$$

Let us see how we can use this to fuse `foldr (+) 0` after `foldr tri []` so as to obtain a one-pass computation. So we want to fuse $(\!|(+) \triangledown 0|\!)\circ(\!|(\otimes) \triangledown [\,]|\!)$, in which $\otimes$ is defined as $(x, xs :: (3\times x) : xs)$, which we can write as $(:) \circ (3\times)\times\mathsf{id}$. We solve the fusion condition for $\oplus$ and e. We start with $\oplus$:

$$
\begin{array}{ll}
& (\oplus) \circ \mathsf{id}\times h \ = \ h\circ(\otimes) \\
\equiv & \{ \quad h \text{ and } \otimes \text{ in this case} \quad \} \\
& (\oplus) \circ \mathsf{id}\times(\!|(+) \triangledown 0|\!) \ = \ (\!|(+) \triangledown 0|\!) \circ (:) \circ (3\times)\times\mathsf{id} \\
\equiv & \{ \quad (\!|\_|\!)\text{-COMP for } \mathit{Natlist} \quad \} \\
& (\oplus) \circ \mathsf{id}\times(\!|(+) \triangledown 0|\!) \ = \ (+) \circ \mathsf{id}\times(\!|(+) \triangledown 0|\!) \circ (3\times)\times\mathsf{id} \\
\equiv & \{ \quad \times \text{ is a functor} \quad \} \\
& (\oplus) \circ \mathsf{id}\times(\!|(+) \triangledown 0|\!) \ = \ (+) \circ (3\times)\times\mathsf{id} \circ \mathsf{id}\times(\!|(+) \triangledown 0|\!) \\
\Leftarrow & \{ \quad \text{LEIBNIZ} \quad \} \\
& (\oplus) \ = \ (+) \circ (3\times)\times\mathsf{id} \\
\equiv & \{ \quad \text{point-level} \quad \} \\
& (\oplus) = (x, y :: 3\times x + y) \ .
\end{array}
$$

As to e:

$$
\begin{array}{ll}
& e = h \circ u \\
\equiv & \quad \{ \quad h \text{ and } u \text{ in this case} \quad \} \\
& e = (\!|(+) \triangledown 0|\!) \circ [\,] \\
\equiv & \quad \{ \quad (\!|\_|\!)\text{-COMP for } \mathit{Natlist} \quad \} \\
& e = 0 \quad .
\end{array}
$$

So we have derived this one-pass algorithm:

```
(foldr (+) 0) ∘ (foldr tri []) = foldr triplus 0

   where triplus n z = 3×n + z .
```

Exercise 6.11 Prove that, slightly more generally, for *Natlist* we can fuse as follows:

$$(\!|(\oplus) \triangledown e|\!) \circ (\!|((:) \circ f \times \mathsf{id}) \triangledown [\,]|\!) = (\!|((\oplus) \circ f \times \mathsf{id}) \triangledown e|\!) \quad .$$

□

Banana split

If we tuple two recursive computations over the same data structure, can we express this as a one-pass computation? The following result is generic; it applies in all categories of algebras, as long as the base category has products.

Put formally, the problem is to find, given F-algebras φ and ψ, an F-algebra χ such that $(\!|\chi|\!) = (\!|\varphi|\!) \vartriangle (\!|\psi|\!)$. We solve this equation for χ:

$$
\begin{array}{ll}
& (\!|\chi|\!) = (\!|\varphi|\!) \vartriangle (\!|\psi|\!) \\
\equiv & \quad \{ \quad \vartriangle\text{-CHAR} \quad \} \\
& (\!|\varphi|\!) = \mathsf{exl} \circ (\!|\chi|\!) \;\wedge\; (\!|\psi|\!) = \mathsf{exr} \circ (\!|\chi|\!) \\
\equiv & \quad \{ \quad (\!|\_|\!)\text{-CHAR} \quad \} \\
& \varphi \circ F(\mathsf{exl} \circ (\!|\chi|\!)) \;=\; \mathsf{exl} \circ (\!|\chi|\!) \circ \mathsf{in} \;\wedge\; \psi \circ F(\mathsf{exr} \circ (\!|\chi|\!)) \;=\; \mathsf{exr} \circ (\!|\chi|\!) \circ \mathsf{in} \\
\equiv & \quad \{ \quad (\!|\_|\!)\text{-COMP} \quad \} \\
& \varphi \circ F(\mathsf{exl} \circ (\!|\chi|\!)) \;=\; \mathsf{exl} \circ \chi \circ F(\!|\chi|\!) \;\wedge\; \psi \circ F(\mathsf{exr} \circ (\!|\chi|\!)) \;=\; \mathsf{exr} \circ \chi \circ F(\!|\chi|\!) \\
\equiv & \quad \{ \quad F \text{ is a functor} \quad \} \\
& \varphi \circ F\mathsf{exl} \circ F(\!|\chi|\!) \;=\; \mathsf{exl} \circ \chi \circ F(\!|\chi|\!) \;\wedge\; \psi \circ F\mathsf{exr} \circ F(\!|\chi|\!) \;=\; \mathsf{exr} \circ \chi \circ F(\!|\chi|\!) \\
\Leftarrow & \quad \{ \quad \text{LEIBNIZ} \quad \}
\end{array}
$$

$$
\begin{array}{ll}
& \varphi \circ F\mathsf{exl} \;=\; \mathsf{exl} \circ \chi \;\;\wedge\;\; \psi \circ F\mathsf{exr} \;=\; \mathsf{exr} \circ \chi \\
\equiv & \qquad \{ \quad \vartriangle\text{-CHAR} \quad \} \\
& \chi = (\varphi \circ F\mathsf{exl}) \vartriangle (\psi \circ F\mathsf{exr}) \\
\equiv & \qquad \{ \quad \times\text{-}\vartriangle\text{-FUSION} \quad \} \\
& \chi \;=\; \varphi \times \psi \;\circ\; F\mathsf{exl} \;\vartriangle\; F\mathsf{exr} \;\; .
\end{array}
$$

We have proved:

Theorem 6.12 For all F-algebras φ and ψ, if the base category has products:

$$(\!|\varphi \times \psi \;\circ\; F\mathsf{exl} \;\vartriangle\; F\mathsf{exr}|\!) = (\!|\varphi|\!) \vartriangle (\!|\psi|\!) \;\; .$$

□

Exercise 6.13 Show that $F\mathsf{exl} \;\vartriangle\; F\mathsf{exr}$ is a natural transformation with typing $(x, y :: Fx \times Fy) \dot{\leftarrow} (x, y :: F(x \times y))$ (or, more succinctly, $(\times)(F \times F) \dot{\leftarrow} F(\times)$).

□

6.4.4 Initial algebra as fixed point

In lattice theory, a least pre-fixpoint is a fixpoint. In category theory, an initial F-algebra is also a fixpoint, in the following sense:

Theorem 6.14 If in is an initial F-algebra, then $F\mathsf{in}$ is also an F-algebra, and $\mathsf{in} \cong F\mathsf{in}$.

Proof In general, if φ is an F-algebra with carrier x, we have in the base category $\varphi \in x \leftarrow Fx$, and so $F\varphi \in Fx \leftarrow FFx$ is also an F-algebra, this time with carrier Fx; so, first, we see that $\mathsf{car}.(F\varphi) \;=\; F\;(\mathsf{car}.\varphi)$, and, second, that $F\mathsf{in}$ is indeed an F-algebra. The typings of the witnesses h and k required for an isomorphism are given by $h \in \mathsf{in} \overset{\mathsf{Alg}.F}{\longleftarrow} F\mathsf{in}$ and $k \in F\mathsf{in} \overset{\mathsf{Alg}.F}{\longleftarrow} \mathsf{in}$.

