Kan Extensions for Program Optimisation *Or:* Art and Dan Explain an Old Trick

Ralf Hinze

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
ralf.hinze@cs.ox.ac.uk
http://www.cs.ox.ac.uk/ralf.hinze/

Abstract. Many program optimisations involve transforming a program in direct style to an equivalent program in continuation-passing style. This paper investigates the theoretical underpinnings of this transformation in the categorical setting of monads. We argue that so-called absolute Kan Extensions underlie this program optimisation. It is known that every Kan extension gives rise to a monad, the codensity monad, and furthermore that every monad is isomorphic to a codensity monad. The end formula for Kan extensions then induces an implementation of the monad, which can be seen as the categorical counterpart of continuation-passing style. We show that several optimisations are instances of this scheme: Church representations and implementation of backtracking using success and failure continuations, among others. Furthermore, we develop the calculational properties of Kan extensions, powers and ends. In particular, we propose a two-dimensional notation based on string diagrams that aims to support effective reasoning with Kan extensions.

Keywords: Haskell, CPS, adjunction, Kan extension, codensity monad, power, end, Church representation, backtracking, string diagram.

1 Introduction

Say you have implemented some computational effect using a monad, and you note that your monadic program is running rather slow. There is a folklore trick to speed it up: transform the monad M into continuation-passing style.

$$\begin{aligned}&\textbf{type}\ \mathsf{C}\ a = \forall z\ .\ (a \to \mathsf{M}\ z) \to \mathsf{M}\ z\\&\textbf{instance}\ \mathit{Monad}\ \mathsf{C}\ \textbf{where}\\&\quad \mathit{return}\ a = \lambda c \to c\ a\\&\quad m \gg\!= k = \lambda c \to m\ (\lambda a \to k\ a\ c)\end{aligned}$$

The type constructor C is a monad, regardless of M. The origins of this trick seem to be unknown. It is implicit in Hughes' tutorial on designing a pretty-printing library [18], which introduces a related construction called context-passing style. Interestingly, Hughes makes C parametric in the type variable z, rather than

J. Gibbons and P. Nogueira (Eds.): MPC 2012, LNCS 7342, pp. 324–362, 2012.

locally quantifying over z. Presumably, this is because no Haskell system supported rank-2 types at the time of writing the paper. Only in 1996 Augustsson added support for local universal quantification to the Haskell B. Compiler (hbc 0.9999.0) and I started using it.

My goal was to provide a fast implementation of backtracking in Haskell—the first promising results were detailed in a long technical report [12]. Briefly, the idea is to use two continuations, a success and a failure continuation. Failure and choice can then be implemented as follows.

$$
\begin{aligned}
&\mathbf{type}\ \mathsf{B}\ a = \forall z\ .\ (a \to z \to z) \to z \to z\\
&\mathit{fail}\ :\ \mathsf{B}\ a\\
&\mathit{fail} = \lambda s\, f \to f\\
&(\mid) : \mathsf{B}\ a \to \mathsf{B}\ a \to \mathsf{B}\ a\\
&m \mid n = \lambda s\, f \to m\ s\ (n\ s\ f)
\end{aligned}
$$

We shall see later that this implementation of backtracking is an instance of the trick. This particular application can be traced back to a paper by Mellish and Hardy [29], who showed how to integrate Prolog into the POPLOG environment. Their setting is an imperative one; Danvy and Filinski [9] explained how to recast the approach in purely functional terms. Since then the trick has made several appearances in the literature, most notably [13,8,33,20,28].

The purpose of this paper is to justify the trick and explain its far-reaching applications. There is no shortage of proofs in the aforementioned papers, but no work relates the original monad M to the improved monad C. Since the transformation is labelled 'program optimisation', one would hope that M is isomorphic to C, but sadly this is not the case. We shall see that $\mathsf{M}\ a$ is instead isomorphic to $\forall z\ .\ (a \to \mathsf{R}\ z) \to \mathsf{R}\ z$ for some magic functor R related to M.

The proofs will be conducted in a categorical setting. We will argue that continuations are an *implementation* of a categorical concept known as a right Kan extension, Kan extension for short. For the most part, we will prove and program against the *specification* of a Kan extension. This is in contrast to the related work, including my papers, which take the rank-2 types as the point of departure. (One could argue that this violates one of the fundamental principles of computer science, that we should program against an interface, not an implementation.) It should come as little surprise that all of the necessary categorical concepts and results appear either explicitly or implicitly in Mac Lane's masterpiece [27]. In fact, the first part of this paper solves Exercise X.7.3 of the textbook. Specifically, we show that

- a Kan extension gives rise to a monad, the so-called codensity monad, thereby solving Exercise X.7.3(a);
- every monad is isomorphic to a codensity monad, solving Exercise X.7.3(c);
- we show that Kan extensions can be implemented using ends and powers [27, Section X.4], which we argue is the gist of continuation-passing style.

Combined these results provide a powerful optimisation scheme. Although the categorical results are known, none of the papers cited above seems to note

the intimate relationship. This paper sets out to fill this gap, showing the relevance of the categorical construction to programming. Furthermore, it aims to complement Mac Lane's diagrammatic reasoning by a calculational approach. Specifically, the paper makes the following original contributions:

- we demonstrate that many program optimisations are instances of the optimisation scheme: Church representations etc;
- we develop the calculational properties of Kan extensions, powers and ends;
- to support effective reasoning, we propose a two-dimensional notation for Kan extensions based on string diagrams.

It is the last aspect I am most excited about. The algebra of programming has aptly demonstrated the power of equational reasoning for program calculation. However, one-dimensional notation reaches its limits when it comes to reasoning about natural transformations, as we will set out to do. Natural transformations are a 2-categorical concept, which lends itself naturally to a two-dimensional notation. Many laws, which otherwise have to be invoked explicitly, are built into the notation.

The remainder of the paper is structured as follows. Section 2 introduces some background, notably adjunctions and monads. The knowledgeable reader may safely skip the material, except perhaps for Section 2.2, which introduces string diagrams. Section 3 defines the notion of a Kan extension and suggests a two-dimensional notation based on string diagrams. Section 4 applies the notation to show that every Kan extension induces a monad, the codensity monad. Sections 5 and 6 move on to discuss the existence of Kan extensions. Section 5 proves that every adjunction induces a Kan extension, and that every monad is isomorphic to a codensity monad. Section 6 derives the so-called end formula for Kan extensions. The development requires the categorical notions of powers and ends, which are introduced in Sections 6.1 and 6.2, respectively. The framework has a multitude of applications, which Section 7 investigates. Finally, Section 8 reviews related work and Section 9 concludes.

A basic knowledge of category theory is assumed. Appendix A summarises the main facts about composition of functors and natural transformations.

2 Background

2.1 Adjunction

The notion of an adjunction was introduced by Daniel Kan in 1958 [23]. Adjunctions have proved to be one of the most important ideas in category theory, predominantly due to their ubiquity. Many mathematical constructions turn out to be adjoint functors that form adjunctions, with Mac Lane [27, p.vii] famously saying, "Adjoint functors arise everywhere." From the perspective of program calculation, adjunctions provide a unified framework for program transformation. As with every deep concept, there are various ways to define the notion of an adjunction. The simplest is perhaps the following:

Let $\mathscr{L}$ and $\mathscr{R}$ be categories. The functors $\mathsf{L} : \mathscr{L} \leftarrow \mathscr{R}$ and $\mathsf{R} : \mathscr{L} \rightarrow \mathscr{R}$ are *adjoint*, written $\mathsf{L} \dashv \mathsf{R}$ and depicted

$$\mathscr{L} \underset{\mathsf{R}}{\overset{\mathsf{L}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathscr{R} \ ,$$

if and only if there is a bijection between the hom-sets

$$\lfloor - \rfloor : \mathscr{L}(\mathsf{L}\, A, B) \cong \mathscr{R}(A, \mathsf{R}\, B) : \lceil - \rceil \ , \tag{1}$$

that is natural both in A and B. The functor L is said to be a *left adjoint* for R, while R is L's *right adjoint*. The isomorphism $\lfloor - \rfloor$ is called the *left adjunct* with $\lceil - \rceil$ being the *right adjunct*. (The notation $\lfloor - \rfloor$ for the left adjunct is chosen as the opening bracket resembles an 'L'. Likewise—but this is admittedly a bit laboured—the opening bracket of $\lceil - \rceil$ can be seen as an angular 'r'.)

