Why Adjunctions Matter—A Functional Programmer Perspective

José Nuno Oliveira(✉)

High Assurance Software Laboratory, INESC TEC and University of Minho, Braga, Portugal
jno@di.uminho.pt

Abstract. For the average programmer, *adjunctions* are (if at all known) more respected than loved. At best, they are regarded as an algebraic device of theoretical interest only, not useful in common practice.

This paper is aimed at showing the opposite: that adjunctions underlie most of the work we do as programmers, in particular those using the functional paradigm. However, functions alone are not sufficient to express the whole spectrum of programming, with its dichotomy between *specifications*—*what* is (often vaguely) required—and *implementations*—*how* what is required is (hopefully well) implemented. For this, one needs to extend functions to *relations*.

Inspired by the pioneering work of Ralf Hinze on "adjoint (un)folds", the core of the so-called (relational) Algebra of Programming is shown in this paper to arise from adjunctions. Moreover, the paper also shows how to calculate recursive programs from specifications expressed by Galois connections—a special kind of adjunction.

Because Galois connections are easier to understand than adjunctions in general, the paper adopts a tutorial style, starting from the former and leading to the latter (a path usually not followed in the literature). The main aim is to reconcile the functional programming community with a concept that is central to software design as a whole, but rarely accepted as such.

Keywords: Algebra of programming · Programming from specifications · Adjunctions

> *"(...) and Jim Thatcher proposed the name* ADJ *as a (terrible) pun on the title of the book that we had planned to write (...)* [recalling] *that* adjointness *is a very important concept in category theory (...)"*
> — Joseph A. Goguen, *Memories of ADJ*, EATCS nr. 36, 1989

1 Context

The notion of an *algebraic data type* is central to the theoretical advances in computer science since the 1980s—a "vintage decade" that turned *program semantics*

A. Madeira and M. A. Martins (Eds.): WADT 2022, LNCS 13710, pp. 25–59, 2023.
https://doi.org/10.1007/978-3-031-43345-0_2

into a branch of scientific knowledge [6,7], and the trend in which the WADT series of workshops arose. In particular, the 'ADJ group' promoted what can be regarded as the first effective use of category theory in computer science, centered upon notions such as *initiality*, *freeness* and *institution* [7,10].

The categorial concept of an *adjunction* [14] underlies all such techniques and is so important that the ADJ group decided to carve it in their own acronym, as quoted above. However, for the average programmer *adjunctions* are regarded (if at all known) as working at the meta-level only [10]. In fact, explicit use of adjunctions as an instrumental device for abstract reasoning in programming is relatively rare. Less rare is the use of Galois connections (a special case of adjunction) to structure relational algebra techniques [1,2], but even so the topic is not mainstream.

This contribution to the WADT series tries to show how relevant adjunctions are in explaining many things we do as programmers. In a tutorial flavour, it will try to show how practical adjunctions are by revealing their "chemistry in action".

It is common practice to introduce the adjunction concept first, and only then refer to what is regarded as a modest instance: the Galois connection (GC). That is, general concept first, instances later. Below we go in the opposite direction, which is easier to grasp: GCs are presented first, together with a set of examples and applications. Only after these are understood and appreciated does one step into full generality.

2 Galois Connections

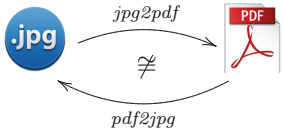
Things in everyday life often come "in pairs", as dichotomies such as e.g. *good/bad, action/reaction, the left/the right, lower/upper, easy/hard* and so on. In a sense, each pair defines itself: one element of the pair exists... because the other also exists, and is its *opposite* (i.e. *antithesis*). Despite the circularity, common everyday language survives over such dualities.

Perfect Antithesis. The perfect antithesis (opposition, inversion) is the *bijection* or *isomorphism*. For instance, *multiplication* and *division* are inverses of each other in the positive reals: $\frac{x}{y} \times y = x$ and $\frac{x \times y}{y} = x$. That is, there is no loss of information when dividing or multiplying. In general, f and g such that

$$B \underset{f}{\overset{g}{\cong}} A \quad \begin{cases} f\ (g\ b) = b \\ g\ (f\ a) = a \end{cases} \tag{1}$$

hold are termed *isomorphisms* and regarded as *lossless* transformations.

Imperfect Antithesis. However, data transformations in practice are *lossy*, e.g.



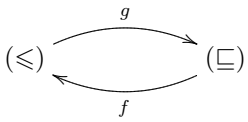
$$jpg2pdf \cdot pdf2jpg \neq id$$
$$pdf2jpg \cdot jpg2pdf \neq id$$

even though our eyes can hardly spot the difference in most cases.

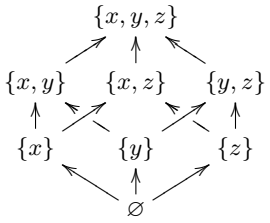
It is often the case that loss of information in such imperfect inversions can be expressed in this way,

$$\begin{cases} f\ (g\ b) \leqslant b \\ a \sqsubseteq g\ (f\ a) \end{cases} \tag{2}$$

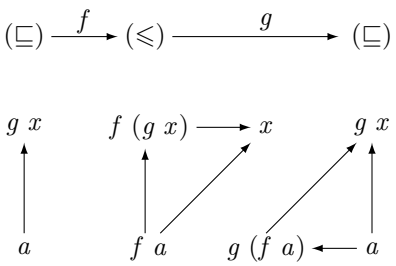
telling "how bad" each *"round trip"* is. This relies on under and over *approximations* captured by two *preorders* (i.e. reflexive and transitive relations),



where f and g are assumed monotonic.



Handling Approximations. Let us write the arrow $x \xrightarrow{(\leqslant)} y$ (resp. $x \xrightarrow{(\sqsubseteq)} y$) to denote $x \leqslant y$ (resp. $x \sqsubseteq y$) in (2). We shall drop the ordering symbols, e.g. simply writing $x \longrightarrow y$, wherever these are clear from the context, as in the Hasse diagram aside. This arrow-notation will enable us to express our reasoning graphically, as in the following diagram:



(3)

Let us "parse" this diagram without rushing: arrow $a \to g\ x$ means $a \sqsubseteq g\ x$. By monotonicity we get $f\ a \leqslant f\ (g\ x)$, i.e. arrow $f\ a \to f\ (g\ x)$. From (2) we get $f\ (g\ x) \to x$ and, by transitivity ("composition" of these two arrows) we get $f\ a \to x$. We are done with the first "triangle".

The triangle on the right starts with $g\ (f\ a) \to g\ x$, which follows by monotonicity from $f\ a \to x$ in the first one. Again from (2) and transitivity we get $a \to g\ x$ where we started from. Summing up:

$$a \sqsubseteq g\ x \Rightarrow f\ a \leqslant x \Rightarrow a \sqsubseteq g\ x$$

By circular implication, the equivalence

$$f\ a \leqslant x \quad \Leftrightarrow \quad a \sqsubseteq g\ x \tag{4}$$

holds for any a and x, and we say that f and g are *Galois connected*, writing $f \dashv g$ to declare so. Terminology: f is said to be the *lower* (a.k.a. *left*) adjoint of the connection and g the *upper* (a.k.a. *right*) adjoint. The intuition behind this terminology is captured by the superlatives in the following interpretation of (4):

- $f\ a$—*lowest* x such that $a \sqsubseteq g\ x$
- $g\ x$—*greatest* a such that $f\ a \leqslant x$.

Did we write *"superlatives"*? Note that we have plenty of these in *software requirements*, e.g.

- ... the *largest* prefix of x with at most n elements (i.e. the meaning of function $\mathsf{take}\ n\ x$ in Haskell)
- ... the *largest* number that multiplied by y is at most x (i.e. the meaning of integer division $x \div y$).

Back to the perfect/imperfect dichotomies above, compare numeric division in the reals ($\mathbb{R}$), for $y \neq 0$,

$$a \times y = x \quad \Leftrightarrow \quad a = x \,/\, y$$

—an *isomorphism*—with (whole) division in the natural numbers ($\mathbb{N}_0$),

$$a \times y \leqslant x \quad \Leftrightarrow \quad a \leqslant x \div y \tag{5}$$

—a Galois *connection*: $(\times y) \dashv (\div y)$.

3 The Easy and the Hard

It is the experience of every school child that $x \div y$ is much harder to calculate by hand than $x \times y$. Indeed, division is perhaps the very first "hard" problem (algorithm) that children encounter in their basic maths education. Interestingly, $(\times y) \dashv (\div y)$ bears a simple message:

hard $(\div y)$ is explained by *easy* $(\times y)$.

This pattern extends to program specifications, recall

$\mathsf{take}\ n\ xs$ should yield the *longest* possible *prefix* of xs not exceeding n in *length*

from above. The corresponding *formal specification*,

$$\underbrace{\mathsf{length}\ ys \leqslant n \wedge ys \sqsubseteq xs}_{easy} \quad \Leftrightarrow \quad \underbrace{ys \sqsubseteq \mathsf{take}\ n\ xs}_{hard} \tag{6}$$

is another GC, where $(\sqsubseteq)$ is the list-prefix partial ordering.[1] Many other examples can be found in programming, for instance:

[1] The reader may wonder how (6) fits in the frame of (4). To see this, let us write (6) in the *uncurried* format: $(\mathsf{length}\ ys, ys)\ ((\leqslant) \times (\sqsubseteq))\ (n, xs) \ \Leftrightarrow \ ys \sqsubseteq \widehat{\mathsf{take}}\ (n, xs)$ where, in general, $\widehat{g}\ (a, b) = g\ a\ b$. Thus $f\ x = (\mathsf{length}\ x, x)$, $g = \widehat{\mathsf{take}}$ and the product ordering has the expected relational meaning, which in general is: $(x, y)\ (R \times S)\ (a, b) \ \Leftrightarrow \ x\ R\ a \wedge y\ S\ b$.

– The function takeWhile p xs should yield the *longest prefix* of xs whose elements all satisfy predicate p.
– The function filter p xs should yield the *longest sublist* of xs all elements of which satisfy predicate p.

4 Indirect Equality

Back to $(\times y) \dashv (\div y)$, consider the following, well-known *implementation* of integer division:

$$x \div y = \textbf{if } x \geqslant y \textbf{ then } 1 + (x \ominus y) \div y \textbf{ else } 0 \tag{7}$$

Can this *implementation* be derived from the *specification* (5)? Note that, because subtraction in $\mathbb{N}_0$ is not invertible, one needs to resort to "truncated subtraction" (written $x \ominus y$ below) which, as one might suspect, is an adjoint of another GC in $\mathbb{N}_0$:

$$a \ominus b \leqslant x \;\Leftrightarrow\; a \leqslant x + b \tag{8}$$

To address the question above, one needs yet another brick in the wall: the principle of *indirect equality*, valid for any partial order [2]:

$$a = b \;\Leftrightarrow\; \langle \forall\ z \;::\; z \leqslant a \;\Leftrightarrow\; z \leqslant b \rangle \tag{9}$$

This principle of indirect equality blends nicely with GCs, as the following calculation sketch suggests:

$$\begin{array}{ll} & z \leqslant g\ a \\ \Leftrightarrow & \quad \{\ \ldots\ \} \\ & \ldots \text{(go to the } \textit{easy} \text{ side, do things there and come back)} \\ \Leftrightarrow & \quad \{\ \ldots\ \} \\ & z \leqslant \ldots g \ldots a' \ldots \\ :: & \quad \{\ \text{indirect equality}\ \} \\ & g\ a = \ldots g \ldots a' \ldots \end{array}$$

Note how a difficult g can in principle be calculated by *going to the* easy *side of the specification GC and coming back*.

As a simple example of using (9), let us calculate $x \div y$ (5) in case $x \geqslant y$ holds:

$$
\begin{array}{ll}
 & z \leqslant x \div y \\
\Leftrightarrow & \quad \{\ (\times y) \dashv (\div y) \text{ and } (x \ominus y) + y = x \text{ for } x \geqslant y\ \} \\
 & z \times y \leqslant (x \ominus y) + y \\
\Leftrightarrow & \quad \{\ (\ominus y) \dashv (+y)\ \} \\
 & (z \times y) \ominus y \leqslant x \ominus y \\
\Leftrightarrow & \quad \{\ \text{factoring out } y \text{ works also for } \ominus\ \} \\
 & (z \ominus 1) \times y \leqslant x \ominus y \\
\Leftrightarrow & \quad \{\ \text{chain the two } GC\text{s}\ \} \\
 & z \leqslant 1 + (x \ominus y) \div y \\
:: & \quad \{\ \text{recursive branch of (7) calculated thanks to indirect equality (9)}\ \} \\
 & x \div y = 1 + (x \ominus y) \div y \\
\Box &
\end{array}
$$

The other case $(x < y)$ also stems from (5):

$$
\begin{array}{ll}
 & x < y \\
\Leftrightarrow & \quad \{\ \text{trivial}\ \} \\
 & \neg\ (y \leqslant x) \\
\Leftrightarrow & \quad \{\ (5), \text{ for } a := 1\ \} \\
 & 1 > x \div y \\
\Leftrightarrow & \quad \{\ \text{trivial (in } \mathbb{N}_0)\ \} \\
 & x \div y = 0 \\
\Box &
\end{array}
$$

Altogether, we have proven that the recursive implementation (7) of $x \div y$ is correct with respect to its GC specification (5).