As to h, we have:

$$
\begin{array}{ll}
& h \in \mathsf{in} \overset{\mathsf{Alg}.F}{\longleftarrow} F\mathsf{in} \\
\equiv & \qquad \{ \quad \text{definition of } \overset{\mathsf{Alg}.F}{\longleftarrow}\text{; } \mathsf{car}.\mathsf{in} = \mu F \quad \} \\
& h \in \mu F \leftarrow F\mu F \;\;\wedge\;\; \mathsf{in} \circ Fh \;=\; h \circ F\mathsf{in} \\
\Leftarrow & \qquad \{ \quad \text{both conjuncts suggest trying } h \;:=\; \mathsf{in} \quad \} \\
& \mathsf{in} \in \mu F \leftarrow F\mu F \;\;\wedge\;\; \mathsf{in} \circ F\mathsf{in} \;=\; \mathsf{in} \circ F\mathsf{in} \\
\equiv & \qquad \{ \quad \mathsf{in}\text{-TYPING; reflexivity of } = \quad \} \\
& \mathsf{true} \;\; .
\end{array}
$$

For k:

$$
\begin{array}{ll}
 & k \in F\mathsf{in} \overset{\mathsf{Alg}.F}{\longleftarrow} \mathsf{in} \\
\equiv & \quad \{ \quad (\!|\text{-}|\!)\text{-CHAR} \quad \} \\
 & (\!|F\mathsf{in}|\!) = k \\
\Leftarrow & \quad \{ \quad k := (\!|F\mathsf{in}|\!) \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

It thus suffices to prove that $\mathsf{in} \circ (\!|F\mathsf{in}|\!) = \mathsf{id}$ and $(\!|F\mathsf{in}|\!) \circ \mathsf{in} = \mathsf{id}$. We have, first,

$$
\begin{array}{ll}
 & \mathsf{in} \circ (\!|F\mathsf{in}|\!) = \mathsf{id} \\
\equiv & \quad \{ \quad (\!|\text{-}|\!)\text{-ID} \quad \} \\
 & \mathsf{in} \circ (\!|F\mathsf{in}|\!) = (\!|\mathsf{in}|\!) \\
\Leftarrow & \quad \{ \quad (\!|\text{-}|\!)\text{-FUSION} \quad \} \\
 & \mathsf{in} \in \mathsf{in} \overset{\mathsf{Alg}.F}{\longleftarrow} F\mathsf{in} \\
\equiv & \quad \{ \quad \text{shown above} \quad \} \\
 & \mathsf{true} \quad ,
\end{array}
$$

and next,

$$
\begin{array}{ll}
 & (\!|F\mathsf{in}|\!) \circ \mathsf{in} \\
= & \quad \{ \quad (\!|\text{-}|\!)\text{-COMP} \quad \} \\
 & F\mathsf{in} \circ F(\!|F\mathsf{in}|\!) \\
= & \quad \{ \quad F \text{ is a functor} \quad \} \\
 & F(\mathsf{in} \circ (\!|F\mathsf{in}|\!)) \\
= & \quad \{ \quad \text{proved immediately above} \quad \} \\
 & F\mathsf{id} \\
= & \quad \{ \quad F \text{ is a functor} \quad \} \\
 & \mathsf{id} \quad .
\end{array}
$$

This completes the proof.
□

6.5 Lawful algebras

In monoid algebras we have not only the signature—expressed by the functor $M = (x :: (x \times x)+1)$—but also the monoid laws. An algebra $\oplus \triangledown e$ is only a monoid algebra if it satisfies the monoid laws. Using α to denote the natural isomorphism witnessing $(x, y, z :: (x \times y) \times z) \cong (x, y, z :: x \times (y \times z))$, these laws are expressed by the arrow equalities $\mathsf{assoc}.(\oplus)$ for the associativity of $\oplus$ and $\mathsf{neutral}.((\oplus),e)$ for e being neutral to $\oplus$, where:

$$\mathsf{assoc}.(\oplus) \quad \equiv \quad (\oplus) \circ (\oplus) \vartriangle \mathsf{id} \circ \alpha \quad = \quad (\oplus) \circ \mathsf{id} \vartriangle (\oplus)$$

and

$$\mathsf{neutral}.((\oplus),e) \quad \equiv \quad (\oplus) \circ e \vartriangle \mathsf{id} \quad = \quad \mathsf{id} \quad = \quad (\oplus) \circ \mathsf{id} \vartriangle e \quad .$$

For an arbitrary M-algebra φ we can obtain the laws in terms of φ by the substitution $((\oplus),e) \quad := \quad (\varphi \circ \mathsf{inl}, \varphi \circ \mathsf{inr})$. So the predicate testing an M-algebra for being a monoid algebra is:

$$(\varphi :: \mathsf{assoc}.(\oplus) \wedge \mathsf{neutral}.((\oplus),e) \text{ where } ((\oplus),e) = (\varphi \circ \mathsf{inl}, \varphi \circ \mathsf{inr})) \quad .$$

In general, a class of algebras may be restricted by a collection of laws, which can be expressed as some predicate. The predicate above is an example of an M-law. Generalising this, we have:

Definition 6.15 (F-law) Given an endofunctor F on a category $\mathcal{C}$, an *F-law* is: a predicate whose domain is the class of F-algebras.

□

Definition 6.16 ($\mathcal{L}$-lawful) Given an F-law $\mathcal{L}$, an *$\mathcal{L}$-lawful* algebra is: an F-algebra satisfying $\mathcal{L}$.

□

We define now the *category* of lawful algebras:

Definition 6.17 ($\mathsf{Alg}.(F, \mathcal{L})$) Given an endofunctor F on a category $\mathcal{C}$ and an F-law $\mathcal{L}$, the category $\mathsf{Alg}.(F, \mathcal{L})$ is: the full subcategory of $\mathsf{Alg}.F$ whose objects are the $\mathcal{L}$-lawful algebras.

□

If $\mathsf{Alg}.(F, \mathcal{L})$ has an initial object, we denote it by $(\mu(F, \mathcal{L}), \mathsf{in}_{(F,\mathcal{L})})$. (A true but somewhat useless theorem is: if in_F is $\mathcal{L}$-lawful, then $(\mu(F, \mathcal{L}), \mathsf{in}_{(F,\mathcal{L})}) \cong (\mu F, \mathsf{in}_F)$. The problem is

that if $\mathcal{L}$, as is the intention, represents a collection of laws, the initial (lawless) F-algebra, if it exist, is only $\mathcal{L}$-lawful when *all* F-algebras are.)

In the definition of $\mathsf{Alg}.(F,\mathcal{L})$ we took a *full* subcategory of $\mathsf{Alg}.F$. This means that the definition of the arrows is as before; they are the F-homomorphisms. Of course, for $h \in \varphi \xleftarrow{\mathsf{Alg}.F} \psi$ to be an arrow in $\mathsf{Alg}.(F,\mathcal{L})$, φ and ψ have to be objects of $\mathsf{Alg}.(F,\mathcal{L})$, that is, they have to satisfy $\mathcal{L}$. But other than that, there is no restriction on h.

Example 6.18 (Join lists) In Section 6.4.2 we looked at binary trees of naturals, being—based upon Fun—the initial T-algebra for $T = (x :: (x \times x)+\mathbb{N})$. Let us impose now a law $\mathcal{L}$ on the T-algebras, so that a T-algebra $(\oplus) \triangledown f$ is $\mathcal{L}$-lawful if: $\oplus$ is associative. Clearly, the join operation $\pm$ on trees is not associative. There is a data type for which there *is* an associative join: lists with the operation $+\!\!+$. We shall argue now that (non-empty) lists of naturals with the operations $+\!\!+$ and $[\_]$ (for forming singleton lists) are the initial $(T,\mathcal{L})$-algebra.

The argument is in two parts. First we show that for any T-algebra there is *at most* one arrow (i.e., T-homomorphism) to it from the proposed algebra by constructing a catamorphism in another category of algebras, namely the (lawless) L-algebras for $L = (x :: (\mathbb{N} \times x)+\mathbb{N})$, giving rise to the data type of non-empty cons lists of naturals. We use that for lists the cons operation $(:)$ equals $(+\!\!+) \circ [\_] \vartriangle \mathsf{id}$. Next we show that if the target algebra is a $(T,\mathcal{L})$-algebra, that arrow *is* a T-homomorphism.

So assume that $(\oplus) \triangledown f$ is a T-algebra, and $h \in (\oplus) \triangledown f \leftarrow (+\!\!+) \triangledown [\_]$ a T-homomorphism. The homomorphic condition requires that $h \circ (+\!\!+) = (\oplus) \circ h \times h$ and $h \circ [\_] = f$. Then:

$$
\begin{array}{ll}
 & h \circ (:) \\
= & \quad \{ \quad \text{cons operation} \quad \} \\
 & h \circ (+\!\!+) \circ [\_] \vartriangle \mathsf{id} \\
= & \quad \{ \quad \text{homomorphic condition} \quad \} \\
 & (\oplus) \circ h \times h \circ [\_] \vartriangle \mathsf{id} \\
= & \quad \{ \quad \times\text{-}\vartriangle\text{-FUSION} \quad \} \\
 & (\oplus) \circ (h \circ [\_]) \vartriangle h \\
= & \quad \{ \quad \text{homomorphic condition} \quad \} \\
 & (\oplus) \circ f \vartriangle h
\end{array}
$$

The combination of $h \circ (:) = (\oplus) \circ f \vartriangle h$ and $h \circ [\_] = f$ tells us that h is an L-homomorphism, namely to $((\oplus) \circ f \vartriangle h) \triangledown f$ from $(:) \triangledown [\_]$. Since the source algebra is initial, there is precisely one such homomorphism: it is a catamorphism.