That $\lfloor - \rfloor$ and $\lceil - \rceil$ are mutually inverse can be captured using an equivalence.

$$f = \lceil g \rceil \iff \lfloor f \rfloor = g \tag{2}$$

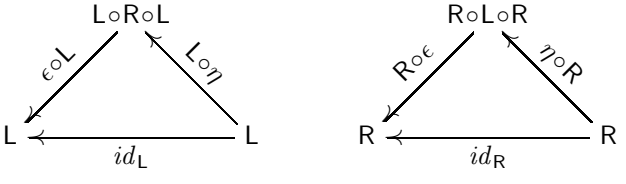
The left-hand side lives in $\mathscr{L}$, and the right-hand side in $\mathscr{R}$.

Let us spell out the naturality properties of the adjuncts: $\lceil g \rceil \cdot \mathsf{L}\, h = \lceil g \cdot h \rceil$ and $\mathsf{R}\, k \cdot \lfloor f \rfloor = \lfloor k \cdot f \rfloor$. The formulæ imply $\lceil id \rceil \cdot \mathsf{L}\, h = \lceil h \rceil$ and $\mathsf{R}\, k \cdot \lfloor id \rfloor = \lfloor k \rfloor$. Consequently, the adjuncts are uniquely defined by their images of the identity: $\epsilon = \lceil id \rceil$ and $\eta = \lfloor id \rfloor$. An alternative definition of adjunctions is based on these two natural transformations, which are called the *counit* $\epsilon : \mathsf{L} \circ \mathsf{R} \dot{\rightarrow} \mathsf{Id}$ and the *unit* $\eta : \mathsf{Id} \dot{\rightarrow} \mathsf{R} \circ \mathsf{L}$ of the adjunction. The units must satisfy the so-called *triangle identities*:

$$\epsilon \circ \mathsf{L} \cdot \mathsf{L} \circ \eta = id_{\mathsf{L}} \ , \tag{3a}$$

$$\mathsf{R} \circ \epsilon \cdot \eta \circ \mathsf{R} = id_{\mathsf{R}} \ . \tag{3b}$$

The diagrammatic rendering explains the name triangle identities.



Remark 1. To understand concepts in category theory it is helpful to look at a simple class of categories: *preorders*, reflexive and transitive relations. Every preorder gives rise to a category whose objects are the elements of the preorder and whose arrows are given by the ordering relation. These categories are special as there is at most one arrow between two objects. Reflexivity provides the identity arrow, transitivity allows us to compose two arrows. A functor between two preorders is a *monotone function*, a mapping on objects that respects the

underlying ordering: $a \leqslant b \Longrightarrow f\ a \leqslant f\ b$. A natural transformation between two monotone functions corresponds to a point-wise ordering: $f \mathrel{\dot{\leqslant}} g \Longleftrightarrow \forall\, x\ .\ f\ x \leqslant g\ x$. When appropriate we shall specialise the development to preorders.

The preorder equivalent of an adjunction is a *Galois connection*. Let L and R be preorders. The maps $l : L \leftarrow R$ and $r : L \rightarrow R$ form a Galois connection between L and R if and only if

$$l\ a \leqslant b \text{ in } L \quad\Longleftrightarrow\quad a \leqslant r\ b \text{ in } R\ , \tag{4}$$

for all $a \in R$ and $b \in L$.

An instructive example of a right adjoint is the floor function $\lfloor - \rfloor : \mathbb{R} \rightarrow \mathbb{Z}$ (not to be confused with the notation for left adjuncts), whose left adjoint is the inclusion map $\iota : \mathbb{Z} \rightarrow \mathbb{R}$. We have

$$\iota\ n \leqslant x \text{ in } \mathbb{R} \quad\Longleftrightarrow\quad n \leqslant \lfloor x \rfloor \text{ in } \mathbb{Z}\ ,$$

for all $n \in \mathbb{Z}$ and $x \in \mathbb{R}$. The inclusion map also has a left adjoint, the ceiling function $\lceil - \rceil : \mathbb{R} \rightarrow \mathbb{Z}$.

The definition of an adjunction in terms of the units corresponds to the following property: the maps $l : L \leftarrow R$ and $r : L \rightarrow R$ form a Galois connection between L and R if and only if l and r are monotone, $l \cdot r \mathrel{\dot{\leqslant}} id$ and $id \mathrel{\dot{\leqslant}} r \cdot l$. Since in a preorder there is at most one arrow between two objects, we furthermore have $r \cdot l \cdot r \mathrel{\dot{\cong}} r$ and $l \mathrel{\dot{\cong}} l \cdot r \cdot l$.

In general, to interpret a category-theoretic result in the setting of preorders, we only consider the types of the arrows: for example, the bijection (1) simplifies to (4). Conversely, an order-theoretic proof can be interpreted as a typing derivation. Category theory has been characterised as *coherently constructive lattice theory* [2], and to generalise an order-theoretic result we additionally have to impose coherence conditions—the triangle identities in the case of adjunctions. □

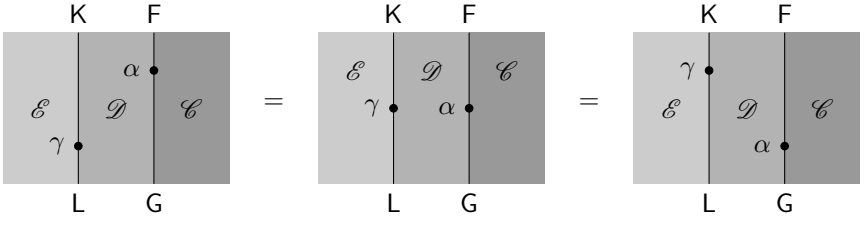
2.2 String Diagram

Throughout the paper we shall recast natural transformations and their properties in two-dimensional notation, based on *string diagrams* [31]. Categories, functors and natural transformations form a so-called 2-category, which lends itself naturally to a two-dimensional notation. From a calculational point of view, two-dimensional notation is attractive because several laws, notably the interchange law (56), are built into the notation. When we use one-dimensional notation, we have to invoke these laws explicitly. (For similar reasons we routinely use one-dimensional notation for objects and arrows: the monoidal properties of identity and composition are built into the notation.)

Here are the string diagrams for the units of an adjunction.

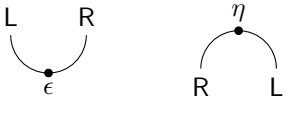
A string diagram is a planar graph. A region in the graph corresponds to a category, a line corresponds to a functor, and a point (usually drawn as a small circle) corresponds to a natural transformation. For readability lines are implicitly directed, and we stipulate that the flow is from right to left for horizontal composition, $\beta \circ \alpha$, and from top to bottom for vertical composition $\beta \cdot \alpha$. The counit ϵ has two incoming functors, L and R, and no outgoing functor—the dotted line hints at the identity functor, which is usually omitted. A natural transformation of type $\mathsf{F} \circ \mathsf{G} \circ \mathsf{H} \dot{\to} \mathsf{T} \circ \mathsf{U}$, for example, would be shown as a point with three incoming arrows (from above) and two outgoing arrows (to below).

Diagrams that differ only in the vertical position of natural transformations are identified—this is the import of the interchange law (56).



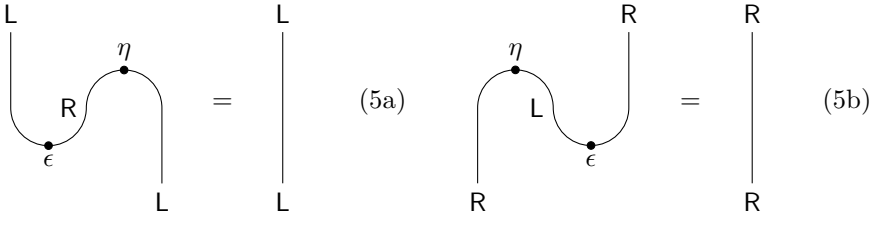
Thus, $\gamma \circ \mathsf{G} \cdot \mathsf{K} \circ \alpha$, $\gamma \circ \alpha$ and $\mathsf{L} \circ \alpha \cdot \gamma \circ \mathsf{F}$ correspond to the same diagram. For turning a string diagram into standard notation it is helpful to draw horizontal lines through the points that denote natural transformations. Each of these lines corresponds to a horizontal composition, where a vertical line that crosses the horizontal line is interpreted as the identity on the respective functor. This step yields $\gamma \circ \mathsf{G}$ and $\mathsf{K} \circ \alpha$ for the diagram on the left. The vertical composition of these terms then corresponds to the diagram.

To reduce clutter we shall usually not label or colour the regions. Also, identity functors (drawn as dotted lines above) and identity natural transformations are omitted. With these conventions the string diagrams for the units simplify to half circles.