5 GCs as Formal Specifications

Let us now try a similar exercise for take, formally specified by (6). This time, however, no known implementation is assumed. Moreover, we wish to show how to draw properties from specification (6) *before* implementing take, i.e. *without* knowing anything about its actual implementation.

For instance, what happens if we chain two take*s* in a row, $(\mathsf{take}\ m) \cdot (\mathsf{take}\ n)$? We *calculate*:

$$
\begin{array}{ll}
 & ys \sqsubseteq \mathsf{take}\ m\ (\mathsf{take}\ n\ xs) \\
\Leftrightarrow & \quad \{\ \text{GC (6)}\ \} \\
 & \mathsf{length}\ ys \leqslant m \wedge ys \sqsubseteq \mathsf{take}\ n\ xs \\
\Leftrightarrow & \quad \{\ \text{again GC (6)}\ \} \\
 & \mathsf{length}\ ys \leqslant m \wedge \mathsf{length}\ ys \leqslant n \wedge ys \sqsubseteq xs \\
\Leftrightarrow & \quad \{\ \text{min GC: } a \leqslant x \wedge a \leqslant y \ \Leftrightarrow\ a \leqslant x\ `min`\ y\ \} \\
 & \mathsf{length}\ ys \leqslant (m\ `min`\ n) \wedge ys \sqsubseteq xs \\
\Leftrightarrow & \quad \{\ \text{again GC (6)}\ \} \\
 & ys \leqslant \mathsf{take}\ (m\ `min`\ n)\ xs \\
:: & \quad \{\ \text{indirect equality (9)}\ \} \\
 & \mathsf{take}\ m\ (\mathsf{take}\ n\ xs)) = \mathsf{take}\ (m\ `min`\ n)\ xs
\end{array}
$$

Note the fully *deductive* calculation—no recursion, no *induction*. There could be none, in fact, because we have no implementation of take yet! Calculating this is the subject of the reasoning that follows.

A quick inspection of (6) invites us to consider the cases $n = 0$ and $xs = [\,]$ because they trivialize the *easy side* of the GC, as is easy to show. Case $n = 0$ first:

$$
\begin{array}{ll}
 & ys \sqsubseteq \mathsf{take}\ 0\ xs \\
\Leftrightarrow & \quad \{\ \text{GC}\ \} \\
 & \mathsf{length}\ ys \leqslant 0 \wedge ys \sqsubseteq xs \\
\Leftrightarrow & \quad \{\ \mathsf{length}\ ys \leqslant 0 \ \Leftrightarrow\ ys = [\,]\ \} \\
 & ys = [\,] \\
\Leftrightarrow & \quad \{\ \text{antisymmetry of } (\sqsubseteq);\ [\,] \sqsubseteq ys \text{ holds for any } ys\ \} \\
 & ys \sqsubseteq [\,] \\
:: & \quad \{\ \text{indirect equality}\ \} \\
 & \mathsf{take}\ 0\ xs = [\,]
\end{array}
$$

Now case $xs = [\,]$:

$$
\begin{array}{ll}
& ys \sqsubseteq \mathsf{take}\ n\ [\,] \\
\Leftrightarrow & \quad \{\ \text{GC}\ \} \\
& \mathsf{length}\ ys \leqslant n \wedge ys \sqsubseteq [\,] \\
\Leftrightarrow & \quad \{\ ys \sqsubseteq [\,]\ \Leftrightarrow\ ys = [\,];\ \mathsf{length}\ [\,] = 0\ \} \\
& ys = [\,] \\
\Leftrightarrow & \quad \{\ \text{antisymmetry of } (\sqsubseteq);\ [\,] \sqsubseteq ys \text{ holds for any } ys\ \} \\
& ys \sqsubseteq [\,] \\
:: & \quad \{\ \text{indirect equality}\ \} \\
& \mathsf{take}\ n\ [\,] = [\,]
\end{array}
$$

Thus we get the base cases:

$$
\begin{array}{l}
\mathsf{take}\ 0\ \_ = [\,] \\
\mathsf{take}\ \_\ [\,] = [\,]
\end{array}
$$

By pattern matching, the remaining case is $\mathsf{take}\ (n+1)\ (h : xs)$. The following fact about list-prefixing,

$$s \sqsubseteq (h : t)\ \Leftrightarrow\ s = [\,]\ \vee\ \langle \exists\ s'\ ::\ s = (h : s') \wedge s' \sqsubseteq t \rangle \tag{10}$$

will be required. This property is quite obvious but... where does it come from?[2] Let us accept it for the moment, leaving the answer to the question to Sect. 16 later on. Once again, we calculate:

$$
\begin{array}{ll}
& ys \sqsubseteq \mathsf{take}\ (n+1)\ (h : xs) \\
\Leftrightarrow & \quad \{\ \text{GC (6) ; prefix (10)}\ \} \\
& \mathsf{length}\ ys \leqslant n+1 \wedge (ys = [\,] \vee \langle \exists\ ys'\ ::\ ys = (h : ys') \wedge ys' \sqsubseteq xs \rangle) \\
\Leftrightarrow & \quad \{\ \text{distribution ; } \mathsf{length}\ [\,] = 0 \leqslant n+1\ \} \\
& ys = [\,] \vee \langle \exists\ ys'\ ::\ ys = (h : ys') \wedge \mathsf{length}\ ys \leqslant n+1 \wedge ys' \sqsubseteq xs \rangle \\
\Leftrightarrow & \quad \{\ \mathsf{length}\ (h : t) = 1 + \mathsf{length}\ t\ \} \\
& ys = [\,] \vee \langle \exists\ ys'\ ::\ ys = (h : ys') \wedge \mathsf{length}\ ys' \leqslant n \wedge ys' \sqsubseteq xs \rangle \\
\Leftrightarrow & \quad \{\ \text{GC (6)}\ \} \\
& ys = [\,] \vee \langle \exists\ ys'\ ::\ ys = (h : ys') \wedge ys' \sqsubseteq \mathsf{take}\ n\ xs \rangle \\
\Leftrightarrow & \quad \{\ \text{fact (10)}\ \} \\
& ys \sqsubseteq h : \mathsf{take}\ n\ xs \\
:: & \quad \{\ \text{indirect equality over list prefixing } (\sqsubseteq)\ \} \\
& \mathsf{take}\ (n+1)\ (h : xs) = h : \mathsf{take}\ n\ xs
\end{array}
$$

[2] The question also applies to $ys \sqsubseteq [\,]\ \Leftrightarrow\ ys = [\,]$ which was taken for granted above.

By putting everything together we have an implementation of take, indeed the standard one in Haskell:

$$\begin{aligned}&\mathsf{take}\ 0\ \_ = [\,]\\&\mathsf{take}\ \_\ [\,] = [\,]\\&\mathsf{take}\ (n+1)\ (h:xs) = h : \mathsf{take}\ n\ xs\end{aligned}$$

In summary, by expressing the formal specification of a particular (e.g. recursive) function in the form of a GC, not only one can prove properties but also calculate an implementation (program) without performing inductive proofs. However, the final implementation is inductive. So, the question arises:

Where does this induction come from?

The answer is not immediate and calls for the generalization from GCs to adjunctions.

6 From GCs to Adjunctions

Recall our arrow notation $a \xrightarrow{(\leqslant)} b$ for $a \leqslant b$ in (3). In the "set of pairs" interpretation of a binary relation, one might write

$$(\leqslant)\ (a, b) = \{(a, b)\}$$

meaning that the evidence that we have that $a \leqslant b$ holds is $\{(a, b)\}$—the singleton set made of exactly the pair $(a, b) \in (\leqslant)$.

Now compare $(\leqslant)\ (a, b) = \{(a, b)\}$ with something like (broadening scope):

$$\mathbf{C}(a, b) = \{\ \text{'the things that relate } a \text{ to } b \text{ in some context } \mathbf{C}\text{'}\ \}$$

So, every such "thing" $m \in \mathbf{C}(a, b)$ acts as a *witness* of the $\mathbf{C}$-relationship between a and b. Moreover, assuming $\mathbf{C}$ (whatever this is), $m \in \mathbf{C}(a, b)$ can be written $m : a \to b$, recovering the arrow notation used before. (Notation $m : a \to b$ can also be read as telling that m is of *type* $a \to b$, a view that matches with some examples below.)

We thus land into a *category*—$\mathbf{C}$—where a and b are *objects* and m is said to be a *morphism*. In general, there will be more than one morphism between a and b, thus the need to name them. The set $\mathbf{C}(a, b)$ of all such morphisms is called a *homset*.

Categories are an extremely versatile concept, as the following instances of categories show,

$$\mathbf{C}(a, b) = \{\ \text{'matrices with } a\text{-many columns and } b\text{-many rows'}\ \}$$

or

$$\mathbf{C}(a, b) = \{\ \text{'Haskell functions from type } a \text{ to type } b\text{'}\ \}$$

or

$$\mathbf{C}(a,b) = \{\ \text{`binary relations in } a \times b\text{'}\ \}$$

among many others relevant to maths and programming.

Compared to the preorders they generalize, categories purport a "dramatic" increase in expressiveness (Fig. 1). For instance, $a \leqslant a$ always holds in a preorder (*reflexivity*), that is, homset $(\leqslant)\ (a,a)$ is non-empty. The categorial extension of reflexivity to an arbitrary category $\mathbf{C}$ also means that $\mathbf{C}\ (a,a)$ is non-empty because it always includes a special morphism, the so-called *identity* morphism id. This is written $id{:}a \to a$ wherever $\mathbf{C}$ is implicit from the context. For instance, in the category $\mathbf{S}$ of sets (objects) and functions between sets (morphisms), $id : a \to a$ is the identity function $id\ x = x$ on set a.

In turn, preorder transitivity, $a \leqslant b \wedge b \leqslant c \Rightarrow a \leqslant c$, generalizes to morphism *composition*: $m \in \mathbf{C}\ (a,b)$ and $n \in \mathbf{C}\ (b,c)$ generate $n \cdot m$ in $\mathbf{C}\ (a,c)$, called the composition of n and m, which is such that $m \cdot id = id \cdot m = m$.

What is the meaning of generalizing (3) from preorders $(\leqslant)$ and $(\sqsubseteq)$ to two categories $\mathbf{S}$ and $\mathbf{D}$? Recall our starting point,

$$\begin{cases} f\ (g\ x) \leqslant x \\ a \sqsubseteq g\ (f\ a) \end{cases}$$

which meanwhile was written thus:

$$\begin{cases} f\ (g\ x) \to x \\ a \leftarrow g\ (f\ a) \end{cases}$$

According to the correspondence of Fig. 1, monotonic functions f and g give place to *functors* F and G, respectively:[3]

$$\begin{cases} \mathsf{F}\ (\mathsf{G}\ X) \xrightarrow{\epsilon} X \\ A \xleftarrow{\eta} \mathsf{G}\ (\mathsf{F}\ A) \end{cases} \tag{11}$$

The "core" morphisms ϵ and η will be explained later. For the moment, our aim is to "replay" (3), now in the categorial setting:

$$\begin{array}{ccccc} \mathbf{D} & \xrightarrow{\mathsf{F}} & \mathbf{C} & \xrightarrow{\quad\mathsf{G}\quad} & \mathbf{D} \end{array}$$

$$\begin{array}{ccccccc} \mathsf{G}\ X & & \mathsf{F}\ (\mathsf{G}\ X) & \xrightarrow{\epsilon} & X & & \mathsf{G}\ X \\ \uparrow k & & \uparrow \mathsf{F}\ k & \nearrow \lfloor k \rfloor = h & & \nearrow \mathsf{G}\ h & \uparrow \lceil h \rceil \\ A & & \mathsf{F}\ A & & & \mathsf{G}\ (\mathsf{F}\ A) & \xleftarrow[\eta]{} A \end{array} \tag{12}$$

[3] Recall that functors are available in Haskell via *fmap*, exported by the *Functor* class. As is well known, properties $\mathsf{F}\ id = id$ and $\mathsf{F}\ (f \cdot g) = (\mathsf{F}\ f) \cdot (\mathsf{F}\ g)$ hold.