Is that L-homomorphism also a T-homomorphism? We have to show that the function h defined by $h \circ (:) = (\oplus) \circ f \vartriangle h$ and $h \circ [\_] = f$ satisfies $h \circ (+\!\!+) = (\oplus) \circ h \times h$ if $\oplus$ is associative. We reason at the point-level and show that $h.(x +\!\!+ y) = h.x \oplus h.y$ by induction on the length of x, with basis: $h.([a] +\!\!+ y) = h.[a] \oplus h.y$, and step: $h.((a : x) +\!\!+ y) = h.(a : x) \oplus h.y \;\Leftarrow\; h.(x +\!\!+ y) = h.x \oplus h.y$. For the basis:

$$
\begin{array}{ll}
 & h.([a]{+\!\!+}y) \\
= & \quad \{ \quad \text{cons operation} \quad \} \\
 & h.(a : y) \\
= & \quad \{ \quad h \text{ is } L\text{-homomorphism} \quad \} \\
 & f.a \oplus h.y \\
= & \quad \{ \quad h \text{ is } L\text{-homomorphism} \quad \} \\
 & h.[a] \oplus h.y \quad .
\end{array}
$$

For the step:

$$
\begin{array}{ll}
 & h.((a : x){+\!\!+}y) \\
= & \quad \{ \quad (a : x){+\!\!+}y = a : (x{+\!\!+}y) \quad \} \\
 & h.(a : (x{+\!\!+}y)) \\
= & \quad \{ \quad \text{as for the basis} \quad \} \\
 & h.[a] \oplus h.(x{+\!\!+}y) \\
= & \quad \{ \quad \text{inductive hypothesis} \quad \} \\
 & h.[a] \oplus (h.x \oplus h.y) \\
= & \quad \{ \quad \oplus \text{ is associative} \quad \} \\
 & (h.[a] \oplus h.x) \oplus h.y \\
= & \quad \{ \quad \text{as for the basis} \quad \} \\
 & h.(a : x) \oplus h.y \quad .
\end{array}
$$

Using functional programming notation, we see that the same pattern that defined `maxval` on binary trees defines a unique function on 'join lists', that is, using the associative constructor $+\!\!+$:

```
maxval (x++y) = max (maxval x) (maxval y)

maxval [i] = i
```

Unfortunately, a pattern like `x++y` is not allowed in existing functional languages. Interpreted as a computational prescription, the first equation can be read as: split the argument list in *any* way into two parts, as long as both parts are simpler than the whole, and recurse. The associativity of the operation `max` guarantees that the result is independent of the way of splitting. Often, splitting into two parts that are about equally long is more efficient.

To accommodate empty lists, we just have to introduce a third constructor [ ] for empty lists, and impose the neutrality law. We leave it to the reader to verify that basically the same argument as before applies.

□

Remark 6.19 Basically, all results for lawless algebras derived earlier apply equally to lawful algebras, with one exception: Theorem 6.14 does not carry over. The problem is in already in the first part: if in is an initial $(F, \mathcal{L})$-algebra, $F\mathsf{in}$ is as before an F-algebra, but this algebra is in general not $\mathcal{L}$-lawful. In that case it is also not an object of $\mathsf{Alg}.(F, \mathcal{L})$. In $\mathsf{Alg}.F$ both in and $F\mathsf{in}$ are objects in any case, and so it is *meaningful* to ask whether they are isomorphic in that category. But if $F\mathsf{in}$ is not $\mathcal{L}$-lawful the answer is no: in is $\mathcal{L}$-lawful, so in and $F\mathsf{in}$ can be distinguished by $\mathcal{L}$, which means that they are not 'categorically indistinguishable'[1] and therefore not isomorphic.

□

6.6 Parametrised initial algebras

Above we have taken trees or list of naturals as example data types. However, a result like $(\!|\_|\!)$-COMP for *Natlist* is (as should be obvious from the proof) equally valid for lists of booleans, or strings, or whatever. Let us examine—with categorical tools—parametrised data types, such as defined by

```
data List x = cons x (List x) | empty
```

(so that the type `Natlist` from section 6.4.3 equals `List Nat`).

Throughout the remainder of this section $\oplus$ denotes a binary functor $\oplus \in \mathcal{C} \leftarrow \mathcal{D}\times\mathcal{C}$. The first argument of $\oplus$ will be the 'parameter'. By fixing it to some $x{\in}\mathcal{D}$ we obtain an endofunctor $x\oplus \in \mathcal{C} \leftarrow \mathcal{C}$, so we can consider the category of $x\oplus$-algebras. For example, for cons-lists we can take $x\oplus y = (x\times y)+1$, so that for the functor $L = (x :: (\mathbb{N}\times x)+1)$ from section 6.4.3 we have $L = \mathbb{N}\oplus$.

We assume furthermore in this section that for all objects $x{\in}\mathcal{D}$ the endofunctor $x\oplus$ has an initial algebra, denoted as usual by $\mathsf{in}_{x\oplus}$, with carrier $\mu(x\oplus)$.

6.6.1 The map functor

Consider the mapping $\varpi = (x :: \mu(x\oplus))$ to objects of $\mathcal{C}$ from objects of $\mathcal{D}$ (the 'parameters'). So, for the cons-list example, we have $\mathit{Natlist} = \mu L = \mu(\mathbb{N}\oplus) = \varpi\mathbb{N}$. Note that the typing of $\mathsf{in}_{x\oplus}$ in $\mathcal{C}$ is given by $\mathsf{in}_{x\oplus} \in \varpi x \leftarrow x \oplus \varpi x$. We will now prove that ϖ can be extended to a functor, which we call the *map functor*. Let us first construct a candidate for the arrow mapping:

[1] The argument assumes that the predicate $\mathcal{L}$ is based upon arrow equalities, as in the monoid example before.

$$
\begin{array}{ll}
 & \varpi f \in \varpi x \leftarrow \varpi y \\
\equiv & \quad \{ \quad \text{definition of } \varpi \text{ on objects} \quad \} \\
 & \varpi f \in \varpi x \leftarrow \mu(y\oplus) \\
\Leftarrow & \quad \{ \quad \varpi f \ := \ (\!|\varphi|\!)_{y\oplus} \quad \} \\
 & (\!|\varphi|\!)_{y\oplus} \in \varpi x \leftarrow \mu(y\oplus) \\
\equiv & \quad \{ \quad (\!|\text{-}|\!)\text{-TYPING} \quad \} \\
 & \varphi \in \varpi x \leftarrow y \oplus \varpi x \\
\Leftarrow & \quad \{ \quad \varphi \ := \ \mathsf{in}_{x\oplus} \circ \psi;\ \circ\text{-TYPING};\ \mathsf{in}\text{-TYPING} \quad \} \\
 & \psi \in x \oplus \varpi x \leftarrow y \oplus \varpi x \\
\Leftarrow & \quad \{ \quad \psi \ := \ f \oplus \varpi x \quad \} \\
 & f \oplus \varpi x \in x \oplus \varpi x \leftarrow y \oplus \varpi x \\
\Leftarrow & \quad \{ \quad \oplus\, \varpi x \text{ is a functor} \quad \} \\
 & f \in x \xleftarrow{\mathcal{D}} y \ .
\end{array}
$$

So the candidate found is $\varpi f = (\!|\mathsf{in}_{x\oplus} \ \circ \ f \oplus \varpi x|\!)_{y\oplus}$ for $f \in x \xleftarrow{\mathcal{D}} y$.