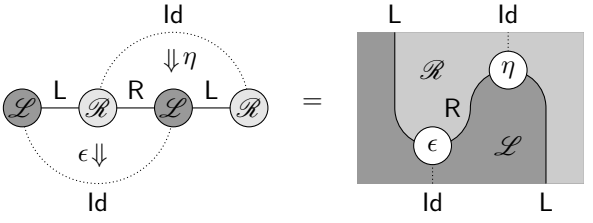
A cup signifies the counit ϵ and a cap the unit η.

It is important to keep in mind that, unlike a commutative diagram, a string diagram is a term, not a property. Properties such as the triangle identities (3a)-(3b) are still written as equations.



The triangle identities have an appealing visual interpretation: they allow us to pull a twisted string straight.

Remark 2. There is an alternative, perhaps more traditional two-dimensional notation, where categories are shown as points, functors as lines and natural transformations as regions (often labelled with a double arrow).



The traditional diagram on the left is the Poincaré dual of the string diagram on the right: d-dimensional objects on the left are mapped to $(2-d)$-dimensional objects on the right, and vice versa. □

2.3 Monad

To incorporate computational effects such as IO, Haskell has adopted the categorical concept of a monad [30]. As with adjunctions, there are several ways to define the notion. The following is known as the monoidal definition.

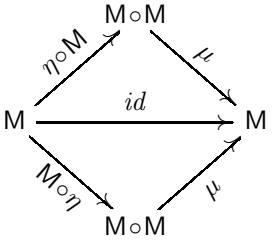
A monad consists of an endofunctor M and natural transformations

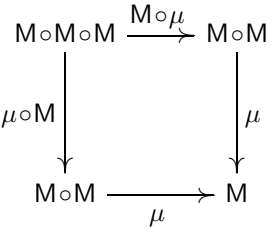
$$\eta : \mathsf{Id} \to \mathsf{M} \ ,$$
$$\mu : \mathsf{M} \circ \mathsf{M} \to \mathsf{M} \ .$$

From the perspective of Haskell, a monad is a mechanism that supports effectful computations. A monadic program is an arrow of type $A \to \mathsf{M}\, B$, where the monad is wrapped around the target. The operations that come with a monad organise effects: the unit η (also called "return") creates a pure computation, the multiplication μ (also called "join") merges two layers of effects. The two operations have to work together:

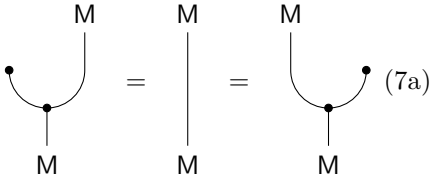
$$\mu \cdot \eta \circ \mathsf{M} = id_{\mathsf{M}} \ , \tag{6a}$$
$$\mu \cdot \mathsf{M} \circ \eta = id_{\mathsf{M}} \ , \tag{6b}$$
$$\mu \cdot \mu \circ \mathsf{M} = \mu \cdot \mathsf{M} \circ \mu \ . \tag{6c}$$

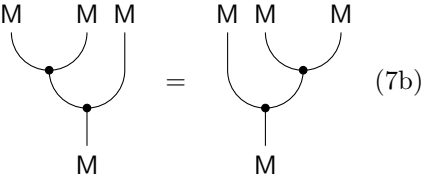
The unit laws (6a) and (6b) state that merging a pure with a potentially effectful computation gives the effectful computation. The associative law (6c) expresses that the two ways of merging three layers of effects are equivalent.





In two-dimensional notation, the natural transformations correspond to constructors of binary leaf trees: η creates a leaf, μ represents a fork. The monad laws correspond to transformations on binary trees: the unit laws allow us to prune or to add leaves and the associative law captures a simple tree rotation.





Every adjunction $\mathsf{L} \dashv \mathsf{R}$ induces a monad [17]:

$$\mathsf{M} = \mathsf{R} \circ \mathsf{L} \ , \tag{8a}$$

$$\eta = \eta \ , \tag{8b}$$

$$\mu = \mathsf{R} \circ \epsilon \circ \mathsf{L} \ . \tag{8c}$$

The monad operations have simple implementations in terms of the units: the unit of the adjunction serves as the unit of the monad; the multiplication is defined in terms of the counit. We will prove this result twice, a first time using one-dimensional notation and a second time using two-dimensional notation.

The unit laws (6a)–(6b) are consequences of the triangle identities (3a)–(3b).

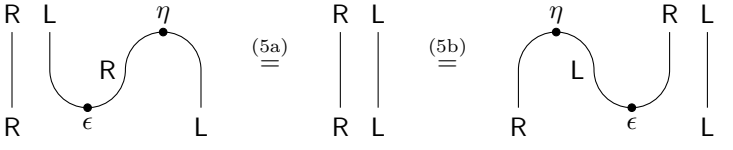
$$\begin{aligned}
& \mu \cdot \eta \circ \mathsf{M} \\
= \ & \{ \text{ definitions (8a)–(8c) } \} \\
& \mathsf{R} \circ \epsilon \circ \mathsf{L} \cdot \eta \circ \mathsf{R} \circ \mathsf{L} \\
= \ & \{ - \circ \mathsf{L} \text{ functor (54c) } \} \\
& (\mathsf{R} \circ \epsilon \cdot \eta \circ \mathsf{R}) \circ \mathsf{L} \\
= \ & \{ \text{ triangle identity (3b) } \} \\
& id_{\mathsf{R}} \circ \mathsf{L} \\
= \ & \{ - \circ \mathsf{L} \text{ functor (54c) } \} \\
& id_{\mathsf{R} \circ \mathsf{L}} \\
= \ & \{ \text{ definition of } \mathsf{M} \text{ (8a) } \} \\
& id_{\mathsf{M}}
\end{aligned}$$

$$\begin{aligned}
& \mu \cdot \mathsf{M} \circ \eta \\
= \ & \{ \text{ definitions (8a)–(8c) } \} \\
& \mathsf{R} \circ \epsilon \circ \mathsf{L} \cdot \mathsf{R} \circ \mathsf{L} \circ \eta \\
= \ & \{ \mathsf{R} \circ - \text{ functor (54a) } \} \\
& \mathsf{R} \circ (\epsilon \circ \mathsf{L} \cdot \mathsf{L} \circ \eta) \\
= \ & \{ \text{ triangle identity (3a) } \} \\
& \mathsf{R} \circ id_{\mathsf{L}} \\
= \ & \{ \mathsf{R} \circ - \text{ functor (54a) } \} \\
& id_{\mathsf{R} \circ \mathsf{L}} \\
= \ & \{ \text{ definition of } \mathsf{M} \text{ (8a) } \} \\
& id_{\mathsf{M}}
\end{aligned}$$

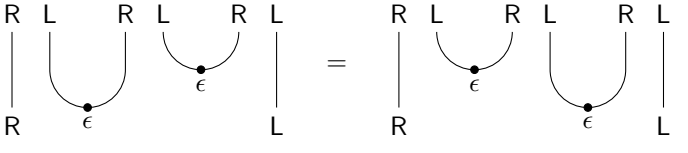
The associative law (6c) follows from the coherence property of horizontal composition, the interchange law (56).

$$
\begin{aligned}
& \mu \cdot \mu \circ \mathsf{M} \\
= \quad & \{ \text{ definition of } \mathsf{M} \text{ (8a) and } \mu \text{ (8c) } \} \\
& \mathsf{R} \circ \epsilon \circ \mathsf{L} \cdot \mathsf{R} \circ \epsilon \circ \mathsf{L} \circ \mathsf{R} \circ \mathsf{L} \\
= \quad & \{ \mathsf{R} \circ - \text{ and } - \circ \mathsf{L} \text{ functors (54b) and (54d) } \} \\
& \mathsf{R} \circ (\epsilon \cdot \epsilon \circ \mathsf{L} \circ \mathsf{R}) \circ \mathsf{L} \\
= \quad & \{ \text{ interchange law (56): } \mathsf{Id} \circ \epsilon \cdot \epsilon \circ (\mathsf{L} \circ \mathsf{R}) = \epsilon \circ \epsilon = \epsilon \circ \mathsf{Id} \cdot (\mathsf{L} \circ \mathsf{R}) \circ \epsilon \ \} \\
& \mathsf{R} \circ (\epsilon \cdot \mathsf{L} \circ \mathsf{R} \circ \epsilon) \circ \mathsf{L} \\
= \quad & \{ \mathsf{R} \circ - \text{ and } - \circ \mathsf{L} \text{ functors (54b) and (54d) } \} \\
& \mathsf{R} \circ \epsilon \circ \mathsf{L} \cdot \mathsf{R} \circ \mathsf{L} \circ \mathsf{R} \circ \epsilon \circ \mathsf{L} \\
= \quad & \{ \text{ definition of } \mathsf{M} \text{ (8a) and } \mu \text{ (8c) } \} \\
& \mu \cdot \mathsf{M} \circ \mu
\end{aligned}
$$

The proofs using one-dimensional notation exhibit a lot of noise. In contrast, the proofs in two-dimensional notation carve out the essential steps. For the unit laws, we use the triangle identities.