Starting from some $A \xrightarrow{k} \mathsf{G}\ X$ we obtain $\epsilon \cdot \mathsf{F}\ k$, granted by functor F (triangle on the left). We define $\lfloor k \rfloor = \epsilon \cdot \mathsf{F}\ k$ and choose to use h to denote $\lfloor k \rfloor$ in the triangle that follows. Picking h in turn, functor G grants $\mathsf{G}\ h \cdot \eta$. We define $\lceil h \rceil = \mathsf{G}\ h \cdot \eta$, that is, $\lceil h \rceil = \lceil \lfloor k \rfloor \rceil$. In case $\lceil h \rceil = k$, h and k are in a 1-to-1 correspondence and we have the isomorphism

$$\mathbf{C}\ (\mathsf{F}\ A, X) \cong \mathbf{D}\ (A, \mathsf{G}\ X) \tag{13}$$

clearly generalizing (4). In this case we say that we have an *adjunction* and that F and G are *adjoint functors*, writing $\mathsf{F} \dashv \mathsf{G}$ as before. F is called the *left adjoint* functor and G the *right adjoint* functor.

Preorder	**Category**
Object pair	Morphism
Reflexivity	Identity
Transitivity	Composition
Monotonic function	Functor
Equivalence	Isomorphism
Pointwise ordering	Natural transformation
Closure	Monad
Galois connection	Adjunction
Indirect equality	Yoneda lemma

Fig. 1. From preorders to categories.

7 Adjunctions

Another way to express (13) is given below (14), where the two adjoint functors $\mathsf{F}\colon \mathbf{D} \to \mathbf{C}$ and $\mathsf{G}\colon \mathbf{C} \to \mathbf{D}$ are renamed to $\mathsf{L}\colon \mathbf{D} \to \mathbf{C}$ and $\mathsf{R}\colon \mathbf{C} \to \mathbf{D}$, respectively, to better match with the *left* and *right* qualifiers above:

$$\begin{array}{ccc} \mathbf{C} & & \mathbf{D} \\ & \overset{\lceil - \rceil}{\curvearrowright} & \\ \mathsf{L}\ A \to X & \cong & A \to \mathsf{R}\ X \\ & \underset{\lfloor - \rfloor}{\curvearrowleft} & \end{array} \tag{14}$$

It also features the two isomorphism witnesses between the two homsets, where $\lceil h \rceil$ is called the R-*transpose* of h and $\lfloor k \rfloor$ the L-*transpose* of k. This isomorphism can be expressed in the standard way,

$$k = \lceil h \rceil \quad \Leftrightarrow \quad \lfloor k \rfloor = h \tag{15}$$

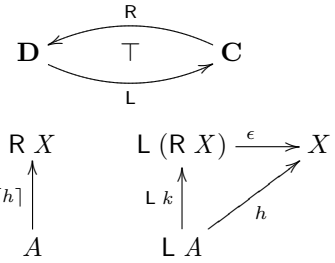
capturing how one transpose is the opposite of the other. Clearly,

$$\begin{cases} \lfloor \lceil h \rceil \rfloor = h \\ \lceil \lfloor k \rfloor \rceil = k \end{cases} \tag{16}$$

and one is back to perfect antithesis (1), but in a much richer setting, as is explained next.

From (12) we know that $\lfloor k \rfloor = \epsilon \cdot \mathsf{F}\ k$. So we can inline this in (15) and draw a diagram to depict what is going on:

$$k = \lceil h \rceil \ \Leftrightarrow\ \underbrace{\epsilon \cdot \mathsf{L}\ k}_{\lfloor k \rfloor} = h \qquad (17)$$



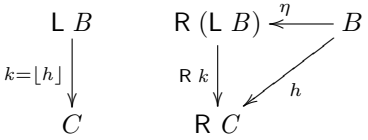
Thus we see how the adjunction $\mathsf{L} \dashv \mathsf{R}$ embodies a *universal property* that tells that $\lceil h \rceil$ is the *unique solution* of the equation $\epsilon \cdot \mathsf{L}\ k = f$ on k, for a given h. Very soon we shall see how productive (17) is. For the moment, we just inspect what happens for $k = id$. Since $\mathsf{L}\ id = id$ and $\epsilon \cdot id = \epsilon$, we get $id = \lceil \epsilon \rceil$, equivalent to

$$\epsilon = \lfloor id \rfloor \qquad (18)$$

by (15), leading to the definition of ϵ. Terminology: ϵ is called the *co-unit* of the adjunction.

Dual Formulation. The term *co-unit* suggests that there might be a *unit* somewhere in the construction—and indeed there is. Above we inlined $\lfloor k \rfloor = \epsilon \cdot \mathsf{L}\ k$ in (15). But we could do otherwise, inlining the other definition $\lceil k \rceil = \mathsf{R}\ k \cdot \eta$. This gives us a dual formulation of the adjunction,

$$k = \lfloor h \rfloor \ \Leftrightarrow\ \underbrace{\mathsf{R}\ k \cdot \eta}_{\lceil k \rceil} = h \qquad (19)$$



—compare with (17)—now telling that $\lfloor h \rfloor$ is the unique solution to equation $\mathsf{R}\ k \cdot \eta = h$. Terminology:

$$\eta = \lceil id \rceil \qquad (20)$$

is called the *unit* of the adjunction.

Natural Transformations. Morphisms such as ϵ and η are of generic type $\mathsf{F}\ X \to \mathsf{G}\ X$, where functors F and G come from and go to the same categories, the identity functor $\mathsf{F}\ X = X$ from a category to itself included. They are said to be *natural transformations*. This invites us to go back to Fig. 1, where the pointwise ordering between two functions, $f \dot{\leqslant} g$ meaning $f\ x \leqslant g\ x$ for every input x, is

said to scale up to such *natural transformations* between two functors F and G, i.e. morphisms of type $\mathsf{F}\ X \to \mathsf{G}\ X$ parametric on X.[4]

8 Examples

A rich theory arises from (17,19) which the reader can find compactly presented in laws (64) to (81) of the appendix. Before exploring such a theory, let us give some adjunction examples. Because we wish to focus on adjunctions that are relevant to programming, the examples are less general than they could be. Thus we stay within the category **S** of sets and functions in the examples that follow.

(Covariant) Exponentials: $(\_ \times K) \dashv (\_^K)$ This is perhaps the most famous adjunction, holding between category **S** and itself:

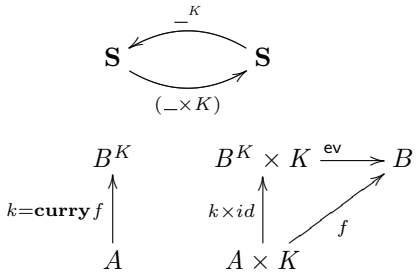
$$A \times K \to X \underset{\textbf{uncurry}}{\overset{\textbf{curry}}{\cong}} A \to X^K \quad \text{where} \begin{cases} \textbf{curry}\, f\ a\ b = f\ (a, b) \\ \textbf{uncurry}\ g\ (a, b) = g\ a\ b \end{cases}$$

This instantiates (17) for

$$\begin{cases} \mathsf{L}\ X = X \times K \\ \mathsf{R}\ X = X^K \\ \epsilon = \mathsf{ev} \end{cases} \quad \begin{cases} \lceil f \rceil = \textbf{curry}\, f \\ \lfloor f \rfloor = \textbf{uncurry}\ f \end{cases} \tag{21}$$

where $X \times K$ denotes the Cartesian product of sets X and K, X^K denotes the set of all functions of type $K \to X$ and $\mathsf{ev}\ (f, k) = f\ k$. Universal property (17) and its diagram become

$$k = \textbf{curry}\, f \ \Leftrightarrow\ \underbrace{\mathsf{ev} \cdot (k \times id)}_{\textbf{uncurry}\ k} = f$$



in this adjunction.

Associated with the Cartesian product $X \times Y$ of two sets X and Y we have the two *projections* $\pi_1 : X \times Y \to X$ and $\pi_2 : X \times Y \to Y$ which are such

[4] Thus natural transformations instantiate to the *polymorphic functions* so dear to the functional programmer, together with their *theorems for free* [23] so useful in program calculation. For the correspondence between *indirect equality* and the so-called *Yoneda lemma* (still Fig. 1) please see [5].

that $\pi_1\ (x, y) = x$ and $\pi_2\ (x, y) = y$. These projections are the essence of the adjunction that follows, which captures the categorial view of *pairing*.

Pairing: $\Delta \dashv (\times)$ In this adjunction we have

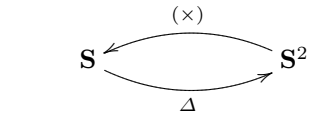
$$\begin{cases} \mathsf{L}\ X = \Delta\ X = (X, X) \\ \mathsf{R}\ (X, Y) = X \times Y \\ \epsilon = (\pi_1, \pi_2) \end{cases} \quad \begin{cases} \lceil (f, g) \rceil = \langle f, g \rangle \\ \lfloor k \rfloor = (\pi_1 \cdot k, \pi_2 \cdot k) \end{cases} \tag{22}$$

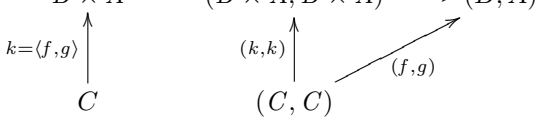
where $\langle f, g \rangle\ x = (f\ x, g\ x)$ pairs up the results of two functions f and g applied to the same input.

Note how the product in the left adjoint of the previous adjunction now participates in the right adjoint of this one, but in a more general way: it takes a pair of sets and builds their Cartesian product.

What is the new left adjoint? It is the functor that duplicates sets, $\mathsf{L}\ X = (X, X)$. This means that its target category is $\mathbf{S}^2$, the category of pairs of both sets and functions. Composition in $\mathbf{S}^2$ is the expected $(f, g) \cdot (h, k) = (f \cdot h, g \cdot k)$. Using this composition rule when instantiating (17) for this adjunction, we get the universal property of pairing:

$$k = \langle f, g \rangle \Leftrightarrow \begin{cases} \pi_1 \cdot k = f \\ \pi_2 \cdot k = g \end{cases}$$





Above we have seen how components of adjoint functors can shift roles, leading to new adjunctions. Is there any adjunction in which the duplication functor Δ, which above plays the left adjoint role, becomes right adjoint? Yes, see our third example below.

Co-pairing: $(+) \dashv \Delta$ The previous adjunction $\Delta \dashv (\times)$ gave us an explanation of what it means to run two functions at the same time, in *parallel*, for the same input. Shifting Δ to the right of the $\dashv$ symbol will provide an explanation for the dual idea of running two functions not in parallel, but in *alternative*:

$$\begin{cases} \mathsf{L}\ (X, Y) = X + Y \\ \mathsf{R}\ X = \Delta\ X = (X, X) \\ \epsilon = \nabla = [id, id] \end{cases} \quad \begin{cases} \lceil k \rceil = (k \cdot i_1, k \cdot i_2) \\ \lfloor (f, g) \rfloor = [f, g] \end{cases} \tag{23}$$

The corresponding left adjoint builds the disjoint union $X + Y$ of a pair of sets (X, Y) inhabited with X and Y data via two range-disjoint injections $i_1 : X \to X + Y$ and $i_2 : Y \to X + Y$. So the equation $i_1\ x = i_2\ y$ has no solution in $X + Y$ and thus any function of type $X + Y \to Z$ is made of two independent components, one of type $X \to Z$ and the other of type $Y \to Z$, which run in

alternative depending on which side of the sum the input is. Such an alternative is denoted by $[f\,,g]$ and, in symbols rather than words, we have $[f\,,g]\cdot i_1 = f$ and $[f\,,g]\cdot i_2 = g$.

By instantiating (17) with (23) one gets the universal property of alternatives:

$$\begin{cases} f = k\cdot i_1 \\ g = k\cdot i_2 \end{cases} \Leftrightarrow k = [f\,,g] \qquad \begin{array}{ccc} & \overset{\Delta}{\underset{(+)}{\rightleftarrows}} & \\ \mathbf{S}^2 & & \mathbf{S} \end{array} \qquad \begin{array}{ccc} (A,A) & & A+A \xrightarrow{\nabla} A \\ {\scriptstyle (f,g)=(k\cdot i_1,k\cdot i_2)}\uparrow & & {\scriptstyle f+g}\uparrow \quad \nearrow_{k} \\ (C,D) & & C+D \end{array} \tag{24}$$

The adjunctions given so far involve the category of sets and (total) functions that provide a basis for so-called *strong* [22] functional programming. For instance, alternatives give rise to conditional computations [4] and so on.