Before we proceed to show the functoriality of ϖ, we first derive a useful rule that will be used in the proof. It shows that any catamorphism can be fused with a map. In the derivation of the rule we omit the subscripts—which can be easily reconstructed assuming a suitable typing of the ingredients—and write $\oplus\mathsf{id}$ for the arrow mapping of $\oplus\, \varpi x$.

$$
\begin{array}{ll}
 & (\!|\varphi|\!) \circ \varpi g \ = \ (\!|\psi|\!) \\
\equiv & \quad \{ \quad \text{definition of } \varpi \text{ on arrows} \quad \} \\
 & (\!|\varphi|\!) \circ (\!|\mathsf{in} \circ g \oplus \mathsf{id}|\!) = (\!|\psi|\!) \\
\Leftarrow & \quad \{ \quad (\!|\text{-}|\!)\text{-FUSION} \quad \} \\
 & \psi \circ \mathsf{id} \oplus (\!|\varphi|\!) \ = \ (\!|\varphi|\!) \circ \mathsf{in} \circ g \oplus \mathsf{id} \\
\equiv & \quad \{ \quad (\!|\text{-}|\!)\text{-COMP} \quad \} \\
 & \psi \circ \mathsf{id} \oplus (\!|\varphi|\!) \ = \ \varphi \circ \mathsf{id} \oplus (\!|\varphi|\!) \circ g \oplus \mathsf{id} \\
\equiv & \quad \{ \quad \textsc{Section-commute} \quad \} \\
 & \psi \circ \mathsf{id} \oplus (\!|\varphi|\!) \ = \ \varphi \circ g \oplus \mathsf{id} \circ \mathsf{id} \oplus (\!|\varphi|\!) \\
\Leftarrow & \quad \{ \quad \psi \ := \ \varphi \circ g \oplus \mathsf{id} \quad \} \\
 & \mathsf{true} \ ,
\end{array}
$$

which gives us:

$$
(\!|\varphi|\!) \circ \varpi g \ = \ (\!|\varphi \circ g \oplus \mathsf{id}|\!) \ . \qquad \varpi\text{-FUSION}
$$

Another rule that is easily derived is:

$$\varpi f \circ \mathsf{in} \;=\; \mathsf{in} \circ f \oplus \varpi f \quad . \qquad \varpi\text{-COMP}$$

Now we are ready to prove the theorem.

Theorem 6.20 Given a binary functor $\oplus \in \mathcal{C} \leftarrow \mathcal{D}\times\mathcal{C}$ such that for each object $x{\in}\mathcal{D}$ the functor $x\oplus$ has an initial algebra, we obtain a functor $\varpi \in \mathcal{C} \leftarrow \mathcal{D}$ by defining:

(6.21) $\varpi x = \mu(x\oplus)$ for $x{\in}\mathcal{D}$,

(6.22) $\varpi f = (\!|\mathsf{in}_{x\oplus} \circ f \oplus \varpi x|\!)_{y\oplus}$ for $f \in x \xleftarrow{\mathcal{D}} y$.

Proof For all $f \in x \xleftarrow{\mathcal{D}} y$ and $g \in y \xleftarrow{\mathcal{D}} z$:

$$\begin{array}{ll} & \varpi f \circ \varpi g \\ = & \quad \{ \quad \text{definition of } \varpi \quad \} \\ & (\!|\mathsf{in}_{x\oplus} \circ f \oplus \varpi x|\!)_{y\oplus} \circ \varpi g \\ = & \quad \{ \quad \varpi\text{-FUSION} \quad \} \\ & (\!|\mathsf{in}_{x\oplus} \circ f \oplus \varpi x \circ g \oplus \varpi x|\!)_{z\oplus} \\ = & \quad \{ \quad \oplus\varpi x \text{ is a functor} \quad \} \\ & (\!|\mathsf{in}_{x\oplus} \circ (f{\circ}g) \oplus \varpi x|\!)_{z\oplus} \\ = & \quad \{ \quad \text{definition of } \varpi \quad \} \\ & \varpi(f{\circ}g) \quad . \end{array}$$

Furthermore,

$$\begin{array}{ll} & \varpi \mathsf{id}_x \\ = & \quad \{ \quad \text{definition of } \varpi \quad \} \\ & (\!|\mathsf{in}_{x\oplus} \circ \mathsf{id}_x \oplus \varpi x|\!)_{x\oplus} \\ = & \quad \{ \quad \oplus\varpi x \text{ is a functor} \quad \} \\ & (\!|\mathsf{in}_{x\oplus}|\!)_{x\oplus} \\ = & \quad \{ \quad (\!|\_|\!)\text{-ID}; \; \mu(x\oplus) = \varpi x \quad \} \\ & \mathsf{id}_{\varpi x} \quad . \end{array}$$

□

Example 6.23 (maps in Fun) For arrows in Fun the object mapping of ϖ is a data type constructor; for example, taking $x \oplus y = (x \times y)+1$, ϖ corresponds to `List` from the beginning of this section, and $\varpi\mathbb{N}$ to `List Nat`. Working out the details of ϖ-COMP and casting them in functional-programming style gives the defining equations for the arrow mapping:

```
listmap f (x:xs) = (f x):(listmap f xs)

listmap f [] = []
```

This is, of course, the well-known standard `map` function on lists.

As is well known, the higher-order function `map` is a polymorphic function: it accepts function arguments of any type. However, it is tied to lists for the next argument. In contrast, ϖ is a generic construction giving a 'map' for *any* data type constructor that can be obtained from the parametrised initial algebra construction. For example, we can generalise the type `Nattree` from section 6.4.2 to

```
data Tree x = join (Tree x) (Tree x) | leaf x
```

(corresponding to ϖx for $x \oplus y = (y \times y)+x$), and obtain:

```
treemap f (join t0 t1) = join (treemap f t0) (treemap f t1)

treemap f (leaf i) = f i
```

Constructions like ϖ which stand for a whole *class of algorithms*, one for each data type constructor, are called *polytypic*.

□

Exercise 6.24 Work out ϖ-FUSION specialised to the bifunctor $x \oplus y = (x \times y)+1$. Compare this to Exercise 6.11.

□

Exercise 6.25 Prove the rule ϖ-COMP.

□

Exercise 6.26 Derive a ϖ-CHAR rule.

□

6.6.2 Reduce

Assume now that $\oplus$ is such that $x \oplus y \;=\; Fy + x$ for some endofunctor $F \in \mathcal{C} \leftarrow \mathcal{C}$, in which x does not occur free in F. This is the case for the bifunctor $x \oplus y = (y \times y) + x$ giving rise to the `Tree` types, but not for that giving the cons-lists. The endofunctors $x\oplus$ are then called *free endofunctors*, and ϖx is called a *free type* over x.

We can then define a polytypic function ϱ, called *reduce*, with the following typing rule:

$$\varrho.\varphi \in x \leftarrow \varpi x \Leftarrow \varphi \in x \leftarrow Fx \quad . \qquad \varrho\text{-TYPING}$$

We construct a definition:

$$\begin{array}{ll}
 & \varrho.\varphi \in x \leftarrow \varpi x \\
\equiv & \quad \{ \quad \text{definition of } \varpi \text{ on objects} \quad \} \\
 & \varrho.\varphi \in x \leftarrow \mu(x\oplus) \\
\Leftarrow & \quad \{ \quad \varrho.\varphi \;:=\; (\!|\psi|\!) \quad \} \\
 & (\!|\psi|\!) \in x \leftarrow \mu(x\oplus) \\
\equiv & \quad \{ \quad (\!|_|\!)\text{-TYPING} \quad \} \\
 & \psi \in x \leftarrow x \oplus x \\
\equiv & \quad \{ \quad x\oplus y \;=\; Fy + x \quad \} \\
 & \psi \in x \leftarrow Fx + x \\
\Leftarrow & \quad \{ \quad \psi \;:=\; \varphi \triangledown \mathsf{id};\ \triangledown\text{-TYPING} \quad \} \\
 & \varphi \in x \leftarrow Fx \quad ,
\end{array}$$

so we obtain:

$$\varrho.\varphi = (\!|\varphi \triangledown \mathsf{id}|\!) \quad .$$

(Note that the typing of ϱ shows that the mapping cannot be extended to a functor.) A catamorphism of the form $\varrho.\varphi$ is called a *reduction*. We examine ϖ-FUSION for a reduction:

$$\begin{array}{ll}
 & \varrho.\psi \circ \varpi f \\
= & \quad \{ \quad \text{definition of } \varrho \quad \} \\
 & (\!|\psi \triangledown \mathsf{id}|\!) \circ \varpi f \\
= & \quad \{ \quad \varpi\text{-FUSION} \quad \} \\
 & (\!|\psi \triangledown \mathsf{id} \circ f\oplus\mathsf{id}|\!) \\
= & \quad \{ \quad x\oplus y \;=\; Fy + x \quad \}
\end{array}$$

$$
\begin{array}{ll}
 & (\!|\psi \triangledown \mathsf{id} \ \circ \ F\mathsf{id} + f|\!) \\
= & \qquad \{ \quad F \text{ is a functor; } \triangledown\text{-+FUSION} \quad \} \\
 & (\!|\psi \triangledown f|\!) \ .
\end{array}
$$

So we have:

$$\varrho.\psi \circ \varpi f \ = \ (\!|\psi \triangledown f|\!) \ . \qquad \varrho\text{-}\varpi\text{-FUSION}$$