The associative law requires no proof as the diagrams for the left- and the right-hand side are identified.



In other words, the one-dimensional proof only contains administrative steps.

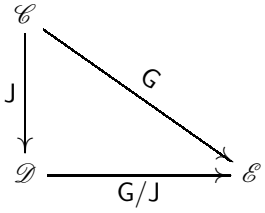
Remark 3. The preorder equivalent of a monad is a *closure operator*. Let P be a preorder. A map $m : P \to P$ is a closure operator on P if it is extensive, $id \mathrel{\dot{\leqslant}} m$, and idempotent, $m \cdot m \mathrel{\dot{\cong}} m$. (The latter condition can be weakened to $m \cdot m \mathrel{\dot{\leqslant}} m$ since $m \mathrel{\dot{\leqslant}} m \cdot id \mathrel{\dot{\leqslant}} m \cdot m$ as composition is monotone.)

A Galois connection $l \dashv r$ between L and R induces a closure operator $m = r \cdot l$ on R. For example, the composition of inclusion $\iota : \mathbb{Z} \to \mathbb{R}$ and the ceiling function $\lceil - \rceil : \mathbb{R} \to \mathbb{Z}$ is a closure operator on $\mathbb{R}$. □

3 Kan Extension—Specification

The continuation types shown in the introduction *implement* so-called right Kan extensions. This section *specifies* the concept formally. As to be expected, the specification will be quite different from the implementation.

Let $\mathsf{J} : \mathscr{C} \to \mathscr{D}$ be a functor. You may want to think of J as an inclusion functor. The functor part of the right Kan extension $\mathsf{G/J} : \mathscr{D} \to \mathscr{E}$ extends a functor $\mathsf{G} : \mathscr{C} \to \mathscr{E}$ to the whole of $\mathscr{D}$.



It is worth pointing out that the functors J and G play quite different roles (see also Remark 4), which is why $\mathsf{G/J}$ is called the right Kan extension *of* G *along* J. The notation $\mathsf{G/J}$ is taken from relation algebra (see also Remark 5) and emphasises the algebraic properties of Kan extensions. (Mac Lane [27] writes $\mathsf{Ran_J\ G}$ for right Kan extensions and $\mathsf{Lan_J\ G}$ for left ones, a notation we do not use). Again, there are various ways to define the concept. The shortest is this:

The functor $\mathsf{G/J}$ is the (functor part of the) *right Kan extension of* G *along* J if and only if there is a bijection between the hom-sets

$$\mathscr{E}^{\mathscr{C}}(\mathsf{F}\circ\mathsf{J},\mathsf{G}) \cong \mathscr{E}^{\mathscr{D}}(\mathsf{F},\mathsf{G/J}) \ , \tag{9}$$

that is natural in the functor $\mathsf{F} : \mathscr{D} \to \mathscr{E}$.

If we instantiate the bijection to $\mathsf{F} := \mathsf{G/J}$, we obtain as the image of the identity $id : \mathscr{E}^{\mathscr{D}}(\mathsf{G/J},\mathsf{G/J})$ a natural transformation $run : \mathscr{E}^{\mathscr{C}}((\mathsf{G/J})\circ\mathsf{J},\mathsf{G})$. The transformation eliminates a right Kan extension and is called the *unit* of the extension. An alternative definition of Kan extensions builds solely on the unit, which is an example of a universal arrow:

The *right Kan extension of* G *along* J consists of a functor written $\mathsf{G/J} : \mathscr{D} \to \mathscr{E}$ and a natural transformation $run : \mathscr{E}^{\mathscr{C}}((\mathsf{G/J})\circ\mathsf{J},\mathsf{G})$. These two things have to satisfy the following *universal property*: for each functor $\mathsf{F} : \mathscr{D} \to \mathscr{E}$ and for each natural transformation $\alpha : \mathscr{E}^{\mathscr{C}}(\mathsf{F}\circ\mathsf{J},\mathsf{G})$ there exists a natural transformation $[\alpha] : \mathscr{E}^{\mathscr{D}}(\mathsf{F},\mathsf{G/J})$ (pronounce "shift α") such that

$$\alpha = run \cdot \beta\circ\mathsf{J} \iff [\alpha] = \beta \ , \tag{10}$$

for all $\beta : \mathscr{E}^{\mathscr{D}}(\mathsf{F},\mathsf{G/J})$. The equivalence witnesses the bijection (9) and expresses that there is a unique way to factor α into a composition of the form $run \cdot \beta\circ\mathsf{J}$.

A universal property such as (10) has three immediate consequences that are worth singling out. If we substitute the right-hand side into the left-hand side, we obtain the *computation law*:

$$\alpha = run \cdot [\alpha]\circ\mathsf{J} \ . \tag{11}$$

Instantiating β in (10) to the identity $id_{\mathsf{G}/\mathsf{J}}$ and substituting the left- into the right-hand side, yields the *reflection law*:

$$[run] = id \ . \tag{12}$$

Finally, the *fusion law* allows us to fuse a shift with a natural transformation to form another shift:

$$[\alpha] \cdot \gamma = [\alpha \cdot \gamma \circ \mathsf{J}] \ , \tag{13}$$

for all $\gamma : \mathscr{E}^{\mathscr{D}}(\hat{\mathsf{F}}, \check{\mathsf{F}})$. The fusion law states that shift is natural in the functor F. For the proof we reason

$$\begin{array}{ll} & [\alpha] \cdot \gamma = [\alpha \cdot \gamma \circ \mathsf{J}] \\ \Longleftrightarrow & \{ \text{ universal property (10) } \} \\ & \alpha \cdot \gamma \circ \mathsf{J} = run \cdot ([\alpha] \cdot \gamma) \circ \mathsf{J} \\ \Longleftrightarrow & \{ \ - \circ \mathsf{J} \text{ functor (54d) } \} \\ & \alpha \cdot \gamma \circ \mathsf{J} = run \cdot [\alpha] \circ \mathsf{J} \cdot \gamma \circ \mathsf{J} \\ \Longleftrightarrow & \{ \text{ computation (11) } \} \\ & \alpha \cdot \gamma \circ \mathsf{J} = \alpha \cdot \gamma \circ \mathsf{J} \ . \end{array}$$

As all universal concepts, right Kan extensions are unique up to isomorphism. This is a consequence of naturality: let $\mathsf{G}/_1\mathsf{J}$ and $\mathsf{G}/_2\mathsf{J}$ be two Kan extensions. Since the string of isomorphisms

$$\mathscr{E}^{\mathscr{D}}(\mathsf{F}, \mathsf{G}/_1\mathsf{J}) \cong \mathscr{E}^{\mathscr{C}}(\mathsf{F} \circ \mathsf{J}, \mathsf{G}) \cong \mathscr{E}^{\mathscr{D}}(\mathsf{F}, \mathsf{G}/_2\mathsf{J})$$

is natural in F, the principle of indirect proof [15] implies that $\mathsf{G}/_1\mathsf{J} \cong \mathsf{G}/_2\mathsf{J}$. There is also a simple calculational proof, which nicely serves to illustrate the laws above. The isomorphism is given by

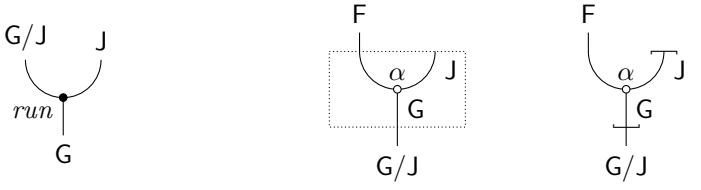
$$[run_1]_2 : \mathsf{G}/_1\mathsf{J} \cong \mathsf{G}/_2\mathsf{J} : [run_2]_1 \ . \tag{14}$$

We show $[run_1]_2 \cdot [run_2]_1 = id$. The proof of the other half proceeds completely analogously.