More examples of adjunctions could be given in this setting, see e.g. [11]. We prefer to give a final example that does not fit (directly) in functional programming practice, but is essential to reasoning about functional programs. It links **S** to another category which extends it: its objects are the same (sets) but the morphisms become binary relations instead of functions. This category of relations will be denoted by **R** and its composition corresponds to relational chaining, as seen below.

Power Transpose: $\mathsf{J} \dashv \mathsf{P}$ This adjunction captures the view that every binary relation $R : A \to B$ (a morphism in **R**) can be expressed by a set-valued function $\Lambda R : A \to \mathsf{P}\ B$ (a morphism in **S**), defined by:[5]

$$(\Lambda R)\ a = \{\, b \mid b\ R\ a \,\} \tag{25}$$

Thus $b \in \Lambda R\ a \Leftrightarrow b\ R\ a$, which in relational pointfree notation (the "internal language" of **R**) is written $\in \cdot\, \Lambda R = R$, where $\in : B \leftarrow \mathsf{P}\ B$ is the set *membership* relation. Thus membership "cancels" the power-transpose Λ:

$$\begin{array}{ccc} \mathbf{R} & & \mathbf{S} \\ A \to X & \overset{\Lambda}{\underset{(\in\cdot)}{\underset{\cong}{\rightleftarrows}}} & A \to \mathsf{P}\ X \end{array} \tag{26}$$

We write $A \to X$ instead of $\mathsf{J}\ A \to X$ because the lower adjoint J is the identity on objects. It just converts a function in **S** to the corresponding relation in **R**[6],

[5] Notation $\mathsf{P}\ B$ means the *powerset* of B, i.e. the set of all subsets of B. Also note that we write $b\ R\ a$ to express $(b, a) \in R$, keeping with the tradition of using infix notation in relational facts, e.g. $a \leqslant b$ instead of $(a, b) \in (\leqslant)$ and so on. In this vein, relation composition is expressed by $b\ (S \cdot R)\ a \Leftrightarrow \langle \exists\ c\ ::\ b\ S\ c \wedge c\ R\ a \rangle$.

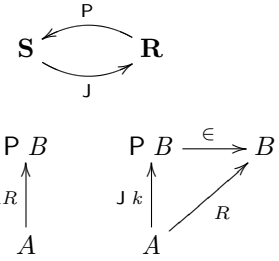
[6] Interestingly, the usual presentation $y = f(x)$ of functions in maths textbooks is relational, not strictly functional.

cf.:

$$\begin{cases} \mathsf{J}\ X = X \\ y\ (\mathsf{J}\ k)\ x \ \Leftrightarrow\ y = k\ x \\ \mathsf{R}\ X = \mathsf{P}\ X = \{\, S \mid S \subseteq X \,\} \end{cases} \quad \begin{cases} \epsilon = (\in) \\ \lceil R \rceil = \Lambda R \\ y\ \lfloor k \rfloor\ x \ \Leftrightarrow\ y\ \in\ (k\ x) \end{cases} \tag{27}$$

Altogether, the adjunction expresses the universal property of power-transposition:

$$k = \Lambda R \ \Leftrightarrow\ \underbrace{\in \cdot k}_{\lfloor k \rfloor} = R \tag{28}$$



This adjunction will prove specially useful later on where dealing with recursion in presence of inductive data types.

9 Properties

The main advantage of a unifying concept such as that of an adjunction is that one can express the rich theory of (17, 19) only once, covering all the particular instances by construction. As already mentioned, several properties that are easy to derive as corollaries of (17, 19) are given in the appendix. The terminology is inspired by [4], among other references that use similar names, see e.g. [18].

To illustrate their application, let us see how the actions of the functors involved in an adjunction can be recovered from the adjunction itself, laws (71) and (80). Taking $\Delta \dashv (\times)$ as example, let us use (71), $\mathsf{R}\ h = \lceil h \cdot \epsilon \rceil$, to find a definition for $f \times g$, which is $\mathsf{R}\ (f, g)$. Since $\epsilon = (\pi_1, \pi_2)$, then $(f, g) \cdot \epsilon = (f \cdot \pi_1, g \cdot \pi_2)$. Since $\lceil (x, y) \rceil = \langle x, y \rangle$, we finally get

$$f \times g = \langle f \cdot \pi_1, g \cdot \pi_2 \rangle \tag{29}$$

Similarly, for $\mathsf{J} \dashv \mathsf{P}$, by (71) we get

$$\mathsf{P}\ R = \Lambda(R \cdot (\in)) \tag{30}$$

that is, $(\mathsf{P}\ R)\ X = \{\, b \mid b\ R\ a \wedge a\ \in\ X \,\}$.[7]

Next, let us calculate $f + g$ as in the left-adjoint of $(+) \dashv \Delta$ (23) using (80), $\mathsf{L}\ g = \lfloor \eta \cdot g \rfloor$. In the same way as above, $f + g = \mathsf{L}\ (f, g) = \lfloor \eta \cdot (f, g) \rfloor$. Since $\eta = \lceil id \rceil$ (74), i.e. $\eta = (i_1, i_2)$ by (23), we get $f + g = \lfloor i_1 \cdot f, i_2 \cdot g \rfloor$ and finally:

$$f + g = [i_1 \cdot f\,, i_2 \cdot g] \tag{31}$$

[7] Note that P is not a relational functor (in $\mathbf{R}$) but rather another way of expressing relations by functions in $\mathbf{S}$. It is often referred to as the *existential image functor* [4]. Interestingly, (30) captures the way the so-called *navigation style* of Alloy [12] works, enabling an (essentially) functional execution of its relational core.

All such properties and those of the appendix involve only one adjunction at a time. Perhaps more interesting are those that arise by composing adjunctions, to be seen shortly. Before this, we address a topic that is very relevant to programming and bears a strong link to adjunctions.

10 Monads

The categorial view of functional programming had a big "push forward" when the concept of a *monad* was incorporated in languages such as e.g. Haskell, making it possible to have purely functional implementations of computations that were regarded as non-functional before. Since the pioneering work by Moogi [15], the concept has gained wider and wider acceptance from both the theory and practice cohorts of the programming community, see e.g. [9,24] among many other references.

The question is: where do monads come from? It turns out that monads arise from adjunctions. Put simply, for every adjunction $\mathsf{L} \dashv \mathsf{R}$, the composition $\mathsf{M} = \mathsf{R} \cdot \mathsf{L}$ is a monad, meaning that M comes equipped with natural transformations η and μ,

$$A \xrightarrow{\eta} \mathsf{M}A \xleftarrow{\mu} \mathsf{M}^2 A$$

such that

$$\mu \cdot \eta = id = \mu \cdot \mathsf{M}\ \eta \tag{32}$$

$$\mu \cdot \mu = \mu \cdot \mathsf{M}\ \mu \tag{33}$$

hold. Definitions

$$\eta = \lceil id \rceil \tag{34}$$

$$\mu = \mathsf{R}\ \epsilon \tag{35}$$

show how the so-called *multiplication* (μ) and *unit* (η) of monad $\mathsf{M} = \mathsf{R} \cdot \mathsf{L}$ arise from $\mathsf{L} \dashv \mathsf{R}$. Proofs that (32, 33) follow from definitions (34, 35) and adjunction properties can be found in the appendix.

As an example, recall the adjunction $\mathsf{J} \dashv \mathsf{P}$ (27). Because J is the identity on objects, it turns out that P, the powerset functor, is a monad. By (34) and (25), its unit is $\eta\ a = \{ a \}$. By (35) and (71), its multiplication $\mu = \Lambda(\in) \cdot (\in)$ is distributed union,

$$\mu\ S = \{ a \mid \langle \exists\ x\ :\ a\ \in\ x :\ x\ \in\ S \rangle \}$$

where S is a set of sets. (The usual notation for $\mu\ S$ is $\bigcup S$.)

In the interest of programming, one may wonder whether, in this powerset adjunction $\mathsf{J} \dashv \mathsf{P}$ (28), one can interpret relational expressions in $\mathbf{R}$ by set-valued functions in $\mathbf{S}$. In particular, one may be interested in implementing relational composition $R \cdot S$ by somehow running their set-valued function counterparts ΛR and ΛS as functional programs.

This is possible as instance of the so-called *monadic* (or *Kleisli*) composition,[8] defined for any adjunction $\mathsf{L} \dashv \mathsf{R}$ as follows,

$$f \bullet g = \mu \cdot \mathsf{M}\, f \cdot g \tag{36}$$

where $\mathsf{M} = \mathsf{R} \cdot \mathsf{L}$ as seen above. One has

$$\lceil f \cdot g \rceil = \lceil f \rceil \bullet \lceil g \rceil \tag{37}$$

as proved in the appendix.

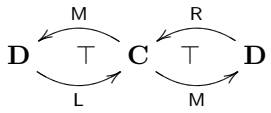
As a well-known example, Kleisli composition enables one to sequence state-based computations in a purely functional, elegant way using the so-called *state monad* which arises from the $(\_ \times K) \dashv (\_^K)$ adjunction (21).[9]

11 Composing Adjunctions

Above we saw the example of a functor (Δ) being at the same time a left adjoint and a right adjoint of a different adjunction. Let us study the situation in which two such adjunctions are chained: $\mathsf{L} \dashv \mathsf{M} \dashv \mathsf{R}$.

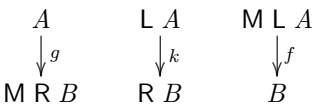
A quick inspection of how a morphism $\mathsf{L}\, A \xrightarrow{k} \mathsf{R}\, B$ can be transformed unveils the composite adjunction $(\mathsf{M}\,\mathsf{L}) \dashv (\mathsf{M}\,\mathsf{R})$:[10]

$$\begin{array}{ll} & \mathsf{M}\,\mathsf{L}\, A \to B \\ \cong & \quad \{\ \mathsf{M} \dashv \mathsf{R}\ \} \\ & \mathsf{L}\, A \to \mathsf{R}\, B \\ \cong & \quad \{\ \mathsf{L} \dashv \mathsf{M}\ \} \\ & A \to \mathsf{M}\,\mathsf{R}\, B \end{array}$$



Given $\mathsf{L}\, A \xrightarrow{k} \mathsf{R}\, B$, $k = \lceil f \rceil_{\mathsf{R}}$ holds for exactly one $\mathsf{M}\,\mathsf{L}\, A \xrightarrow{f} B$. (See the diagram aside.) On the other hand, $k = \lfloor g \rfloor_{\mathsf{L}}$ holds for exactly one $A \xrightarrow{g} \mathsf{M}\,\mathsf{R}\, B$. So the *exchange law*

$$\lceil f \rceil_{\mathsf{R}} = \lfloor g \rfloor_{\mathsf{L}} \tag{38}$$



holds for such $\mathsf{M}\,\mathsf{L}\, A \xrightarrow{f} B$ and $A \xrightarrow{g} \mathsf{M}\,\mathsf{R}\, B$. Observe in the diagram that f and g in (38) live in the same category (**D**).

[8] This is the way relational specifications are handled in [4], for instance.

[9] For more details about this monad and how to calculate with it please see e.g. [18].

[10] In the sequel we adopt the usual shortcut for functor composition, e.g. $\mathsf{M}\,\mathsf{L}$ instead of $\mathsf{M} \cdot \mathsf{L}$ and so on.