Because of $\triangledown$-FORM, this means that any catamorphism on a free type can be decomposed into a reduce after a map:

$$
\begin{array}{ll}
 & (\!|\varphi|\!) \\
= & \qquad \{ \quad \triangledown\text{-FORM} \quad \} \\
 & (\!|(\varphi \circ \mathsf{inl}) \triangledown (\varphi \circ \mathsf{inr})|\!) \\
= & \qquad \{ \quad \varrho\text{-}\varpi\text{-FUSION} \quad \} \\
 & \varrho.(\varphi \circ \mathsf{inl}) \circ \varpi(\varphi \circ \mathsf{inr}) \ .
\end{array}
$$

Although this is essentially the same rule as ϱ-ϖ-FUSION, we record it separately:

$$(\!|\varphi|\!) \ = \ \varrho.(\varphi \circ \mathsf{inl}) \circ \varpi(\varphi \circ \mathsf{inr}) \ . \qquad \text{FREE-}(\!|\_|\!)\text{-DECOMP}$$

We can use ϱ to construct a natural transformation called *flatten* and denoted by μ[2], with typing $\mu \in \varpi \dot{\leftarrow} \varpi\varpi$:

$$
\begin{array}{ll}
 & \mu_x \in \varpi x \leftarrow \varpi\varpi x \\
\Leftarrow & \qquad \{ \quad \mu_x := \varrho.\varphi; \quad \} \\
 & \varrho.\varphi \in \varpi x \leftarrow \varpi\varpi x \\
\Leftarrow & \qquad \{ \quad \varrho\text{-TYPING} \quad \} \\
 & \varphi \in \varpi x \leftarrow F\varpi x \\
\Leftarrow & \qquad \{ \quad \varphi \ := \ \psi \circ \mathsf{inl};\ \circ\text{-TYPING};\ \mathsf{inl}\text{-TYPING} \quad \} \\
 & \psi \in \varpi x \leftarrow F\varpi x + x \\
\equiv & \qquad \{ \quad x \oplus y \ = \ Fy + x \quad \} \\
 & \psi \in \varpi x \leftarrow x \oplus \varpi x \\
\Leftarrow & \qquad \{ \quad \psi \ := \ \mathsf{in}_{x\oplus} \quad \} \\
 & \mathsf{true} \ ,
\end{array}
$$

[2]This μ has no relationship to the fixed-point operator μ

giving the definition:

$$\mu = \varrho.(\mathsf{in}\circ\mathsf{inl}) \quad .$$

That this is indeed a natural transformation will be shown later. First we give fusion rules for μ, which we call *promotion* rules. The most general rule is for promoting an arbitrary catamorphism. First:

$$\begin{array}{ll} & (\!|\varphi|\!)\circ\mu = (\!|\psi|\!) \\ \equiv & \quad \{ \quad \text{definitions of } \mu \text{ and } \varrho \quad \} \\ & (\!|\varphi|\!)\circ(\!|(\mathsf{in}\circ\mathsf{inl}) \triangledown \mathsf{id}|\!) = (\!|\psi|\!) \\ \Leftarrow & \quad \{ \quad (\!|\_|\!)\text{-FUSION} \quad \} \\ & (\!|\varphi|\!) \circ (\mathsf{in}\circ\mathsf{inl}) \triangledown \mathsf{id} \;\; = \;\; \psi \;\circ\; F(\!|\varphi|\!) + \mathsf{id} \quad . \end{array}$$

We interrupt the derivation and first develop the last lhs:

$$\begin{array}{ll} & (\!|\varphi|\!) \circ (\mathsf{in}\circ\mathsf{inl}) \triangledown \mathsf{id} \\ = & \quad \{ \quad \triangledown\text{-FUSION} \quad \} \\ & ((\!|\varphi|\!)\circ\mathsf{in}\circ\mathsf{inl}) \;\triangledown\; (\!|\varphi|\!) \\ = & \quad \{ \quad (\!|\_|\!)\text{-COMP} \quad \} \\ & (\varphi \;\circ\; F(\!|\varphi|\!) + \mathsf{id} \;\circ\; \mathsf{inl}) \;\triangledown\; (\!|\varphi|\!) \\ = & \quad \{ \quad \mathsf{inl}\text{-LEAP} \quad \} \\ & (\varphi \;\circ\; \mathsf{inl} \;\circ\; F(\!|\varphi|\!)) \;\triangledown\; (\!|\varphi|\!) \\ = & \quad \{ \quad \triangledown\text{-}+\text{-FUSION} \quad \} \\ & \varphi\circ\mathsf{inl} \;\triangledown\; (\!|\varphi|\!) \;\circ\; F(\!|\varphi|\!) + \mathsf{id} \quad . \end{array}$$

Now we continue where we left off:

$$\begin{array}{ll} & (\!|\varphi|\!) \circ (\mathsf{in}\circ\mathsf{inl}) \triangledown \mathsf{id} \;\; = \;\; \psi \;\circ\; F(\!|\varphi|\!) + \mathsf{id} \\ \equiv & \quad \{ \quad \text{just derived} \quad \} \\ & \varphi\circ\mathsf{inl} \;\triangledown\; (\!|\varphi|\!) \;\circ\; F(\!|\varphi|\!) + \mathsf{id} \;\; = \;\; \psi \;\circ\; F(\!|\varphi|\!) + \mathsf{id} \\ \Leftarrow & \quad \{ \quad \text{LEIBNIZ} \quad \} \\ & \varphi\circ\mathsf{inl} \;\triangledown\; (\!|\varphi|\!) \;\; = \;\; \psi \quad , \end{array}$$

giving $(\!|\varphi|\!)\circ\mu = (\!|\varphi\circ\mathsf{inl} \;\triangledown\; (\!|\varphi|\!)|\!)$, which by virtue of FREE-$(\!|\_|\!)$-DECOMP can be expressed as the rule:

$$(\!|\varphi|\!)\circ\mu \;\; = \;\; \varrho.(\varphi\circ\mathsf{inl}) \circ \varpi(\!|\varphi|\!) \quad . \qquad\qquad (\!|\_|\!)\text{-PROMO}$$

Specialised promotion rules that follow easily from this general rule, and whose derivation is left as an exercise to the reader,are:

$$\varpi f \circ \mu \;=\; \mu \circ \varpi\varpi f \quad ; \qquad \varpi\text{-PROMO}$$

$$\varrho.\varphi \circ \mu \;=\; \varrho.\varphi \circ \varpi\varrho.\varphi \quad . \qquad \varrho\text{-PROMO}$$

Conversely, general catamorphism promotion follows from map and reduce promotion (using $(\!|\_|\!)$-DECOMP).

For the even more special case of $\mu = \varrho.(\mathsf{in}\circ\mathsf{inl})$, the promotion rule obtained is:

$$\mu\circ\mu \;=\; \mu \circ \varpi\mu \quad . \qquad \mu\text{-PROMO}$$

The ϖ-PROMO rule is in fact the promised proof that μ is a natural transformation.

A kind of counterpart to μ is given by:

$$\eta = \mathsf{in}\circ\mathsf{inr} \in \varpi\dot{\leftarrow}\mathsf{Id} \quad .$$

We list its fusion properties: _

$$\varpi f \circ \eta \;=\; \eta\circ f \quad ; \qquad \varpi\text{-}\eta\text{-COMP}$$

$$\varrho.\varphi \circ \eta = \mathsf{id} \quad ; \qquad \varrho\text{-}\eta\text{-COMP}$$

$$\mu\circ\eta = \mathsf{id} \quad . \qquad \mu\text{-}\eta\text{-COMP}$$

The ϖ-η-COMP rule states that η is a natural transformation.

To conclude this investigation, we derive:

$$\begin{array}{ll}
 & \mu \circ \varpi\eta \\
= & \quad \{ \quad \text{definition of } \mu \quad \} \\
 & \varrho.(\mathsf{in}\circ\mathsf{inl}) \circ \varpi\eta \\
= & \quad \{ \quad \varrho\text{-}\varpi\text{-FUSION} \quad \} \\
 & (\!|(\mathsf{in}\circ\mathsf{inl}) \triangledown \eta|\!) \\
= & \quad \{ \quad \text{definition of } \eta \quad \} \\
 & (\!|(\mathsf{in}\circ\mathsf{inl}) \triangledown (\mathsf{in}\circ\mathsf{inr})|\!) \\
= & \quad \{ \quad \triangledown\text{-FUSION} \quad \} \\
 & (\!|\mathsf{in} \circ \mathsf{inl} \triangledown \mathsf{inr}|\!) \\
= & \quad \{ \quad \triangledown\text{-ID} \quad \} \\
 & (\!|\mathsf{in}|\!) \\
= & \quad \{ \quad (\!|_|\!)\text{-ID} \quad \} \\
 & \mathsf{id} \quad .
\end{array}$$

Typing shows that both this equality of natural transformations and the one in μ-η-COMP have the most general typing $\varpi \dot{\leftarrow} \varpi$, so that we can combine the two rules into:

$$\mu \circ \eta \;=\; \mathsf{id} \;=\; \mu \circ \varpi\eta \quad .$$

Exercise 6.27 Prove the rules ϖ-PROMO and ϱ-PROMO.