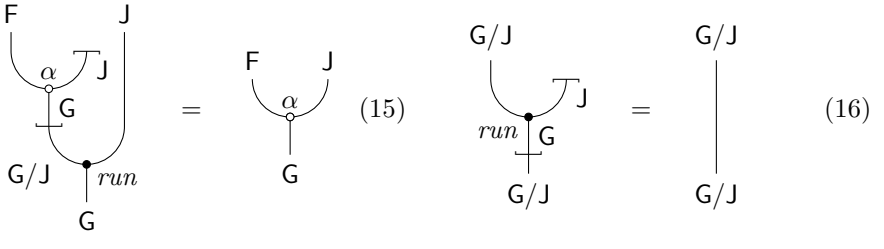
$$\begin{array}{ll} & [run_1]_2 \cdot [run_2]_1 \\ = & \{ \text{ fusion (13) } \} \\ & [run_1 \cdot [run_2]_1 \circ \mathsf{J}]_2 \\ = & \{ \text{ computation (11) } \} \\ & [run_2]_2 \\ = & \{ \text{ reflection (12) } \} \\ & id \end{array}$$

Remark 4. If the Kan extension along J exists for every G, then $-/\mathsf{J}$ itself can be turned into a functor, so that the bijection $\mathscr{E}^{\mathscr{C}}(\mathsf{F}\circ\mathsf{J},\mathsf{G}) \cong \mathscr{E}^{\mathscr{D}}(\mathsf{F},\mathsf{G}/\mathsf{J})$ is also natural in G. In other words, we have an adjunction $-\circ\mathsf{J} \dashv -/\mathsf{J}$. If furthermore the adjunction $-\circ\mathsf{J} \dashv -/\mathsf{J}$ exists for every J—we have an adjunction with a parameter—then there is a unique way to turn $=/-$ into a higher-order bifunctor of type $(\mathscr{D}^{\mathscr{C}})^{\mathsf{op}} \times \mathscr{E}^{\mathscr{C}} \to \mathscr{E}^{\mathscr{D}}$, so that the bijection is also natural in J [27, Th. IV.7.3, p102]. □

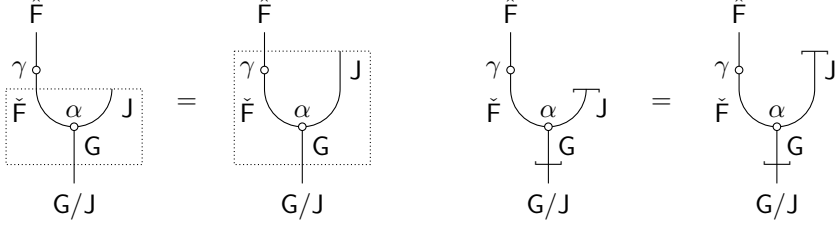
Turning to the two-dimensional notation, the unit *run* is drawn as a *solid* circle • (diagram on the left below). This convention allows us to omit the label *run* to avoid clutter. More interesting is the diagrammatic rendering of $[\alpha]$. My first impulse was to draw a dotted box around α, pruning J and relabelling G to G/J (diagram in the middle).



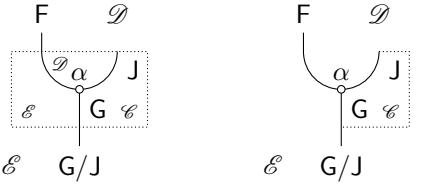
However, as we shall see in a moment, the diagram on the right is a better choice. The F branch is left untouched; the J and G branches are enclosed in square brackets (⌐ and ¬). Computation (11) and reflection (12) are then rendered as follows.



Seen as a graph transformation, the computation law (15) allows us to replace the node labelled *run* by the sub-graph α. The reflection law (16) means that we can cut off a 'dead branch', a *run* node enclosed in brackets. The fusion law is the most interesting one—it shows the advantage of the bracket notation.



If we used the box notation, then fusion would allow us to shift γ in and out of the box. The bracket notation on the other hand incorporates the fusion law—recall that diagrams that differ only in the vertical position of natural transformations are identified—and is thus our preferred choice. (As an aside, the box notation is advantageous if we choose to label the regions: the category inside the box to the right of J and G is $\mathscr{C}$, whereas the category outside the box to the right of F and G/J is $\mathscr{D}$. A compromise is to draw only the lower right compartment of the box as shown on the right below.)



It is important to note that the computation law (15) contains universally quantified variables: it holds for all functors F and for all natural transformations α, the latter denoted by a *hollow* circle $\circ$ for emphasis. This is in contrast to all of the other two-dimensional laws we have seen before: the triangle identities and the monad laws involve only constants. When the computation law (15) is invoked, we have to substitute a subgraph for α and a bundle of strings for F. In particular, if F is replaced by Id, then the bundle is empty. The two-dimensional matching process is actually not too difficult: essentially one has to watch out for a solid circle ($\bullet$) to the right below of a closing bracket (—).

Finally, let us record that the diagrammatic reasoning is complete since computation, reflection and fusion imply the universal property (10). '$\Longleftarrow$': This implication amounts to the computation law (11). '$\Longrightarrow$': We reason

$$
\begin{aligned}
& [\mathit{run} \cdot \beta \circ \mathsf{J}] \\
= \quad & \{ \text{ fusion (13) } \} \\
& [\mathit{run}] \cdot \beta \\
= \quad & \{ \text{ reflection (12) } \} \\
& \beta \ .
\end{aligned}
$$

Remark 5. We can specialise Kan extensions to the preorder setting, if we equip a preorder with a monoidal structure: an associative operation that is monotone and that has a neutral element. Consider as an example the integers equipped with multiplication $*$. The bijection (9) then corresponds to the equivalence

$$m * k \leqslant n \quad \Longleftrightarrow \quad m \leqslant n \div k \ ,$$

which specifies integer division $\div$ for $k > 0$. The equivalence uniquely defines division since the ordering relation is antisymmetric. The notation for Kan extensions is, in fact, inspired by this instance.

We obtain more interesting examples if we generalise monoids to categories. For instance, Kan extensions correspond to so-called *factors* in relation algebra, which are also known as residuals or weakest postspecifications [16].

$$F \cdot J \subseteq G \iff F \subseteq G \,/\, J \tag{17}$$

Informally, $G \,/\, J$ is the weakest (most general) postspecification that approximates G after specification J has been met. Again, the universal property uniquely defines $G \,/\, J$ since the subset relation is antisymmetric. The type of *run* corresponds to the computation law

$$(G \,/\, J) \cdot J \subseteq G \ . \tag{18}$$

(Kan extensions, integer quotients and factors are, in fact, instances of a more general 2-categorical concept. Actually, the development in this and in the following two sections can be readily generalised to 2-categories. Relation algebra is a simple instance of a 2-category where the vertical categories are preorders. A 'monoidal preorder' such as the integers with multiplication is an even simpler instance where the horizontal category is a monoid and *the* vertical category is a preorder.) □

4 Codensity Monad

The right Kan extension of J along J is a monad, $\mathsf{M} = \mathsf{J}/\mathsf{J}$, the so-called *codensity monad of* J. To motivate the definition of the monad operations, let us instantiate the Kan bijection (9) to $\mathsf{G} := \mathsf{J}$:

$$\mathscr{D}^{\mathscr{C}}(\mathsf{F} \circ \mathsf{J}, \mathsf{J}) \cong \mathscr{D}^{\mathscr{D}}(\mathsf{F}, \mathsf{M}) \ . \tag{19}$$

Recall that the bijection is natural in the functor F. For the return of the monad we set F to the identity functor, which suggests that return is just the transpose of the identity. The unit *run* of the Kan extension has type $\mathsf{M} \circ \mathsf{J} \dot{\to} \mathsf{J}$. To define the multiplication of the monad, we instantiate F to $\mathsf{M} \circ \mathsf{M}$, which leaves us with the task of providing a natural transformation of type $\mathsf{M} \circ \mathsf{M} \circ \mathsf{J} \dot{\to} \mathsf{J}$: the composition $run \cdot \mathsf{M} \circ run$ will do nicely. To summarise, the codensity monad of J is given by

$$\mathsf{M} = \mathsf{J}/\mathsf{J} \ , \tag{20a}$$

$$\eta = [id] \ , \tag{20b}$$

$$\mu = [run \cdot \mathsf{M} \circ run] \ . \tag{20c}$$

Of course, we have to show that the data satisfies the monad laws. As in the previous section, we provide two proofs, one using traditional notation and one using two-dimensional notation.