The Product-Coproduct "Mix". Let us see an instance of (38) that emerges from composing $(+) \dashv \Delta \dashv (\times)$ and is dear to algebra of programming practitioners [4]: in this case, $\mathsf{M}\,\mathsf{L}\,A \xrightarrow{f} B$ is of type $\Delta\,(+)\,(A,C) \to (B,D)$,

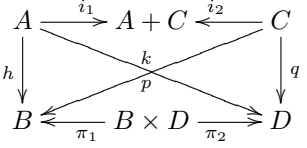
$$f = (A+C, A+C) \xrightarrow{(m,n)} (B,D)$$

and $A \xrightarrow{g} \mathsf{M}\,\mathsf{R}\,B$ is of type $(A,C) \to \Delta\,(\times)\,(B,D)$:

$$g = (A,C) \xrightarrow{(i,j)} (B \times D, B \times D) \tag{39}$$

So, $\lceil f \rceil_{\mathsf{R}} = \lfloor g \rfloor_{\mathsf{L}}$ becomes $\langle m,n \rangle = [i,j]$, which we want to solve next. Looking at (39), we have $i = \langle h,k \rangle$ and $j = \langle p,q \rangle$ for some h,k,p,q. Then:

$$\begin{array}{ll} & \langle m,n \rangle = [\langle h,k \rangle, \langle p,q \rangle] \\ \Leftrightarrow & \quad \{\ (+) \dashv \Delta\ \} \\ & \begin{cases} (m,n) \cdot (i_1,i_1) = (h,k) \\ (m,n) \cdot (i_2,i_2) = (p,q) \end{cases} \\ \Leftrightarrow & \quad \{\ \text{re-arranging}\ \} \\ & \begin{cases} (m,m) \cdot (i_1,i_2) = (h,p) \\ (n,n) \cdot (i_1,i_2) = (k,q) \end{cases} \\ \Leftrightarrow & \quad \{\ \Delta \dashv (\times)\ \} \\ & \begin{cases} m = [h,p] \\ n = [k,q] \end{cases} \end{array}$$

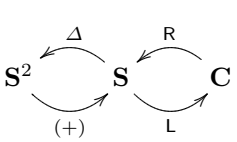


The composite adjunction $(+) \dashv \Delta \dashv (\times)$ therefore yields the well-known *exchange law* [4],

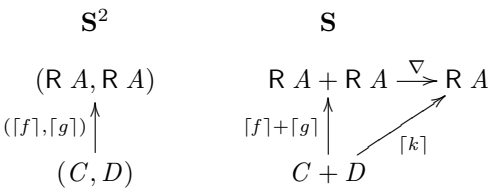
$$\langle [h,p], [k,q] \rangle = [\langle h,k \rangle, \langle p,q \rangle] \tag{40}$$

which is very useful in handling functions that input sums and output products. As will be seen in the sequel, (40) will play an important role when dealing with mutual recursion.

$(+) \dashv \Delta$ *meets* $\mathsf{L} \dashv \mathsf{R}$. As we have seen, adjunction (24) brings with it the possibility of expressing alternative computations. One wonders whether such a possibility can be extended "across" other adjunctions via the composition



that is



meaning:[11]

$$\begin{cases} \lceil f \rceil = \lceil k \rceil \cdot i_1 \\ \lceil g \rceil = \lceil k \rceil \cdot i_2 \end{cases} \Leftrightarrow \lceil k \rceil = [\lceil f \rceil, \lceil g \rceil] \tag{41}$$

Clearly, the right side of (41) can be written $k = \lfloor [\lceil f \rceil, \lceil g \rceil] \rfloor$. Concerning the left side:

$$\begin{cases} \lceil k \rceil \cdot i_1 = \lceil f \rceil \\ \lceil k \rceil \cdot i_2 = \lceil g \rceil \end{cases}$$

$$\Leftrightarrow \quad \{ \text{ fusion (67) and isomorphism (72) (twice) } \}$$

$$\begin{cases} k \cdot \mathsf{L}\ i_1 = f \\ k \cdot \mathsf{L}\ i_2 = g \end{cases}$$

In summary, (41) re-writes to the universal property

$$k = \lfloor [\lceil f \rceil, \lceil g \rceil] \rfloor \Leftrightarrow \begin{cases} k \cdot \mathsf{L}\ i_1 = f \\ k \cdot \mathsf{L}\ i_2 = g \end{cases} \tag{42}$$

that is, we have coproducts in $\mathbf{C}$ induced by the lower adjoint L.

Relational Coproducts. Let us inspect (42) for $\mathsf{L} \dashv \mathsf{R} := \mathsf{J} \dashv \mathsf{P}$ (27). In this case, $\lfloor k \rfloor = (\in) \cdot k$ and $\mathsf{L}\ i_1 = \mathsf{J}\ i_1$ is injection i_1 regarded as a relation, $y\ i_1\ x \Leftrightarrow y = i_1\ x$, which is usually abbreviated to i_1 (similarly for i_2):

$$X = \underbrace{(\in) \cdot [\Lambda R, \Lambda S]}_{[R,S]} \Leftrightarrow \begin{cases} X \cdot i_1 = R \\ X \cdot i_2 = S \end{cases} \tag{43}$$

Thus relational coproducts are born, in which *alternatives* are still denoted by $[R, S]$, as in the functional case, since types always tell us whether we are in $\mathbf{S}$ or $\mathbf{R}$.

As another example, this time concerning $(\_ \times K) \dashv (\_^K)$ (21), we get

$$k = \mathbf{uncurry}\ [\mathbf{curry}\, f, \mathbf{curry}\, g] \Leftrightarrow \begin{cases} k \cdot (i_1 \times id) = f \\ k \cdot (i_2 \times id) = g \end{cases}$$

and so on and so forth for other $\mathsf{L} \dashv \mathsf{R}$.[12]

12 More About R

We have just seen that the category of relations $\mathbf{R}$ has coproducts. In fact, it has a much richer structure which stems from the powerset construction in $\mathbf{S}$ (27).

[11] The $\lceil \_ \rceil$ and $\lfloor \_ \rfloor$ that occur in (41) and (42) have to do with $\mathsf{L} \dashv \mathsf{R}$, since the corresponding transposes of $(+) \dashv \nabla$ are spelt out via (23).

[12] These facts are actually instances of a more general result: coproducts generalize to so-called *colimits* and these are preserved by left adjoints [14].

The fact that powersets are ordered by set inclusion induces a partial order on relations in $\mathbf{R}$ easy to define:[13]

$$R \subseteq S \Leftrightarrow \langle \forall\ a\ ::\ (\Lambda R\ a) \subseteq (\Lambda S\ a) \rangle \tag{44}$$

Put in another way, every homset $\mathbf{R}(A, B)$ is partially ordered and we say that $\mathbf{R}$ is *order-enriched*.

This enrichment is actually "richer": (44) carries with it a complete Boolean algebra and therefore relation union $(R \cup S)$ and intersection $(R \cap S)$ are defined within the same homset $\mathbf{R}\ (A, B)$ by construction, whose least element is usually denoted by $\bot$ and the largest by $\top$.

The other interesting structural property is that homsets $\mathbf{R}(A, B)$ and $\mathbf{R}(B, A)$ are isomorphic, that is, $\mathbf{R}$ is *self dual*. For each $R \in \mathbf{R}(A, B)$, the corresponding relation in $\mathbf{R}\ (B, A)$ is denoted by R° (the *converse* of R) and we have:[14]

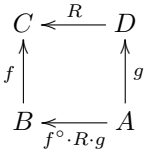
$$b\ R\ a \Leftrightarrow a\ R^\circ\ b$$

This is a major advantage of $\mathbf{R}$ when compared to $\mathbf{S}$, where only isomorphisms can be reversed. Moreover, it turns out that converses of functions play a major role in $\mathbf{R}$. In particular, the useful rule

$$b\ (f^\circ \cdot R \cdot g)\ a \Leftrightarrow (f\ b)\ R\ (g\ a) \tag{45}$$

holds, for suitably typed functions f and g and relation R,[15] please see the type diagram below.

The use of this rule can be appreciated by applying it to both sides of a Galois connection, recall (4): term $f\ a \leqslant x$ becomes $a\ (f^\circ \cdot (\leqslant))\ x$ and term $a \sqsubseteq g\ x$ becomes $a\ ((\sqsubseteq) \cdot g)\ b$. That is, the logical equivalence of a GC (4) becomes the relational *equality*:



$$f^\circ \cdot (\leqslant) = (\sqsubseteq) \cdot g \tag{46}$$

This version of (4) is said to be *pointfree* in the sense that it dispenses with variables, or *points*, a and x.[16] The question arises: how does one describe the preorders $(\leqslant)$ and $(\sqsubseteq)$ at such a *pointfree* level? This is related to a previous question: how does a recursive program such as e.g. take get generated from an equality like (46)? With no further delay we need to bring recursion into our framework of reasoning.

[13] The fact that we use the same symbol to order relations and order powersets should not be a problem, as types disambiguate its use.

[14] Self-duality in $\mathbf{R}$ arises from isomorphism $\mathsf{P}\ X \cong 2^X$ ("sets are predicates") in $\mathbf{S}$. By this and uncurrying, $A \to \mathsf{P}\ B \cong 2^{A \times B}$. Since $A \times B \cong B \times A$, we can go in reverse order and obtain $B \to \mathsf{P}\ A$, etc.

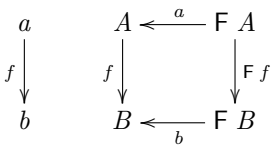
[15] Following a widely adopted convention [4] to save text, we denote "relations that are functions" by lowercase letters.

[16] This is the way (in $\mathbf{R}$) Galois connections are handled in e.g. [1,16].

13 Recursion Comes In

For a given (endo)functor F, any morphism $A \xleftarrow{a} \mathsf{F}\ A$ is said to be an F-algebra, where A is said to be the *carrier* of the algebra. F-algebras form a category provided its morphisms $a \xrightarrow{f} b$ satisfy a particular property,

$$f \cdot a = b \cdot \mathsf{F}\ f \tag{47}$$



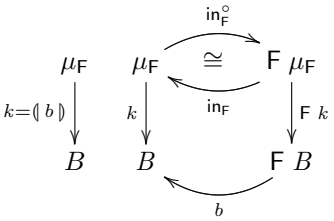
captured in the diagram aside.[17] Property (47) states that A-elements are mapped to B-elements in a structural way. Think for instance of $A = B = \mathbb{N}_0$ being the natural numbers, $\mathsf{F}\ X = X \times X$, $a\ (n, m) = n + m$, $b\ (x, y) = x \times y$ and $f\ x = c^x$, for some fixed c. Then (47) becomes $f\ (a\ (n, m)) = b\ (f\ n, f\ m)$, then $f\ (n + m) = (f\ n) \times (f\ m)$ and finally $c^{n+m} = c^n \times c^m$, which holds in $\mathbb{N}_0$. Thus $(+) \xrightarrow{c^{(-)}} (\times)$ is a F-algebra morphism.

Some situations arise in which a is such that, for every b, f is unique. In such cases, a is an isomorphism[18], that is, there exists some morphism a° such that $a^\circ \cdot a = id$ and $a \cdot a^\circ = id$. Such algebras a are said to be *initial* and usually denoted by in, i.e. $\mathsf{F}\ \mathsf{T} \xrightarrow{\mathsf{in}} \mathsf{T}$ assuming their carrier set denoted by T. The uniqueness of f wrt. b is written $f = (\!|\, b \,|\!)$ and we have the universal property:

$$k = (\!|\, b \,|\!) \ \Leftrightarrow\ k \cdot \mathsf{in} = b \cdot \mathsf{F}\ k$$

Due to the tight relationship between in and F, it is common to write μ_F instead of T and in_F instead of in:

$$k = (\!|\, b \,|\!) \ \Leftrightarrow\ k \cdot \mathsf{in}_\mathsf{F} = b \cdot \mathsf{F}\ k \tag{48}$$



In words, $(\!|\, b \,|\!)$ is referred to as *the*[19] F-*catamorphism* induced by algebra b. As illustrated in the sequel, it is a generic, recursive construct expressing the transformation of μ_F into B in a "recursive-descent" manner dictated by functor F.

[17] Such F-algebra morphisms are often called F-homomorphisms. Note the overloading of f in $a \xrightarrow{f} b$, a F-algebra morphism; and f in (47), a function between the corresponding carriers.

[18] This is known as the *Lambek lemma* [4].

[19] Definite article because it is unique.

A very simple example of catamorphism is the "for-loop" combinator defined over the natural numbers ($\mu_{\mathsf{F}} = \mathbb{N}_0$) in which $\mathsf{F}\ X = 1 + X$:

$$\mathsf{for}\ b\ i = (\!|\ [\underline{i}, b]\ |\!) \tag{49}$$

In this case,

$$\mathsf{in}_{\mathsf{F}} = [\mathsf{zero}, \mathsf{succ}] \tag{50}$$

is the so-called *Peano algebra* which builds natural numbers, where $1 \xrightarrow{\mathsf{zero}} \mathbb{N}_0 = \underline{0}$ generates 0 and $\mathbb{N}_0 \xrightarrow{\mathsf{succ}} \mathbb{N}_0$, the successor function $\mathsf{succ}\ n = n + 1$, generates all other numbers. (By $1 \xrightarrow{\underline{k}} X$ we mean the constant function $\underline{k}\ \_ = k$, where 1 is a singleton object.)

By unfolding (49) through (48) one derives

$$\begin{array}{l} \mathsf{for}\ b\ i\ 0 = i \\ \mathsf{for}\ b\ i\ (n+1) = b\ (\mathsf{for}\ b\ i\ n) \end{array}$$

clearly showing that b is the loop-body and i is the loop-initialization.[20]

Due to its genericity, the catamorphism concept has proved very useful in studying functional recursion. Similarly to [11], but extending this work towards the relational setting, the remainder of this paper addresses the "chemistry" between adjunctions and catamorphisms.