□

Exercise 6.28 Work out a definition for the functions `treereduce` and `treeflatten`, corresponding to ϱ and μ, for the data type constructor `Tree` from the end of Section 6.6.1. Derive an expression for `maxval` from Section 6.4.2 using `treereduce`.

□

Exercise 6.29 Use promotion rules and the result of the previous exercise to derive a more efficient way for computing `maxval`$\circ$`treeflatten`.

□

6.7 Existence of initial algebras

This section is informal; it contains no proofs. We are concerned with the question when an initial F-algebra exists in Fun. If F is such that it can be viewed as a 'signature', we can construct a so-called *term algebra*, by taking the signature as defining a tree grammar, and the terms are then the productions of the grammar. Together they form the carrier of the algebra. This is in fact how data types (defined with `data`) are implemented in functional programming languages. Further, previously defined parametrised data types may be applied in such data definitions.

We introduce the concept of a *rational* endofunctor over an 'alphabet' $\mathcal{A}$ of objects and 'object variables' to capture this categorically. We use $\dot{+}$ to denote the functor operation $F\dot{+}G = (x :: Fx + Gx)$, and likewise $\dot{\times}$ for the lifted version of $\times$. The rational endofunctors over $\mathcal{A}$ are the class of functors built inductively from Id, $\mathsf{K}.a$ for $a \in \mathcal{A}$, $\dot{+}$, $\dot{\times}$, and $\varpi = (x :: \mu(x\oplus))$ provided that $x\oplus$ is a rational endofunctor over $\mathcal{A} \cup \{x\}$. The rational endofunctors over the objects of Fun all have initial algebras; they can in fact all be expressed as defined data types in Haskell or Gofer.

If laws are imposed that can be expressed equationally, all rational endofunctors have lawful initial algebras. The laws force certain terms to be identified; for example, the *terms* $(x \# y) \# z$ are different trees than the *terms* $x \# (y \# z)$ but must represent the same value and so be lumped together if associativity is imposed. Mathematically this can be expressed by creating equivalence classes of terms that can be converted into each

other by using the laws as two-way rewrite rules. To obtain again an algebra, now with these equivalence classes as the elements of the carrier, the constructors are defined to operate on the equivalence classes by: (i) take representatives (which are terms) from the classes; (ii) apply the constructor on the terms giving a new term; (iii) take the equivalence class to which it belongs. The algebra obtained is then the desired lawful initial algebra.

As long as we do not use equality tests, we can compute efficiently on representatives. For equality testing it may help to convert to some 'normal form', but in general no efficient method is known.

Chapter 7

Monads

For μ and η in Section 6.6.2 we had the following situation:

$$\mu \in \varpi \dot{\leftarrow} \varpi\varpi \quad ;$$

$$\eta \in \varpi \dot{\leftarrow} \mathsf{Id} \quad ;$$

$$\mu \circ \mu \;=\; \mu \circ \varpi\mu \quad ;$$

$$\mu \circ \eta \;=\; \mathsf{id} \;=\; \mu \circ \varpi\eta \quad .$$

This is an example of a monad.

7.1 Definition of monad

Definition 7.1 (Monad) A *monad* over a category $\mathcal{C}$ is: a triple (M,μ,η), in which M is an endofunctor on $\mathcal{C}$ and $\mu \in M \dot{\leftarrow} MM$ and $\eta \in M \dot{\leftarrow} \mathsf{Id}$ are two natural transformations, satisfying the *monad laws*:

$$\mu \circ \mu_M \;=\; \mu \circ M\mu \in M \dot{\leftarrow} MMM \quad ;$$

$$\mu \circ \eta_M \;=\; \mathsf{id}_M \;=\; \mu \circ M\eta \in M \dot{\leftarrow} M \quad .$$

□

Remark 7.2 The monad laws are examples of coherence conditions.

□

We show that every adjunction gives rise to a monad.

Theorem 7.3 Assume that (F, G), for functors $F \in \mathcal{C} \leftarrow \mathcal{D}$ and $G \in \mathcal{D} \leftarrow \mathcal{C}$, forms an adjunction with unit $\mathsf{unit} \in GF \dot{\leftarrow} \mathsf{Id}$ and counit $\mathsf{counit} \in \mathsf{Id} \dot{\leftarrow} FG$. Define $M = GF$, $\mu = G\mathsf{counit}_F$ and $\eta = \mathsf{unit}$. Then (M,μ,η) is a monad over $\mathcal{D}$.

Proof We first check the typing of μ, the check for η being trivial.

$$
\begin{array}{ll}
 & \mu \in M \dot{\leftarrow} MM \\
\equiv & \quad \{ \quad \text{definitions of } \mu \text{ and } M \quad \} \\
 & G\mathsf{counit}_F \in GF \dot{\leftarrow} GFGF \\
\Leftarrow & \quad \{ \quad G \text{ and } F \text{ are functors; naturality} \quad \} \\
 & \mathsf{counit} \in \mathsf{Id} \dot{\leftarrow} FG \quad .
\end{array}
$$

In checking the monad laws, we omit the subscripts. First,

$$
\begin{array}{ll}
 & \mu \circ \mu \;=\; \mu \circ M\mu \\
\equiv & \quad \{ \quad \text{definitions of } \mu \text{ and } M \quad \} \\
 & G\mathsf{counit} \circ G\mathsf{counit} \;=\; G\mathsf{counit} \circ GFG\mathsf{counit} \\
\equiv & \quad \{ \quad G \text{ is a functor} \quad \} \\
 & G(\mathsf{counit} \circ \mathsf{counit}) = G(\mathsf{counit} \circ FG\mathsf{counit}) \\
\Leftarrow & \quad \{ \quad \textsc{Leibniz} \quad \} \\
 & \mathsf{counit} \circ \mathsf{counit} \;=\; \mathsf{counit} \circ FG\mathsf{counit} \\
\equiv & \quad \{ \quad \mathsf{counit} \in \mathsf{Id} \dot{\leftarrow} FG \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

Next,

$$
\begin{array}{ll}
 & \mu \circ \eta = \mathsf{id} \\
\equiv & \quad \{ \quad \text{definitions of } \mu \text{ and } \eta \quad \} \\
 & G\mathsf{counit} \circ \mathsf{unit} \;=\; \mathsf{id} \\
\equiv & \quad \{ \quad \mathsf{unit}\text{-}\textsc{inverse} \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

Finally,

$$
\begin{array}{ll}
 & \mu \circ M\eta \;=\; \mathsf{id} \\
\equiv & \quad \{ \quad \text{definitions of } \mu, M \text{ and } \eta \quad \}
\end{array}
$$

$$
\begin{array}{ll}
 & G\mathsf{counit} \circ GF\mathsf{unit} \;=\; \mathsf{id} \\
\Leftarrow & \quad \{ \quad G \text{ is a functor} \quad \} \\
 & \mathsf{counit} \circ F\mathsf{unit} \;=\; \mathsf{id} \\
\equiv & \quad \{ \quad \mathsf{counit\text{-}INVERSE} \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

□

We shall see later that the reverse is also true: every monad gives rise to an adjunction.

7.2 Examples of monads

Example 7.4 (The effect monad) A really purely functional language can have no side-effects; in particular it cannot have the side-effect of producing output on the screen or otherwise. In practice this is not very useful for a programming language. So, while pretending that some function has typing $x \leftarrow y$, if it can have an effect on the screen its typing is perhaps more appropriately given as $(x \times E) \leftarrow y$, in which the elements of the object E are 'effects' (on the screen). We need a way to combine subsequent effects into a cumulative effect, so assume that we have some monad $(E, \odot, e)$. For example, for the teletype model, E could be lists of lines, $\odot$ append ($s \odot t = t +\!\!+ s$), and e the empty list of lines. Or E could be a set of screen-transforming functions, with $\odot$ being functional composition and e the identity transformation.