For the unit laws (6a)–(6b) we reason

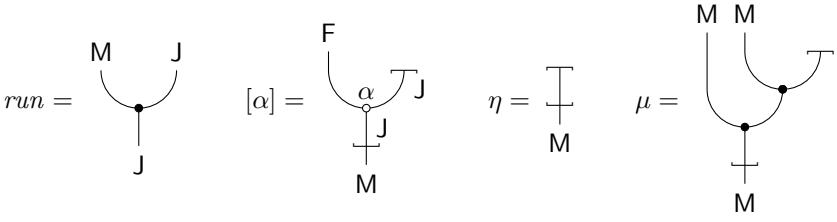
$$
\begin{array}{ll}
 & \mu \cdot \eta \circ \mathsf{M} \\
= & \quad \{ \text{ definition of } \mu \text{ (20c) } \} \\
 & [run \cdot \mathsf{M} \circ run] \cdot \eta \circ \mathsf{M} \\
= & \quad \{ \text{ fusion (13) } \} \\
 & [run \cdot \mathsf{M} \circ run \cdot \eta \circ \mathsf{M} \circ \mathsf{J}] \\
= & \quad \{ \text{ interchange law (56) } \} \\
 & [run \cdot \eta \circ \mathsf{J} \cdot run] \\
= & \quad \{ \text{ definition of } \eta \text{ (20b) } \} \\
 & [run \cdot [id] \circ \mathsf{J} \cdot run] \\
= & \quad \{ \text{ computation (11) } \} \\
 & [run] \\
= & \quad \{ \text{ reflection (12) } \} \\
 & id \ ,
\end{array}
\qquad
\begin{array}{ll}
 & \mu \cdot \mathsf{M} \circ \eta \\
= & \quad \{ \text{ definition of } \mu \text{ (20c) } \} \\
 & [run \cdot \mathsf{M} \circ run] \cdot \mathsf{M} \circ \eta \\
= & \quad \{ \text{ fusion (13) } \} \\
 & [run \cdot \mathsf{M} \circ run \cdot \mathsf{M} \circ \eta \circ \mathsf{J}] \\
= & \quad \{ \mathsf{M} \circ - \text{ functor (54b) } \} \\
 & [run \cdot \mathsf{M} \circ (run \cdot \eta \circ \mathsf{J})] \\
= & \quad \{ \text{ definition of } \eta \text{ (20b) } \} \\
 & [run \cdot \mathsf{M} \circ (run \cdot [id] \circ \mathsf{J})] \\
= & \quad \{ \text{ computation (11) } \} \\
 & [run \cdot \mathsf{M} \circ id] \\
= & \quad \{ \mathsf{M} \circ - \text{ functor (54a) } \} \\
 & [run] \\
= & \quad \{ \text{ reflection (12) } \} \\
 & id \ .
\end{array}
$$

All of the basic identities are used: reflection (12), computation (11) and fusion (13).

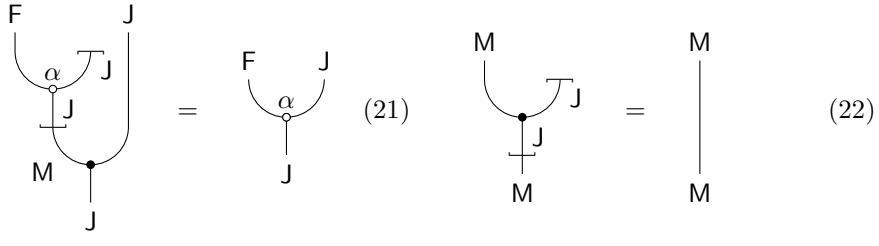
For the associative law (6c) we show that both sides of the equation simplify to $[run \cdot \mathsf{M} \circ run \cdot \mathsf{M} \circ \mathsf{M} \circ run]$, which merges three layers of effects:

$$
\begin{array}{ll}
 & \mu \cdot \mu \circ \mathsf{M} \\
= & \quad \{ \text{ definition of } \mu \text{ (20c) } \} \\
 & [run \cdot \mathsf{M} \circ run] \cdot \mu \circ \mathsf{M} \\
= & \quad \{ \text{ fusion (13) } \} \\
 & [run \cdot \mathsf{M} \circ run \cdot \mu \circ \mathsf{M} \circ \mathsf{J}] \\
= & \quad \{ \text{ interchange law (56) } \} \\
 & [run \cdot \mu \circ \mathsf{J} \cdot \mathsf{M} \circ \mathsf{M} \circ run] \\
= & \quad \{ \text{ definition of } \mu \text{ (20c) } \} \\
 & [run \cdot [run \cdot \mathsf{M} \circ run] \circ \mathsf{J} \cdot \mathsf{M} \circ \mathsf{M} \circ run] \\
= & \quad \{ \text{ computation (11) } \} \\
 & [run \cdot \mathsf{M} \circ run \cdot \mathsf{M} \circ \mathsf{M} \circ run]
\end{array}
\qquad
\begin{array}{ll}
 & \mu \cdot \mathsf{M} \circ \mu \\
= & \quad \{ \text{ definition of } \mu \text{ (20c) } \} \\
 & [run \cdot \mathsf{M} \circ run] \cdot \mathsf{M} \circ \mu \\
= & \quad \{ \text{ fusion (13) } \} \\
 & [run \cdot \mathsf{M} \circ run \cdot \mathsf{M} \circ \mu \circ \mathsf{J}] \\
= & \quad \{ \mathsf{M} \circ - \text{ functor (54b) } \} \\
 & [run \cdot \mathsf{M} \circ (run \cdot \mu \circ \mathsf{J})] \\
= & \quad \{ \text{ definition of } \mu \text{ (20c) } \} \\
 & [run \cdot \mathsf{M} \circ (run \cdot [run \cdot \mathsf{M} \circ run] \circ \mathsf{J})] \\
= & \quad \{ \text{ computation (11) } \} \\
 & [run \cdot \mathsf{M} \circ (run \cdot \mathsf{M} \circ run)] \\
= & \quad \{ \mathsf{M} \circ - \text{ functor (54b) } \} \\
 & [run \cdot \mathsf{M} \circ run \cdot \mathsf{M} \circ \mathsf{M} \circ run] \ .
\end{array}
$$

Turning to the second set of proofs, here are the two-dimensional counterparts of the natural transformations involved.

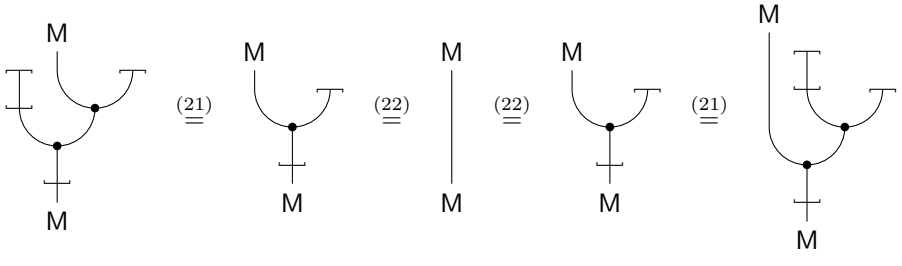


Diagrammatically, η indeed resembles a leaf, whereas μ is a nested fork with one branch cut off. For the calculations it is useful to specialise the computation law (15) and the reflection law (16) to $\mathsf{G} := \mathsf{J}$.



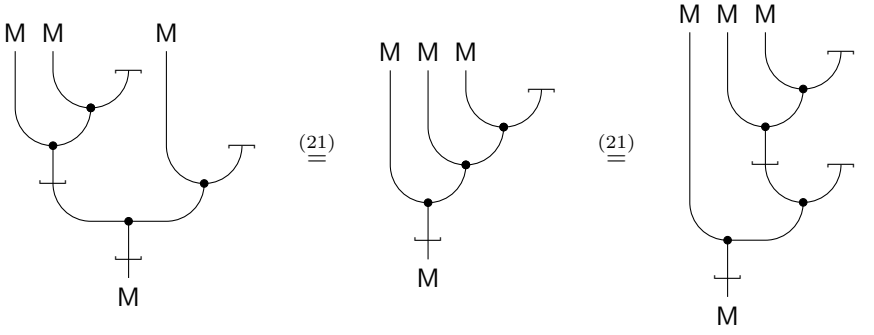
Recall that (21) holds for all functors F and for all natural transformations α.

Now, to show the unit laws we simply combine computation and reflection.



To follow the graph transformations involving (21), first identify the occurrence of a closing bracket (⌙) which leads to a solid circle (•) below. Then replace the solid circle (•) by the graph enclosed in ⌜ and ⌙, additionally removing the brackets. The instances of (21) above are in a sense extreme: F is in both cases instantiated to Id and α to id.

For the proof of associativity, we invoke the computation law twice.