$(\!|\ \_\ |\!)$ meets $\mathsf{L} \dashv \mathsf{R}$. As a first step in the investigation of such "chemistry", we set ourselves the task of solving the equation $\lceil f \rceil = (\!|\ \lceil h \rceil\ |\!)$, where $\lceil \_ \rceil$ is the R-transpose of some adjunction $\mathsf{L} \dashv \mathsf{R}$, $\lceil f \rceil : \mu_{\mathsf{F}} \to \mathsf{R}\ A$ and $\lceil h \rceil : \mathsf{F}\ \mathsf{R}\ A \to \mathsf{R}\ A$:

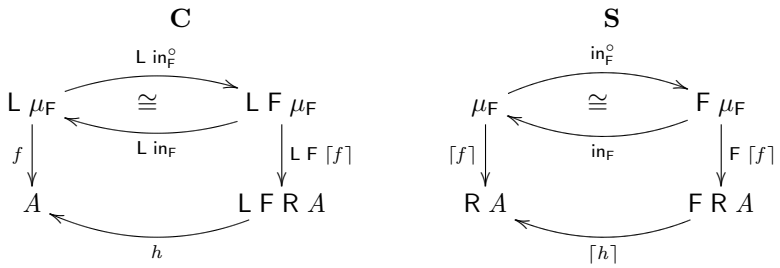
$$\begin{array}{ll} & \lceil f \rceil = (\!|\ \lceil h \rceil\ |\!) \\ \Leftrightarrow & \quad \{\ \text{cata-universal (48)}\ \} \\ & \lceil f \rceil \cdot \mathsf{in}_{\mathsf{F}} = \lceil h \rceil \cdot \mathsf{F}\ \lceil f \rceil \\ \Leftrightarrow & \quad \{\ \text{fusion (67) twice}\ \} \\ & \lceil f \cdot \mathsf{L}\ \mathsf{in}_{\mathsf{F}} \rceil = \lceil h \cdot \mathsf{L}\ \mathsf{F}\ \lceil f \rceil \rceil \\ \Leftrightarrow & \quad \{\ \text{isomorphism } \lceil \_ \rceil \text{ (72)}\ \} \\ & f \cdot \mathsf{L}\ \mathsf{in}_{\mathsf{F}} = h \cdot \mathsf{L}\ \mathsf{F}\ \lceil f \rceil \end{array}$$

Altogether,

$$f \cdot \mathsf{L}\ \mathsf{in}_{\mathsf{F}} = h \cdot \mathsf{L}\ \mathsf{F}\ \lceil f \rceil \quad \Leftrightarrow \quad \lceil f \rceil = (\!|\ \lceil h \rceil\ |\!) \tag{51}$$

[20] In spite of its elementary nature, the for-loop combinator is very useful in programming, see e.g. [8,19]. The unfolding of (49) down to the given pointwise definition is routine work in algebra of programming practice, see e.g. [18]. Starting from (48), it mainly uses the laws of the $(+) \dashv \nabla$ adjunction.

cf. the diagrams:



Although we did not get rid of $\lceil \_ \rceil$ from the left side of (51), this result already offers us something useful, as the following example shows.

Let us see how $(\!|\ \_\ |\!)$ meets $\Delta \dashv (\times)$, the pairing adjunction (22) where (recall) $\mathsf{L}\ f = \Delta\ f = (f, f)$, $\epsilon = (\pi_1, \pi_2)$ and $\lceil (f, g) \rceil = \langle f, g \rangle$. In this case, the left-hand side of (51) becomes:

$$
\begin{array}{ll}
& (f, g) \cdot \mathsf{L}\ \mathsf{in_F} = (h, k) \cdot \mathsf{L}\ (\mathsf{F}\ \lceil (f, g) \rceil) \\
\Leftrightarrow & \quad \{\ \mathsf{L}\ f = (f, f)\ ;\ \lceil (f, g) \rceil = \langle f, g \rangle\ \} \\
& (f, g) \cdot (\mathsf{in_F}, \mathsf{in_F}) = (h, k) \cdot (\mathsf{F}\ \langle f, g \rangle, \mathsf{F}\ \langle f, g \rangle) \\
\Leftrightarrow & \quad \{\ \text{composition and equality of pairs of functions}\ \} \\
& \begin{cases} f \cdot \mathsf{in_F} = h \cdot \mathsf{F}\ \langle f, g \rangle \\ g \cdot \mathsf{in_F} = k \cdot \mathsf{F}\ \langle f, g \rangle \end{cases}
\end{array}
$$

Concerning the right-hand side:

$$
\begin{array}{ll}
& \lceil (f, g) \rceil = (\!|\ \lceil (h, k) \rceil\ |\!) \\
\Leftrightarrow & \quad \{\ \lceil (f, g) \rceil = \langle f, g \rangle \text{ twice}\ \} \\
& \langle f, g \rangle = (\!|\ \langle h, k \rangle\ |\!)
\end{array}
$$

Putting both sides together we get the so-called *mutual recursion* law:

$$
\langle f, g \rangle = (\!|\ \langle h, k \rangle\ |\!) \Leftrightarrow \begin{cases} f \cdot \mathsf{in_F} = h \cdot \mathsf{F}\ \langle f, g \rangle \\ g \cdot \mathsf{in_F} = k \cdot \mathsf{F}\ \langle f, g \rangle \end{cases} \tag{52}
$$

This first outcome of the interplay between recursion and adjunctions is already useful in programming, as it can help reduce the complexity of some dynamic programming (DP) problems by calculation. In particular, it can be used to convert complex multiple recursion into Peano-recursion, i.e., for-loops (49).

Many examples of application of (52) could be given.[21] Perhaps the most famous (and shortest to explain) is the Fibonacci series, a classic in *DP*:

[21] See e.g. [18], where examples include the derivation of efficient implementations of $\mathbb{R}$-valued functions from their Taylor series expansion into mutually recursive functions that are "packed together" via (52).

$$
\begin{aligned}
&\mathit{fib}\ 0 = 1\\
&\mathit{fib}\ 1 = 1\\
&\mathit{fib}\ (n+2) = \mathit{fib}\ (n+1) + \mathit{fib}\ n
\end{aligned}
$$

By defining $f\ n = \mathit{fib}\ (n+1)$ and expanding it through the Peano-algebra, one gets,

$$
\begin{cases} f\ 0 = 1 \\ f\ (n+1) = f\ n + \mathit{fib}\ n \end{cases}
$$

$$
\begin{cases} \mathit{fib}\ 0 = 1 \\ \mathit{fib}\ (n+1) = f\ n \end{cases}
$$

that is:

$$
\begin{cases} f \cdot [\mathsf{zero}, \mathsf{succ}] = [\underline{1}, \mathsf{add}] \cdot \langle f, \mathit{fib} \rangle \\ \mathit{fib} \cdot [\mathsf{zero}, \mathsf{succ}] = [\underline{1}, \pi_1] \cdot \langle f, \mathit{fib} \rangle \end{cases}
$$

By (52) and the *exchange law* (40), this leads to the for-loop

$$\langle f, \mathit{fib} \rangle = (\!|\, [\underline{(1,1)}, \langle \mathsf{add}, \pi_1 \rangle] \,|\!)$$

that is (in Haskell syntax):

$$
\begin{aligned}
&\mathit{fib} = \pi_2 \cdot \mathsf{for}\ \mathit{loop}\ (1,1)\ \mathbf{where}\\
&\quad \mathit{loop}\ (x,y) = (x+y, x)
\end{aligned}
$$

In retrospect, note how the main ingredients of the calculation above rely mainly on adjunctions: law (52), which instantiates (51) for $\Delta \dashv (\times)$, and law (40), which arises from $(+) \dashv \Delta \dashv (\times)$.

14 Towards Adjoint-Recursion

The relevance of (51) is, as already seen in (52), the possibility of "converting" a non-standard recursive construct (f) into a catamorphism by right-adjoint transposition. However, (51) still needs the transpose $\lceil f \rceil$ on the left side of the equivalence. Can we do without this transpose?

For this to happen, we need to get rid of $\lceil f \rceil$ in the recursive call $\mathsf{L}\ \mathsf{F}\ \lceil f \rceil$. The resource we have for this is the *cancellation* law (66), $\epsilon \cdot \mathsf{L}\ \lceil f \rceil = f$. However, L in $\mathsf{L}\ \mathsf{F}\ \lceil f \rceil$ is in the wrong position and needs to commute with F. So we need a *distributive* law $\mathsf{L}\ \mathsf{F} \to \mathsf{F}\ \mathsf{L}$ or, more generally, a *natural transformation*

$$\phi : \mathsf{L}\ \mathsf{F} \to \mathsf{G}\ \mathsf{L} \tag{53}$$

enabling such a commutation over some other functor G. Still, for $\epsilon \cdot \mathsf{L}\ \lceil f \rceil = f$ to be of use, we need $\mathsf{G}\ \epsilon$ somewhere in the pipeline. We thus refine $h := h \cdot \mathsf{G}\ \epsilon \cdot \phi$ in (51) and carry on:

$$
\begin{array}{ll}
& \lceil f \rceil = (\!|\, \lceil h \cdot \mathsf{G}\ \epsilon \cdot \phi \rceil \,|\!) \\
\Leftrightarrow & \quad \{\ (51)\ \} \\
& f \cdot \mathsf{L}\ \mathsf{in_F} = h \cdot \mathsf{G}\ \epsilon \cdot \phi \cdot \mathsf{L}\ \mathsf{F}\ \lceil f \rceil \\
\Leftrightarrow & \quad \{\ \text{natural-}\phi\text{: } \phi \cdot \mathsf{L}\ \mathsf{F}\ f = \mathsf{G}\ \mathsf{L}\ f \cdot \phi\ \} \\
& f \cdot \mathsf{L}\ \mathsf{in_F} = h \cdot \mathsf{G}\ \epsilon \cdot \mathsf{G}\ \mathsf{L}\ \lceil f \rceil \cdot \phi \\
\Leftrightarrow & \quad \{\ \text{functor } \mathsf{G};\ \text{cancellation } \epsilon \cdot \mathsf{L}\ \lceil f \rceil = f\ (66)\ \} \\
& f \cdot \mathsf{L}\ \mathsf{in_F} = h \cdot \mathsf{G}\ f \cdot \phi
\end{array}
$$

We thus reach

$$f \cdot (\mathsf{L}\ \mathsf{in_F}) = h \cdot \mathsf{G}\ f \cdot \phi \quad \Leftrightarrow \quad \lceil f \rceil = (\!|\, \lceil h \cdot \mathsf{G}\ \epsilon \cdot \phi \rceil \,|\!) \tag{54}$$

where natural transformation ϕ:$\mathsf{L}\ \mathsf{F} \rightarrow \mathsf{G}\ \mathsf{F}$ captures a switch of recursion-pattern between f and the F-catamorphism $\lceil f \rceil$, through L.