Define the functor M by $M = (x \;::\; x \times E)$, and take $\mu = ((a,u),v::(a, u \odot v))$ and $\eta = (a :: (a, e))$. We verify that (M, μ, η) is a monad.

$$
\begin{array}{ll}
 & \mu \circ \mu_M \\
= & \quad \{ \quad \text{definitions of } \mu \text{ and } M \quad \} \\
 & ((a,u),t::(a, u \odot t)) \circ (((a,u),v),w::((a,u), v \odot w)) \\
= & \quad \{ \quad \textsc{Lambda-lambda-fusion} \quad \} \\
 & (((a,u),v),w::(a, u \odot (v \odot w))) \\
= & \quad \{ \quad \text{operation } \odot \text{ of monoid is associative} \quad \} \\
 & (((a,u),v),w::(a, (u \odot v) \odot w)) \\
= & \quad \{ \quad \textsc{Lambda-lambda-fusion} \quad \} \\
 & ((a,s),w::(a, s \odot w)) \;\circ\; (((a,u),v),w::((a, u \odot v), w)) \\
= & \quad \{ \quad \text{product in } \mathsf{Fun} \quad \} \\
 & ((a,s),w::(a, s \odot w)) \;\circ\; ((a,u),v::(a, u \odot v)) \times \mathsf{id}
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \quad \text{definitions of } \mu \text{ and } M \quad \} \\
& \mu \circ M\mu \quad .
\end{array}
$$

and

$$
\begin{array}{ll}
& \mu \circ \eta_M \\
= & \{ \quad \text{definitions of } \mu,\ \eta \text{ and } M \quad \} \\
& ((a,u),v::(a,u \odot v)) \circ (a,u::((a,u),e)) \\
= & \{ \quad \textsc{Lambda-lambda-fusion} \quad \} \\
& (a,u::(a,u \odot e)) \\
= & \{ \quad \odot \text{ has neutral element } e \quad \} \\
& (a,u::(a,u)) \\
= & \{ \quad \mathsf{id}_M \text{ in } \mathsf{Fun} \quad \} \\
& \mathsf{id}_M \\
= & \{ \quad \text{idem} \quad \} \\
& (a,u::(a,u)) \\
= & \{ \quad \odot \text{ has neutral element } e \quad \} \\
& (a, u :: (a, e \odot u)) \\
= & \{ \quad \textsc{Lambda-lambda-fusion} \quad \} \\
& ((a,v),u::(a,v \odot u)) \circ (a, u :: ((a,e),u)) \\
= & \{ \quad \text{product in } \mathsf{Fun} \quad \} \\
& ((a,v),u::(a,v \odot u)) \circ (a :: (a,e)) \times \mathsf{id} \\
= & \{ \quad \text{definitions of } \mu,\ \eta \text{ and } M \quad \} \\
& \mu \circ \eta \quad .
\end{array}
$$

□

Remark 7.5 The term *monad* originates from this particular *monoid*-based example.

□

Example 7.6 (The exception monad) A way of modelling partial functions in Fun is to assume the existence of an object (set or type) E whose elements are 'exceptions' or 'error messages'. A possible choice is $E = 1$. A partial function to x from y can then be modelled as a total function $f \in x + E \leftarrow y$.

Define the functor M (for 'Maybe') by $M = (x :: x{+}E)$, and take $\mu = \mathsf{id} \triangledown \mathsf{inr}$ and $\eta = \mathsf{inl}$. Then (M,μ,η) is a monad. The verification of the monad laws is left as an exercise to the reader.

□

Example 7.7 (The state-transformer monad) Think of S as being a set of states. In a category with exponents each pair $(\times S, {\leftarrow}S)$ forms an adjunction. The unit is $\mathsf{pair} = (u :: (v :: (u,v)))$ and the counit is $\mathsf{eval} = (f, x :: f.x)$. So we obtain a monad (M,μ,η) by defining

$$M = ({\leftarrow}S){\circ}(\times S) = (x :: (x{\times}S){\leftarrow}S) \quad ,$$

$$\mu = \mathsf{eval}{\leftarrow}\mathsf{id} = (h :: (f, x :: f.x){\circ}h) \quad ,$$

$$\eta = \mathsf{pair} = (u :: (v :: (u,v))) \quad .$$

This monad is useful for modelling state-transforming programs, as in imperative programming, in a functional world. The idea is suggested by the following schematic representation of a program as a function:

```
prog input starting_state = (output, new_state)
```

Note that if the type of `output` is x and that of `input` is y, the type of `prog` is $Mx \leftarrow y$.

□

7.3 Kleisli composition

Given two functions $f \in Mx \leftarrow y$ and $g \in My \leftarrow z$ for any of these monads, we would like to be able to keep up the pretense that they have typings $x{\leftarrow}y$ and $y{\leftarrow}z$, respectively, and compose them accordingly. The 'extra' part given by M should then be silently connected in the composition in an appropriate way: for the effect monad by combining the effects; for the exception monad by transferring the first exception raised, if any; for the state-transformer monad by feeding the new state as starting state into the next 'program'. There is a generic construction for accomplishing this.

Definition 7.8 (Kleisli arrow) A *Kleisli arrow* for a monad (M,μ,η) over a category $\mathcal{C}$ is: a $\mathcal{C}$-arrow with typing $Mx \leftarrow y$ for some $x, y{\in}\mathcal{C}$.

□

Definition 7.9 (Kleisli composition) Given two Kleisli arrows $f \in Mx \leftarrow y$ and $g \in My \leftarrow z$ for some monad (M,μ,η) (so $M(\mathsf{dom}.f) = \mathsf{cod}.g$), the *Kleisli composition* of f and g, denoted by $f \diamond g$, is defined as:

$$f \diamond g \;=\; \mu \circ Mf \circ g \in Mx \leftarrow z \quad .$$

□

Lemma 7.10 Kleisli composition is associative, and has η as neutral element.

Proof We show in fact that the claim of the lemma is *equivalent* to the monad laws. First, for all suitably typed Kleisli arrows f, g and h:

$$\begin{array}{ll} & (f \diamond g) \diamond h = f \diamond (g \diamond h) \\ \equiv & \quad \{ \quad \text{definition of } \diamond \quad \} \\ & \mu \circ M(\mu \circ Mf \circ g) \circ h \;=\; \mu \circ Mf \circ \mu \circ Mg \circ h \\ \equiv & \quad \{ \quad M \text{ is a functor} \quad \} \\ & \mu \circ M\mu \circ MMf \circ Mg \circ h \;=\; \mu \circ Mf \circ \mu \circ Mg \circ h \\ \equiv & \quad \{ \quad \mu \in M \dot{\leftarrow} MM \quad \} \\ & \mu \circ M\mu \circ MMf \circ Mg \circ h \;=\; \mu \circ \mu \circ MMf \circ Mg \circ h \\ \equiv & \quad \{ \quad \Leftarrow\text{: Leibniz; } \Rightarrow\text{: take } f,g,h \;:=\; \mathsf{id},\mathsf{id},\mathsf{id} \quad \} \\ & \mu \circ M\mu \;=\; \mu \circ \mu \quad . \end{array}$$

Next:

$$\begin{array}{ll} & f \diamond \eta = f \;\wedge\; \eta \diamond f = f \\ \equiv & \quad \{ \quad \text{definition of } \diamond \quad \} \\ & \mu \circ Mf \circ \eta \;=\; f \;\wedge\; \mu \circ M\eta \circ f \;=\; f \\ \equiv & \quad \{ \quad \eta \in M \dot{\leftarrow} \mathsf{Id} \quad \} \\ & \mu \circ \eta \circ f = f \;\wedge\; \mu \circ M\eta \circ f \;=\; f \\ \equiv & \quad \{ \quad \Leftarrow\text{: Leibniz; } \Rightarrow\text{: take } f \;:=\; \mathsf{id} \quad \} \\ & \mu \circ \eta = \mathsf{id} \;\wedge\; \mu \circ M\eta \;=\; \mathsf{id} \quad . \end{array}$$

□ This gives us all we need to construct a category.

Definition 7.11 (Kleisli category) The *Kleisli category* for a monad (M,μ,η) over a base category $\mathcal{C}$ is the category $\mathcal{K}$ defined as follows:

Objects: Those of $\mathcal{C}$.

Arrows: $x \overset{\mathcal{K}}{\leftarrow} y$ consists of the Kleisli arrows $Mx \overset{\mathcal{C}}{\leftarrow} y$.

Composition: Kleisli composition.

Identities: η.

□

This gives us a direct simulation of, e.g., the category Par in terms of Fun by using the exception monad, or of state-transforming 'functions', as in imperative programming languages, in terms of Fun.