Now, F is instantiated to $\mathsf{M}\circ\mathsf{M}$ and α to $run \cdot \mathsf{M}\circ run$. Again, the two-dimensional proofs carve out the essential steps.

Remark 6. Continuing Remark 5, let us specialise the above to relational algebra. The factor $J \,/\, J$ is a closure operator. The proofs correspond to the typing derivations of $\eta = [id]$ and $\mu = [run \cdot \mathsf{M}\circ run]$. Specifically, to prove $Id \subseteq J \,/\, J$ we appeal to the universal property (17) which leaves us with $Id \cdot J \subseteq J$. Likewise, to prove $(J \,/\, J) \cdot (J \,/\, J) \subseteq (J \,/\, J)$ if suffices to show that $(J \,/\, J) \cdot (J \,/\, J) \cdot J \subseteq J$. The obligation can be discharged using the computation law (18), twice. □

5 Absolute Kan Extension—Implementation

So far we have been concerned with general properties of right Kan extensions. Let us now turn our attention to the existence of extensions, which in computer-science terms is the implementation. A general result is this: if R is a right adjoint, then the right Kan extension along R exists for any functor G. To prove the result we take a short detour.

Adjunctions can be lifted to functor categories: if $\mathsf{L} \dashv \mathsf{R}$ is an adjunction then both $\mathsf{L}\circ- \dashv \mathsf{R}\circ-$ and $-\circ\mathsf{R} \dashv -\circ\mathsf{L}$ are adjunctions. (Recall that both $\mathsf{K}\circ-$ and $-\circ\mathsf{E}$ are functors, see Appendix A.) Since pre-composition is post-composition in the opposite category, the two statements are actually dual—note that L and R are flipped in the adjunction $-\circ\mathsf{R} \dashv -\circ\mathsf{L}$. For reasons to become clear in a moment, let us focus on pre-composition:

$$\text{if} \qquad \mathscr{L} \;\underset{\mathsf{R}}{\overset{\mathsf{L}}{\leftarrows \bot \rightarrow}}\; \mathscr{R} \qquad \text{then} \qquad \mathscr{X}^{\mathscr{R}} \;\underset{-\circ\mathsf{L}}{\overset{-\circ\mathsf{R}}{\leftarrows \bot \rightarrow}}\; \mathscr{X}^{\mathscr{L}} \;.$$

For the proof of this fact we establish the equivalence

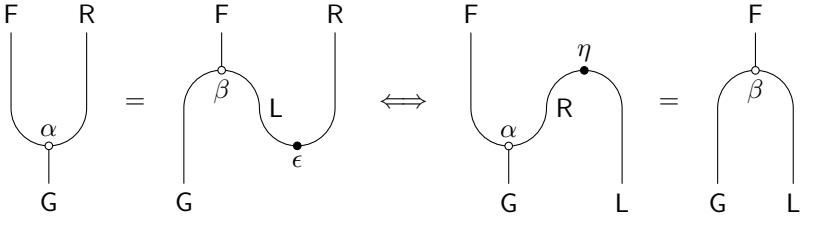
$$\alpha = \mathsf{G}\circ\epsilon \cdot \beta\circ\mathsf{R} \quad \Longleftrightarrow \quad \alpha\circ\mathsf{L} \cdot \mathsf{F}\circ\eta = \beta \;, \tag{23}$$

for all functors F and G and for all natural transformations $\alpha : \mathsf{F}\circ\mathsf{R} \to \mathsf{G}$ and $\beta : \mathsf{F} \to \mathsf{G}\circ\mathsf{L}$. This equivalence amounts to (2) phrased in terms of the units.

We show the implication from left to right, the proof for the opposite direction proceeds completely analogously.

$$
\begin{aligned}
& (\mathsf{G}\circ\epsilon \cdot \beta\circ\mathsf{R})\circ\mathsf{L} \cdot \mathsf{F}\circ\eta \\
= \quad & \{\ -\circ\mathsf{L} \text{ functor (54d)}\ \} \\
& \mathsf{G}\circ\epsilon\circ\mathsf{L} \cdot \beta\circ\mathsf{R}\circ\mathsf{L} \cdot \mathsf{F}\circ\eta \\
= \quad & \{\ \text{interchange law (56)}\ \} \\
& \mathsf{G}\circ\epsilon\circ\mathsf{L} \cdot \mathsf{G}\circ\mathsf{L}\circ\eta \cdot \beta \\
= \quad & \{\ \mathsf{G}\circ- \text{ functor (54b)}\ \} \\
& \mathsf{G}\circ(\epsilon\circ\mathsf{L} \cdot \mathsf{L}\circ\eta) \cdot \beta \\
= \quad & \{\ \text{assumption: triangle identity (3a)}\ \} \\
& \mathsf{G}\circ id_{\mathsf{L}} \cdot \beta \\
= \quad & \{\ \text{identity}\ \} \\
& \beta
\end{aligned}
$$

If we write the equivalence (23) using two-dimensional notation,



then the proof becomes more perspicuous. For the left-to-right direction we focus on the left-hand side of the equivalence and put a cap on the R branches (on both sides of the equation) and then pull the L string straight down (on the right-hand side of the equation). Conversely, for the right-to-left direction we place a cup below the L branches and then pull the R string straight up.

Returning to the original question of existence of right Kan extensions, we have established

$$\mathscr{X}^{\mathscr{R}}(\mathsf{F}\circ\mathsf{R}, \mathsf{G}) \cong \mathscr{X}^{\mathscr{L}}(\mathsf{F}, \mathsf{G}\circ\mathsf{L}) \ , \tag{24}$$

which is an instance of the Kan bijection (9). In other words, $\mathsf{G}\circ\mathsf{L}$ is the right Kan extension of G along R. To bring the definition of unit and shift to light, we align the equivalence (23) with the universal property of right Kan extensions (10). We obtain

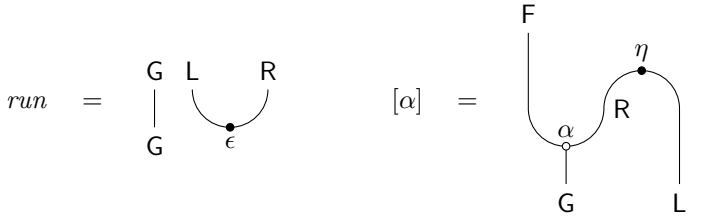
$$\mathsf{G}/\mathsf{R} = \mathsf{G}\circ\mathsf{L} \ , \tag{25a}$$

$$run = \mathsf{G}\circ\epsilon \ , \tag{25b}$$

$$[\alpha] = \alpha\circ\mathsf{L} \cdot \mathsf{F}\circ\eta \ . \tag{25c}$$

Since the bijection (24) is also natural in G, the right Kan extension along R exists for every G.

The unit *run* and $[\alpha]$ are rendered as follows.



To shift $\alpha : \mathsf{F} \circ \mathsf{R} \dot{\to} \mathsf{G}$ we simply put a cap on the rightmost branch labelled R.

Two special cases of (25a) are worth singling out: $\mathsf{L} = \mathsf{Id}/\mathsf{R}$ and $\mathsf{R} \circ \mathsf{L} = \mathsf{R}/\mathsf{R}$. Thus, the *left* adjoint can be expressed as a *right* Kan extension—$\mathsf{L} = \mathsf{Id}/\mathsf{R}$ is a so-called *absolute Kan extension* [27, p.249]. Very briefly, G/J is an absolute Kan extension if and only if it is preserved by any functor: $\mathsf{F} \circ (\mathsf{G}/\mathsf{J})$ and $\mathsf{F} \circ \mathit{run}$ is the Kan extension of $\mathsf{F} \circ \mathsf{G}$ along J. The associativity of horizontal composition implies that the Kan extension $\mathsf{G} \circ \mathsf{R}$ is indeed absolute. Moreover, the monad induced by the adjunction $\mathsf{L} \dashv \mathsf{R}$ coincides with the codensity monad of R:

$$(\mathsf{R} \circ \mathsf{L}, \eta, \mathsf{R} \circ \epsilon \circ \mathsf{L}) = (\mathsf{R}/\mathsf{R}, [\mathit{id}], [\mathit{run} \cdot \mathsf{M} \circ \mathit{run}]) \ . \tag{26}$$

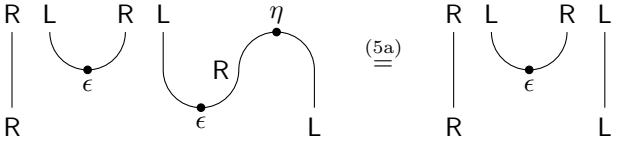
It remains to show that the two alternative definitions of unit and multiplication actually coincide. For the unit, the proof is straightforward.