What kind of function is f? Because $\mathsf{in_F}$ is an isomorphism, f can also be written as

$$f = h \cdot \mathsf{G}\ f \cdot \phi \cdot \mathsf{L}\ \mathsf{in_F^\circ}$$

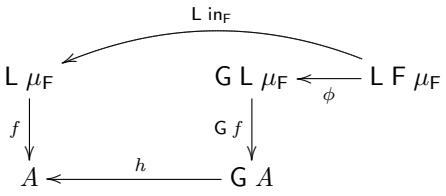
that is, f is a fixpoint. It is an instance of the recursive scheme

$$f = c \cdot \mathsf{G}\ f \cdot d \tag{55}$$

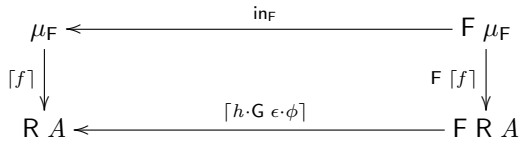
which is often termed *hylomorphism* [4] and generalizes the catamorphism combinator. (For $d = \mathsf{in}^\circ_{\mu_\mathsf{F}}$ one would have, by (48), $f = (\!|\, c \,|\!)$.) Altogether, (54) shows the equivalence of a G-hylomorphism and an F-catamorphism made possible by natural transformation $\phi : \mathsf{L}\ \mathsf{F} \rightarrow \mathsf{G}\ \mathsf{F}$:

$$\underbrace{f \cdot (\mathsf{L}\ \mathsf{in_F}) = h \cdot \mathsf{G}\ f \cdot \phi}_{\mathsf{G}\text{-hylomorphism}} \quad \Leftrightarrow \quad \underbrace{\lceil f \rceil = (\!|\, \lceil h \cdot \mathsf{G}\ \epsilon \cdot \phi \rceil \,|\!)}_{\text{adjoint } \mathsf{F}\text{-catamorphism}}$$

Note that, in general, they sit in different categories. The G-hylomorphism (in say $\mathbf{C}$) is depicted in a diagram by:



The diagram of its adjoint F-catamorphism (in $\mathbf{S}$) is:



$$A \xleftarrow{h} \mathsf{G}\ A \xleftarrow{\mathsf{G}\ \epsilon} \mathsf{G}\ \mathsf{L}\ \mathsf{R}\ A \xleftarrow{\phi} \mathsf{L}\ \mathsf{F}\ \mathsf{R}\ A$$

We shall refer to (54) as the *adjoint-cata* theorem. Its main interest is that one can use the "*cata*-artillery" that stems from universal property (48) to reason about hylomorphism f by converting f to $\lceil f \rceil$.[22] But not necessarily: by (17) on the right side of (54), we get

$$\underbrace{f \cdot (\mathsf{L}\ \mathsf{in}_{\mathsf{F}}) = h \cdot \mathsf{G}\ f \cdot \phi}_{\mathsf{G}\text{-hylomorphism}} \quad \Leftrightarrow \quad f = \underbrace{\lfloor (\!| \lceil h \cdot \mathsf{G}\ \epsilon \cdot \phi \rceil |\!) \rfloor}_{(\!|h|\!)} \tag{56}$$

giving birth to a new recursion combinator with *universal property*:

$$f = (\!|h|\!) \quad \Leftrightarrow \quad f \cdot \mathsf{L}\ \mathsf{in}_{\mathsf{F}} = h \cdot \mathsf{G}\ f \cdot \phi$$

In case ϕ is invertible, i.e. an isomorphism, the above converts to

$$f = (\!|h|\!) \quad \Leftrightarrow \quad f \cdot \underbrace{\mathsf{L}\ \mathsf{in}_{\mathsf{F}} \cdot \phi^{\circ}}_{\alpha} = h \cdot \mathsf{G}\ f \tag{57}$$

which shares the structure of (48), where we started from. Indeed, for the trivial adjunction in which L and R are the identity functors, $\phi = id$, $\mathsf{F} = \mathsf{G}$ and $(\!|h|\!)$ coincides with $(\!|\, h \,|\!)$. But, in general, (57) has a much wider scope as it enables us to handle recursive structures (μ_{F}) "embraced" by some contex information ($\mathsf{L}\ \mu_{\mathsf{F}}$), a quite common situation in programming.

For instance, f may be applied to a recursive structure x paired with some data k, $f\ (x, k)$. While this falls off the scope of (48), it is handled by (56) for $\mathsf{L}\ X = X \times K$, the lower adjoint of the exponentials adjunction (21). This is precisely the situation in a result known as the *Structural Recursion Theorem* which is proved in [4] with no explicit connection to the underlying adjunction.[23]

Clearly, (56) is much wider in scope. And, as it turns out, it also covers another result in [4] as special case, this time involving the already mentioned category of relations $\mathbf{R}$. This is addressed in the following section.

[22] As explained extensively in [26], the expressive power of hylomorphisms (55) comes at the cost of being more difficult to reason with when compared to catamorphisms, as they lack, in general, a universal property.

[23] See Theorem 3.1 in [4], which we can now regard as a corollary of (56).

15 Going Relational

Let us inspect what (56) means in presence of the power-transpose adjunction $\mathsf{J} \dashv \mathsf{P}$ (27). Thanks to J being the identity on objects, we may choose G (in $\mathbf{R}$) as defined by:

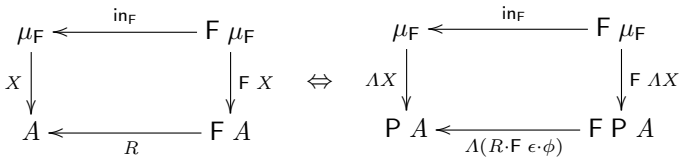
$$y \; \mathsf{G} \; (\mathsf{L} \; f) \; x \;\; \Leftrightarrow \;\; y = \mathsf{F} \; f \; x \tag{58}$$

In words, G establishes a structural relationship between object structures x and y, via the relation $\mathsf{L} \; f$, iff y is the outcome of mapping f over x in $\mathbf{S}$. That is, G is the *relator* [3] that models $\mathsf{F} \; f$ in $\mathbf{R}$. Moreover, (58) is nothing but (53) written pointwise, for $\phi = id$.

Given the close association of G to F expressed by (58), there is no harm in writing only one such symbol (e.g. F) knowing that F in a relational context means G. Assuming this notation convention, and knowing that ϕ "is" the identity, (54) instantiates to

$$X \cdot \mathsf{in_F} = R \cdot \mathsf{F} \; X \;\; \Leftrightarrow \;\; \Lambda X = (\!|\, \Lambda(R \cdot \mathsf{F} \in) \,|\!) \tag{59}$$

depicted by diagrams as follows:[24]



Finally, there is little harm in denoting the new combinator of (56) by $(\!|\, R \,|\!)$ instead of $\langle\!|R|\!\rangle$, giving birth to the *relational catamorphism* combinator:

$$X \cdot \mathsf{in_F} = R \cdot \mathsf{F} \; X \;\; \Leftrightarrow \;\; X = \underbrace{\in \cdot (\!|\, \Lambda(R \cdot \mathsf{F} \in) \,|\!)}_{(\!|\, R \,|\!)} \tag{60}$$

Thus "banana-brackets" are extended to *relations*, giving birth to *inductive relations*. Note that R is a *relational* F-algebra, which is checked in every recursive descent of $(\!|\, R \,|\!)$ across the input data.

Before proceeding to examples, it should be mentioned that the equivalence

$$X = (\!|\, R \,|\!) \Leftrightarrow \Lambda X = (\!|\, \Lambda(R \cdot \mathsf{F} \in) \,|\!) \tag{61}$$

—which is another way of expressing (60)—is known in the literature as the *Eilenberg-Wright* Lemma [4]. So we have just shown that this lemma follows from the more general "adjoint-cata" theorem (54) via the *power-transpose* adjunction $\mathsf{J} \dashv \mathsf{P}$.

$(\!|\_|\!)$-reflection and More. As a first introduction to reasoning about inductive relations in $\mathbf{R}$, let us see what we get from (60) when $X = id$. Put in another

[24] Note that the left diagram lives in $\mathbf{R}$ while the right one lives in $\mathbf{S}$.

way, we wish to solve $id = (\!|\, R \,|\!)$ for R. Since $\mathsf{F}\ id = id$, (60) immediately gives us $\mathsf{in_F} = R \ \Leftrightarrow\ id = (\!|\, R \,|\!)$, meaning that the equation $id = (\!|\, R \,|\!)$ has one sole solution, $R = \mathsf{in_F}$. Substituting, we get

$$(\!|\, \mathsf{in_F} \,|\!) = id \tag{62}$$

known as the *reflection* law [4]. In words, it means that recursively dismantelling a tree-structure into its parts and assembling these back again yields the original tree-structure.

Taking the case of the Peano algebra $\mathsf{in_F} = [\mathsf{zero}, \mathsf{succ}]$ (50), where $\mathsf{F}\ X = 1 + X$, as example, we get $(\!|\, [\mathsf{zero}, \mathsf{succ}] \,|\!) = \mathsf{for}\ \mathsf{succ}\ 0 = id$. Note that $[\mathsf{zero}, \mathsf{succ}]$ is a function, and so we actually do not need (60) for this, (48) where we started from is enough.

Now, since we can plug relations into (60), how about going for something larger than $[\mathsf{zero}, \mathsf{succ}]$, for instance $(\!|\, [\mathsf{zero}, \mathsf{zero} \cup \mathsf{succ}] \,|\!)$? (Recall from (43) that relation union and alternatives involving relations are well-defined.)

Let us first of all see what kind of relation $X = (\!|\, [\mathsf{zero}, \mathsf{zero} \cup \mathsf{succ}] \,|\!)$ is, governed by universal property (60):

$$\begin{array}{ll}
& X = (\!|\, [\mathsf{zero}, \mathsf{zero} \cup \mathsf{succ}] \,|\!) \\
\Leftrightarrow & \quad \{\ (60) \text{ for } \mathsf{F}\ X = id + X\ \} \\
& X \cdot [\mathsf{zero}, \mathsf{succ}] = [\mathsf{zero}, \mathsf{zero} \cup \mathsf{succ}] \cdot (id + X) \\
\Leftrightarrow & \quad \{\ (+) \dashv \Delta \text{ in } \mathbf{R}\text{: fusion (76) and absorption (77)}\ \} \\
& [X \cdot \mathsf{zero}, X \cdot \mathsf{succ}] = [\mathsf{zero}, (\mathsf{zero} \cup \mathsf{succ}) \cdot X] \\
\Leftrightarrow & \quad \{\ (+) \dashv \Delta \text{ in } \mathbf{R}\text{: isomorphism}\ \} \\
& \begin{cases} X \cdot \mathsf{zero} = \mathsf{zero} \\ X \cdot \mathsf{succ} = (\mathsf{zero} \cup \mathsf{succ}) \cdot X \end{cases} \\
\Leftrightarrow & \quad \{\ \text{go pointwise in } \mathbf{R} \text{ via relation composition and (45), several times}\ \} \\
& \begin{cases} n\ X\ 0 \ \Leftrightarrow\ n = 0 \\ m\ X\ (n+1) = \langle \exists\ k\ :\ k\ X\ n :\ m = 0 \vee m = k + 1 \rangle \end{cases} \\
\Leftrightarrow & \quad \{\ \text{simplify}\ \} \\
& \begin{cases} n\ X\ 0 \ \Leftrightarrow\ n = 0 \\ m\ X\ (n+1) \ \Leftrightarrow\ m = 0 \vee (m-1)\ X\ n \end{cases}
\end{array}$$

By inspection, it can be seen that X is the *less-or-equal* relation in $\mathbb{N}_0$. Indeed, by replacing X by $\leqslant$ we get:

- base clause—$n \leqslant 0 \ \Leftrightarrow\ n = 0$, which means that 0 is the infimum of the ordering.
- inductive clause—$m \leqslant n + 1$, which means that either $m = 0$ (the infimum of the ordering again) or, for $m \neq 0$, we have $m \leqslant n + 1 \ \Leftrightarrow\ m - 1 \leqslant n$, something we have seen many times in school algebra.

All in all, our calculations show that the $(\leqslant)$ ordering on the natural numbers is an *inductive relation*. Note, however, this is not a privilege of $(\leqslant) : \mathbb{N}_0 \to \mathbb{N}_0$, as we shall see next.

16 Back to Galois Connections

Recall from Sect. 5 that the prefix $(\sqsubseteq)$ ordering over finite lists was handled assuming basic "axiom" (10),

$$s \sqsubseteq (h : t) \;\Leftrightarrow\; s = [\,] \;\vee\; \langle \exists\; s' \;::\; s = (h : s') \wedge s' \sqsubseteq t \rangle$$

as well as the assumption that the empty sequence $[\,]$ is the infimum of the ordering. (Further recall that these assumptions were needed where e.g. calculating take from its Galois connection specification.)

Let us work out $(\sqsubseteq)$ in the same way as $(\leqslant)$ above. There are two constructors of finite lists, either $\mathsf{nil}\ \_ = [\,]$ generating the empty list; or $\mathsf{cons}\ (a, x) = a : x$ generating a new list $a : x$ from an existing one (x) by placing a new element (a) at the front of x.

Thus the initial algebra in this case is $\mathsf{in_F} = [\mathsf{nil}, \mathsf{cons}]$, giving way to catamorphisms over a slightly more complex relator, $\mathsf{F}\ R = id + id \times R$, where $id \times R$ has to do with the fact that cons requires two arguments.

The parallel between $[\mathsf{zero}, \mathsf{succ}]$ and $[\mathsf{nil}, \mathsf{cons}]$ is obvious, and so we move straight to defining $(\sqsubseteq)$ as the *inductive relation* (a.k.a. relational catamorphism):

$$(\sqsubseteq) = (\!|\, [\mathsf{nil}, \mathsf{nil} \cup \mathsf{cons}] \,|\!) \tag{63}$$

The reader can easily replay the calculation of $(\leqslant)$ this time for $(\sqsubseteq)$ and conclude that "axioms" (10) and $x \sqsubseteq [\,] \;\Leftrightarrow\; x = [\,]$ are indeed the *pointwise equivalent* to defining the list-catamorphism (63) in $\mathbf{R}$.

Eventually, we are in position to answer the main question in Sect. 5, raised by the calculation of recursive right-adjoints such as integer division and take:

"Where does this induction come from?"