Example 7.12 Let us work out Kleisli composition for the state-transformer monad in Fun.

$$
\begin{array}{ll}
 & f \diamond g \\
= & \quad \{ \quad \text{definition of } \diamond \quad \} \\
 & \mu \circ Mf \circ g \\
= & \quad \{ \quad \text{state-transformer monad} \quad \} \\
 & \mathsf{eval} \leftharpoonup \mathsf{id} \circ (f \times \mathsf{id}) \leftharpoonup \mathsf{id} \circ g \\
= & \quad \{ \quad \leftharpoonup \mathsf{id} \text{ is a functor} \quad \} \\
 & (\mathsf{eval} \circ f \times \mathsf{id}) \leftharpoonup \mathsf{id} \circ g \\
= & \quad \{ \quad \textsc{Lambda-abstraction} \text{ (nested)} \quad \} \\
 & (v :: (s :: (((\mathsf{eval} \circ f \times \mathsf{id}) \leftharpoonup \mathsf{id} \circ g).v).s)) \\
= & \quad \{ \quad \textsc{Fun-}\circ\textsc{-comp} \quad \} \\
 & (v :: (s :: (((\mathsf{eval} \circ f \times \mathsf{id}) \leftharpoonup \mathsf{id}).(g.v)).s)) \\
= & \quad \{ \quad \text{definition of } \leftharpoonup \text{ in } \mathsf{Fun} \quad \} \\
 & (v :: (s :: (\mathsf{eval} \circ f \times \mathsf{id} \circ g.v).s)) \\
= & \quad \{ \quad \textsc{Fun-}\circ\textsc{-comp} \quad \} \\
 & (v :: (s :: \mathsf{eval} \;.\; ((f \times \mathsf{id}).((g.v).s)))) \\
= & \quad \{ \quad \text{where-abstraction} \quad \} \\
 & (v :: (s :: \mathsf{eval} \;.\; ((f \times \mathsf{id}).(u,t)) \text{ where } (u,t) = (g.v).s)) \\
= & \quad \{ \quad \text{Cartesian product} \quad \} \\
 & (v :: (s :: \mathsf{eval} \;.\; (f.u, t) \text{ where } (u,t) = (g.v).s)) \\
= & \quad \{ \quad \text{definition of } \mathsf{eval} \text{ in } \mathsf{Fun} \quad \}
\end{array}
$$

$$(v :: (s :: (f.u).t \text{ where } (u,t) = (g.v).s)) \quad .$$

Here u corresponds to an intermediate result, while t is an intermediate state. Kleisli composition tells us how to fit everything properly together.

□

Theorem 7.13 For any monad there is an adjunction between the Kleisli category and the base category for that monad.

Proof We use (M,μ,η) to name the components of the monad, and $\mathcal{C}$ and $\mathcal{K}$ for the base and the Kleisli category. If we guess the correct adjungates, all other ingredients follow. Since Kleisli arrows are in fact arrows of $\mathcal{C}$, we have

$$f \in x \xleftarrow{\mathcal{K}} y \equiv f \in Mx \xleftarrow{\mathcal{C}} y \quad .$$

Pattern matching against the typing rules

$$\lceil g \rceil \in x \xleftarrow{\mathcal{K}} Fy \Leftarrow g \in Gx \xleftarrow{\mathcal{C}} y$$

$$f \in x \xleftarrow{\mathcal{K}} Fy \Rightarrow \lfloor f \rfloor \in Gx \xleftarrow{\mathcal{C}} y$$

leads us to suspect that the adjungates are both the identity mapping, and that the action of F on objects is the identity, while that of G is the same as M. Then, by unit-DEF and counit-DEF, the unit and counit are the identity arrow from the other category: $\mathsf{unit} = \eta$ and $\mathsf{counit} = \mathsf{id}$. Theorem 4.20 tells us now how to construct the adjoints:

$$Fg = \eta \circ g \text{ and } Gf \;=\; f \diamond \mathsf{id} \;=\; \mu \circ Mf \quad .$$

We verify the functor requirements for F:

$$\begin{array}{ll}
 & Ff \diamond Fg \;=\; F(f \circ g) \\
\equiv & \quad \{ \quad \text{construction of } F \quad \} \\
 & (\eta \circ f) \diamond (\eta \circ g) = \eta \circ f \circ g \\
\equiv & \quad \{ \quad \text{definition of } \diamond \quad \} \\
 & \mu \circ M(\eta \circ f) \circ \eta \circ g \;=\; \eta \circ f \circ g \\
\equiv & \quad \{ \quad \text{lhs: } M \text{ is a functor; rhs: } \eta \in M \dot{\leftarrow} \mathsf{Id} \quad \} \\
 & \mu \circ M\eta \circ Mf \circ \eta \circ g \;=\; Mf \circ \eta \circ g \\
\Leftarrow & \quad \{ \quad \text{Leibniz} \quad \} \\
 & \mu \circ M\eta \;=\; \mathsf{id} \\
\equiv & \quad \{ \quad \text{monad laws} \quad \} \\
 & \mathsf{true} \quad ,
\end{array}$$

and, of course, $F\mathsf{id} = \eta$.

All that is needed to apply (the dual of) Theorem 4.17 is to verify a right-fusion rule:

$$
\begin{array}{ll}
 & \lceil h \rceil \diamond Fg \;=\; \lceil h \circ g \rceil \\
\equiv & \quad \{ \quad \text{the adjungates are identity mappings; construction of } F \quad \} \\
 & h \diamond (\eta \circ g) = h \circ g \\
\equiv & \quad \{ \quad \text{definition of } \diamond \quad \} \\
 & \mu \circ Mh \circ \eta \circ g \;=\; h \circ g \\
\equiv & \quad \{ \quad \eta \in M \dot{\leftarrow} \mathsf{Id} \quad \} \\
 & \mu \circ \eta \circ h \circ g = h \circ g \\
\Leftarrow & \quad \{ \quad \textsc{Leibniz} \quad \} \\
 & \mu \circ \eta = \mathsf{id} \\
\equiv & \quad \{ \quad \text{monad laws} \quad \} \\
 & \mathsf{true} \quad .
\end{array}
$$

□

Exercise 7.14 Give the monad for the adjunction between Rel and Set.

□

Exercise 7.15 Express Kleisli composition for the exception monad in a functional programming language.

□

Exercise 7.16 Using the obvious correspondence between $f \in x \xleftarrow{\mathsf{Par}} y$ and $f \in x{+}1 \xleftarrow{\mathsf{Set}} y$, and the adjunction for the exception monad, work out an adjunction between Par and Set.

□

Exercise 7.17 Given a category $\mathcal{C}$ with exponents, give the monad for the adjunction $(R\leftharpoonup -, R\leftharpoonup -)$ between $\mathcal{C}^{op}$ and $\mathcal{C}$ (see Exercise 5.2 and note the warning there). Express Kleisli composition for this monad, the so-called *continuation-passing* monad, in a functional programming language. (The idea is that a value v can be represented by

$@v = (f :: f.v)$. Think of f as standing for the future computation to which v will be subjected. If there is no continuation of the computation, we take $f = \mathsf{id}$, obtaining $(@v).\mathsf{id} = v$. If v has type x, putting R for the result type of the continuation, f is required to have typing $R \leftarrow x$, and so $@v$ (as a value) has type $R \leftarrow (R \leftarrow x)$.)

□

Exercise 7.18 Given a monad over a category $\mathcal{C}$ and an adjunction (F, G) between $\mathcal{C}$ and $\mathcal{D}$, the following construction combines them into a monad over $\mathcal{D}$: First, decompose the monad (using Theorem 7.13) into an adjunction between $\mathcal{K}$ and $\mathcal{C}$. Next, use Theorem 4.21 to compose this adjunction with (F, G), thus obtaining an adjunction between $\mathcal{K}$ and $\mathcal{D}$. Finally, use Theorem 7.3 to obtain a monad over $\mathcal{D}$.

Work out the details, and phrase the result as a theorem. Note that the intermediate construction involving $\mathcal{K}$ dissolves.

□

Exercise 7.19 Using the results of the previous exercise, combine the exception monad with the adjunction $(\times S, \leftarrow S)$. Work out the details of Kleisli composition for the new monad, which tells us how to handle exceptions for state-transforming functions.

□

Further Reading

M. Arbib, E. Manes (1975). *Arrows, Structures and Functors*: *The Categorical Imperative.* Academic Press.

Andrea Asperti, Giuseppe Longo (1991). *Categories, Types, and Structures.* The MIT Press.

Michael Barr, Charles Wells (1990). *Category Theory for Computing Science.* Prentice Hall.

Richard Bird, Oege de Moor (1996). *The Algebra of Programming.* (To appear).

J. Lambek, P.J. Scott (1986). *Introduction to Higher Order Categorical Logic.* Cambridge University Press.

S. Mac Lane (1971). *Categories for the Working Mathematician.* Graduate texts in mathematics **5**, Springer-Verlag.

Benjamin C. Pierce (1991). *Basic Category Theory for Computing Scientists.* The MIT Press.

D.E. Rydeheard, R.M. Burstall (1988). *Computational Category Theory.* Prentice Hall.

R.F.C. Walters (1991). *Categories and Computer Science.* Cambridge University Press.

Index