It is now clear that what turns such adjoints into recursive (inductive) functions is the very nature of the partial orderings that express them as "best solutions", which are bound to be inductive relations as dictated by the inductive structure of the underlying data.

Last but not least, there is yet another advantage: in $\mathbf{R}$ we can resort to the pointfree version of Galois connections (46), where all the components of the connections are homogeneous—all of them are *morphisms* of a (single) category, $\mathbf{R}$—be them functions, orderings or other relations. By catamorphism algebra, the reasonings of Sect. 5 can be performed at *pointfree* level, in a more calculational style, possibly assisted by GC-oriented proof assistants [21], as detailed below.

17 Related and Current Work

In his landmark paper [11] on "adjoint folds and unfolds", Ralf Hinze leaves the following suggestion:

> *(...) Finally, we have left the exploration of* ***relational*** *adjoint (un)folds to future work.*

Following this hint was the main motivation for the research reported in this paper. The main outcome is a unified view of the relational algebra of programming, in particular concerning results in the literature [4] that now fit together as outcomes of the generic *adjoint-cata* theorem of Sect. 14.

The paper is also aimed at framing, in a wider setting, the author's long standing interest in Galois connections as a generic reasoning device [16,17,21]. Previous work also includes the use of adjunctions in a categorial approach to linear algebra [13] and in calculating tail-recursive programs by "left Peano recursion" [8,19].

Current work is going in two main directions. On the applications' side, trying to evaluate how generic and useful the idea of deriving programs from Galois connections is (recall Sect. 5) and whether this can be (semi)automated by tools such as the Galculator [21]. This would have the advantage over e.g. [16] of not requiring in-depth knowledge of the algebra of relational operators such as e.g. *shrinking*.

On the side of foundations of program semantics, we would like to explore the hint in [20] of working out the relationship between *denotational semantics* and *structured operational semantics* (SOS, regarded as an inductive relation) [25] as an instance of the adjoint-cata theorem. This is expected to enable a calculational flavour in programming language semantics theory.

18 Summary

Science proceeds from the particular to the general. Scientific maturity is achieved when convincing explanations are given around simple (but expressive) concepts, generic enough to encompass an entire theoretical framework. Simplicity and elegance in science enhance scientific communication, make concepts more understandable and knowledge more lasting.

Adjunctions are one such concept, expressive and general enough to capture much of mathematics and theory of programming.

Throughout this work, the author learned to appreciate "adjoint folds" even more and to regard *adjunctions* as a very fertile device for explaining programming as a whole. So important that *teaching* them (inc. *Galois connections*) should be mainstream. May the tutorial flavour of the current paper contribute, however little, to this end.

Acknowledgments. The author wishes to thank the organizers of WADT'22 for inviting him to give the talk which led to this paper. His work is financed by National Funds

through the FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the IBEX project, with reference PTDC/CCI-COM/4280/2021.

A Properties of Adjunctions and Monads

Corollaries of $k = \lceil f \rceil \Leftrightarrow \epsilon \cdot \mathsf{L}\ k = f$ *(17)*

reflection:

$$\lceil \epsilon \rceil = id \tag{64}$$

that is,

$$\epsilon = \lfloor id \rfloor \tag{65}$$

cancellation:

$$\epsilon \cdot \mathsf{L}\ \lceil f \rceil = f \tag{66}$$

fusion:

$$\lceil h \rceil \cdot g = \lceil h \cdot \mathsf{L}\ g \rceil \tag{67}$$

absorption:

$$(\mathsf{R}\ g) \cdot \lceil h \rceil = \lceil g \cdot h \rceil \tag{68}$$

naturality:

$$h \cdot \epsilon = \epsilon \cdot \mathsf{L}\ (\mathsf{R}\ h) \tag{69}$$

closed definition:

$$\lfloor k \rfloor = \epsilon \cdot (\mathsf{L}\ k) \tag{70}$$

functor:

$$\mathsf{R}\ h = \lceil h \cdot \epsilon \rceil \tag{71}$$

isomorphism:

$$\lceil f \rceil = \lceil g \rceil \Leftrightarrow f = g \tag{72}$$

Dual corollaries of $k = \lfloor f \rfloor \Leftrightarrow \mathsf{R}\ k \cdot \eta = f$ *(19)*

reflection:

$$\lfloor \eta \rfloor = id \tag{73}$$

that is,

$$\eta = \lceil id \rceil \tag{74}$$

cancellation:

$$\mathsf{R}\ \lfloor f \rfloor \cdot \eta = f \tag{75}$$

fusion:

$$g \cdot \lfloor h \rfloor = \lfloor \mathsf{R}\ g \cdot h \rfloor \tag{76}$$

absorption:

$$\lfloor h \rfloor \cdot \mathsf{L}\ g = \lfloor h \cdot g \rfloor \tag{77}$$

naturality:

$$h \cdot \epsilon = \epsilon \cdot \mathsf{L}\ (\mathsf{R}\ h) \tag{78}$$

closed definition:

$$\lceil g \rceil = (\mathsf{R}\ g) \cdot \eta \tag{79}$$

functor

$$\mathsf{L}\ g = \lfloor \eta \cdot g \rfloor \tag{80}$$

cancellation (corollary):

$$\epsilon \cdot \mathsf{L}\ \eta = id \tag{81}$$

Monads. Proof of (32):

$$\begin{array}{ll} & \mu \cdot \mu = \mu \cdot \mathsf{M}\ \mu \\ \Leftrightarrow & \quad \{\ \mu = \mathsf{R}\ \epsilon\ (35);\ \text{functor}\ \mathsf{R}\ \} \\ & \mathsf{R}\ (\epsilon \cdot \epsilon) = (\mathsf{R}\ \epsilon) \cdot (\mathsf{R}\ (\mathsf{L}\ (\mathsf{R}\ \epsilon))) \\ \Leftrightarrow & \quad \{\ \text{functor}\ \mathsf{R}\ \} \\ & \mathsf{R}\ (\epsilon \cdot \epsilon) = \mathsf{R}\ (\epsilon \cdot \mathsf{L}\ (\mathsf{R}\ \epsilon)) \\ \Leftrightarrow & \quad \{\ \text{natural-}\epsilon\ (69)\ \} \\ & \mathsf{R}\ (\epsilon \cdot \epsilon) = \mathsf{R}\ (\epsilon \cdot \epsilon) \end{array}$$

□

Proof of (33):

$$\begin{array}{ll} & \mu \cdot \eta = id = \mu \cdot \mathsf{M}\ \eta \\ \Leftrightarrow & \quad \{\ \mu = \mathsf{R}\ \epsilon,\ \eta = \lceil id \rceil\ \text{etc}\ \} \\ & \mathsf{R}\ \epsilon \cdot \lceil id \rceil = id = \mathsf{R}\ \epsilon \cdot (\mathsf{R}\ \mathsf{L}\ \eta) \\ \Leftrightarrow & \quad \{\ \text{absorption}\ (68);\ \text{functor}\ \mathsf{R}\ \} \\ & \lceil \epsilon \rceil = id = \mathsf{R}\ (\epsilon \cdot \mathsf{L}\ \eta) \\ \Leftrightarrow & \quad \{\ \text{reflection}\ (64);\ \text{cancellation}\ (81)\ \} \\ & \mathit{true} \end{array}$$

□

Proof of (37):

$$
\begin{array}{ll}
& f \bullet g \\
= & \quad \{\ f \bullet g = \mu \cdot \mathsf{M}\, f \cdot g\ \} \\
& \mu \cdot \mathsf{M}\, f \cdot g \\
= & \quad \{\ \mathsf{M} = \mathsf{R}\,\mathsf{L};\ \mu = \mathsf{R}\,\epsilon\ \} \\
& \mathsf{R}\,\epsilon \cdot (\mathsf{R}\,(\mathsf{L}\,f)) \cdot g \\
= & \quad \{\ \text{functor } \mathsf{R}\ \} \\
& \mathsf{R}\,(\epsilon \cdot \mathsf{L}\,f) \cdot g \\
= & \quad \{\ \text{cancellation: } \epsilon \cdot \mathsf{L}\,f = \lfloor f \rfloor;\ g = \lceil \lfloor g \rfloor \rceil\ \} \\
& \mathsf{R}\,\lfloor f \rfloor \cdot \lceil \lfloor g \rfloor \rceil \\
= & \quad \{\ \text{absorption: } (\mathsf{R}\,g) \cdot \lceil h \rceil = \lceil g \cdot h \rceil\ \} \\
& \lceil \lfloor f \rfloor \cdot \lfloor g \rfloor \rceil \\
\Box &
\end{array}
$$

References

1. Backhouse, K., Backhouse, R.C.: Safety of abstract interpretations for free, via logical relations and Galois connections. SCP **15**(1–2), 153–196 (2004)
2. Backhouse, R.C.: Mathematics of Program Construction, p. 608. University of Nottingham. Unpublished Book Draft (2004)
3. Backhouse, R.C., de Bruin, P., Hoogendijk, P., Malcolm, G., Voermans, T.S., van der Woude, J.: Polynomial relators. In: Nivat, M., Rattray, C.S., Rus, T., Scollo, G. (eds.) Proceedings of the 2nd Conference on Algebraic Methodology and Software Technology, AMAST 1991, pp. 303–326. Springer, Heidelberg (1991). Workshops in Computing (1992)
4. Bird, R., de Moor, O.: Algebra of Programming. Prentice-Hall, Upper Saddle River (1997)
5. Boisseau, G., Gibbons, J.: What you Needa know about Yoneda: profunctor optics and the Yoneda lemma (functional pearl). Proc. ACM Program. Lang. **2**(ICFP), 84:1–84:27 (2018)
6. Burstall, R., Lampson, B.: A Kernel language for abstract data types and modules. In: Kahn, G., MacQueen, D.B., Plotkin, G. (eds.) SDT 1984. LNCS, vol. 173, pp. 1–50. Springer, Heidelberg (1984). https://doi.org/10.1007/3-540-13346-1_1
7. Burstall, R.M., Goguen, J.A.: Algebras, theories and freeness. Technical report CSR-101-82, University of Edinburgh, February 1982
8. Danvy, O.: Folding left and right matters: direct style, accumulators, and continuations. J. Funct. Program. **33**, e2 (2023)
9. Gibbons, J., Hinze, R.: Just do it: simple monadic equational reasoning. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, pp. 2–14. ACM, New York (2011)

10. Goguen, J.A., Burstall, R.M.: Institutions: abstract model theory for specification and programming. J. ACM **39**(1), 95–146 (1992)
11. Hinze, R.: Adjoint folds and unfolds – an extended study. SCP **78**(11), 2108–2159 (2013)
12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2012). Revised edition, ISBN 0-262-01715-2
13. Macedo, H.D., Oliveira, J.N.: Typing linear algebra: a biproduct-oriented approach. SCP **78**(11), 2160–2191 (2013)
14. Mac Lane, S.: Categories for the Working Mathematician. GTM, vol. 5. Springer, New York (1978). https://doi.org/10.1007/978-1-4757-4721-8
15. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991)
16. Mu, S.-C., Oliveira, J.N.: Programming from Galois connections. In: de Swart, H. (ed.) RAMICS 2011. LNCS, vol. 6663, pp. 294–313. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21070-9_22
17. Oliveira, J.N.: Biproducts of Galois connections. Presentation at the IFIP WG 2.1 #79 Meeting, Otterlo, NL, January 2019
18. Oliveira, J.N.: Program Design by Calculation (2021). Unpublished book draft, February 2021. Informatics Dept., U. Minho (pdf)
19. Oliveira, J.N.: A note on the under-appreciated for-loop. Technical report TR-HASLab:01:2020 (2020). (pdf), HASLab/U.Minho and INESC TEC
20. Oliveira, J.N.: On the power of adjoint recursion. Presentation at the IFIP WG 2.1 #06 Meeting (Online), October 2021
21. Silva, P.F., Oliveira, J.N.: 'Galculator': functional prototype of a Galois-connection based proof assistant. In: PPDP 2008, pp. 44–55. ACM (2008)
22. Turner, D.A.: Elementary strong functional programming. In: Hartel, P.H., Plasmeijer, R. (eds.) FPLE 1995. LNCS, vol. 1022, pp. 1–13. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60675-0_35
23. Wadler, P.L.: Theorems for free! In: 4th International Symposium on Functional Programming Languages and Computer Architecture, London, pp. 347–359. ACM, September 1989
24. Wadler, P.L.: Comprehending monads. In: Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, Nice, France (1990)
25. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993)
26. Yang, Z., Wu, N.: Fantastic morphisms and where to find them - a guide to recursion schemes. In: MPC 2022. LNCS, vol. 13544, pp. 222–267. Springer, Cham (2022)