$$\begin{aligned}
& [\mathit{id}] \\
= \ & \{ \text{ definition of } [-] \text{ (25c) } \} \\
& \mathit{id} \circ \mathsf{L} \cdot \mathsf{Id} \circ \eta \\
= \ & \{ \text{ identity (54c) and (55d) } \} \\
& \eta
\end{aligned}$$

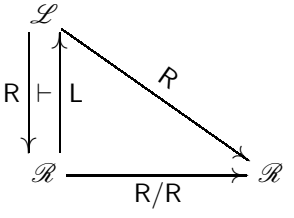
The proof for the multiplication rests on the triangle identity (3b).

$$\begin{aligned}
& [\mathit{run} \cdot \mathsf{M} \circ \mathit{run}] \\
= \ & \{ \text{ definition of } [-] \text{ (25c) } \} \\
& (\mathit{run} \cdot \mathsf{M} \circ \mathit{run}) \circ \mathsf{L} \cdot \mathsf{M} \circ \mathsf{M} \circ \eta \\
= \ & \{ \ - \circ \mathsf{L} \text{ functor (54d) } \} \\
& \mathit{run} \circ \mathsf{L} \cdot \mathsf{M} \circ \mathit{run} \circ \mathsf{L} \cdot \mathsf{M} \circ \mathsf{M} \circ \eta \\
= \ & \{ \text{ definition of } \mathit{run} \text{ (25b) } \} \\
& \mathsf{R} \circ \epsilon \circ \mathsf{L} \cdot \mathsf{M} \circ \mathsf{R} \circ \epsilon \circ \mathsf{L} \cdot \mathsf{M} \circ \mathsf{M} \circ \eta \\
= \ & \{ \text{ definition of } \mathsf{M} \text{ (25a) } \} \\
& \mathsf{R} \circ \epsilon \circ \mathsf{L} \cdot \mathsf{M} \circ \mathsf{R} \circ \epsilon \circ \mathsf{L} \cdot \mathsf{M} \circ \mathsf{R} \circ \mathsf{L} \circ \eta \\
= \ & \{ \ \mathsf{M} \circ \mathsf{R} \circ - \text{ functor (54b) } \} \\
& \mathsf{R} \circ \epsilon \circ \mathsf{L} \cdot \mathsf{M} \circ \mathsf{R} \circ (\epsilon \circ \mathsf{L} \cdot \mathsf{L} \circ \eta) \\
= \ & \{ \text{ triangle identity (3b) } \} \\
& \mathsf{R} \circ \epsilon \circ \mathsf{L}
\end{aligned}$$

As before, the two-dimensional proofs are much shorter. For the unit, $\mathsf{F} := \mathsf{Id}$, there is nothing to do—putting a cap on id_{R} is just the cap. For the multiplication, $\mathsf{F} := \mathsf{R} \circ \mathsf{L} \circ \mathsf{R} \circ \mathsf{L}$, the proof collapses to a single application of the triangle identity—note that $run \cdot \mathsf{M} \circ run = \mathsf{R} \circ \epsilon \circ \epsilon$.



As an intermediate summary, we have shown that every monad is isomorphic to a codensity monad! This perhaps surprising result follows from the fact that every monad is induced by an adjoint pair of functors—this was shown independently by Kleisli [24] and Eilenberg and Moore [10].



Remark 7. To connect the development above to relation algebra, we first have to adopt the notion of an adjunction. The relations L and R are adjoint if and only if $L \cdot R \subseteq Id$ and $Id \subseteq R \cdot L$. In relation algebra this implies $R = L^\circ$, where $(-)^\circ$ is the converse operator. Thus, left adjoints are exactly the functions, simple and entire arrows (denoted by a lower-case letter below). The lifting of adjunctions to functor categories corresponds to the so-called *shunting rules* for functions [3].

$$\begin{aligned} l \cdot F \subseteq G \quad &\Longleftrightarrow \quad F \subseteq l^\circ \cdot G \\ F \cdot l^\circ \subseteq G \quad &\Longleftrightarrow \quad F \subseteq G \cdot l \end{aligned}$$

Specifically, bijection (24) corresponds to the latter equivalence. Using the principle of indirect proof, it is then straightforward to show that $G \cdot l = G \,/\, l^\circ$. In particular, $l = Id \,/\, l^\circ$ and $l^\circ \cdot l = l^\circ \,/\, l^\circ$. □

6 Kan Extension as an End—Implementation

Let us now turn to the heart of the matter. There is an elegant formula, the end formula, which describes right Kan extensions in terms of powers and ends [27, p.242].

$$(\mathsf{G}/\mathsf{J})\,(A : \mathscr{D}) = \forall Z : \mathscr{C} \;.\; \Pi\, \mathscr{D}(A, \mathsf{J}\, Z) \;.\; \mathsf{G}\, Z \tag{27}$$

The object on the right, which lives in $\mathscr{E}$, can be interpreted as a generalised continuation type. This can be seen more clearly if we write both the hom-set

$\mathscr{D}(A, \mathsf{J}\,Z)$ and the power $\Pi\,\mathscr{D}(A, \mathsf{J}\,Z)\ .\ \mathsf{G}\,Z$ as function spaces: $(A \to \mathsf{J}\,Z) \to \mathsf{G}\,Z$. Informally, an element of $(\mathsf{G}/\mathsf{J})\,A$ is a polymorphic function that given a continuation of type $A \to \mathsf{J}\,Z$ yields an element of type $\mathsf{G}\,Z$ for all Z.

The purpose of this section is to prove the end formula and to derive the associated implementations of *run* and shift. To keep the paper sufficiently self-contained, we first introduce powers in Section 6.1 and ends in Section 6.2. (Some reviewers wondered why string diagrams do not appear beyond this point. The reason is simple: to be able to use string diagrams we have to know that the entities involved are functors and natural transformations. Here we set out to establish these properties. More pointedly, we use string diagrams if we wish to prove something against the *specification* of Kan extensions. Here, we aim to to prove an *implementation* correct.) The reader who is not interested in the details may wish to skip to Section 7, which investigates applications of the framework.

Remark 8. The end formula can be *derived* using a calculus of ends [27]—the calculus is introduced in [7]. The details are beyond the scope of this paper. □

6.1 Background: Power

Let $\mathscr{C}$ be a category. The *power* [27, p.70] of a set $A : \mathbf{Set}$ and an object $X : \mathscr{C}$ consists of an object written $\Pi\,A\ .\ X : \mathscr{C}$ and a function $\pi : A \to \mathscr{C}(\Pi\,A\ .\ X, X)$. These two things have to satisfy the following *universal property*: for each object $B : \mathscr{C}$ and for each function $g : A \to \mathscr{C}(B, X)$, there exists an arrow $(\bigwedge\ a \in A\ .\ g(a)) : \mathscr{C}(B, \Pi\,A\ .\ X)$ (pronounce "split g") such that

$$f = (\bigwedge\ \hat{a} \in A\ .\ g(\hat{a})) \quad \Longleftrightarrow \quad (\lambda\,\check{a} \in A\ .\ \pi(\check{a}) \cdot f) = g \ , \tag{28}$$

for all $f : \mathscr{C}(B, \Pi\,A\ .\ X)$.

The power $\Pi\,A\ .\ X$ is an iterated product of the *object* X indexed by elements of the *set* A. The projection $\pi(a)$ is an arrow in $\mathscr{C}$ that selects the component whose index is a; the *arrow* $\bigwedge\ a \in A\ .\ g(a)$ creates an iterated product, whose components are determined by the *function* g. A note on notation: the mediating arrow $\bigwedge\ a{\in}A\ .\ g(a)$ is a binding construct as this allows us to leave the definition of g implicit. The notation also makes explicit that a ranges over a *set*. (The power $\Pi\,A\ .\ X$ is sometimes written X^A, a notation we do not use.) Furthermore, we use λ for function abstraction and $-\,(=)$ for function application in **Set**.

As an example, for a two-element set, say, $A := \{0, 1\}$, the power $\Pi\,A\ .\ X$ specialises to $X \times X$ with $\pi(0) = \mathit{outl}$, $\pi(1) = \mathit{outr}$ and $\bigwedge\ a \in A\ .\ g(a) = g(0) \vartriangle g(1)$.

In **Set**, the power $\Pi\,A\ .\ X$ is the set of all functions from A to X, that is, $\Pi\,A\ .\ X = A \to X$. The projection π is just reverse function application: $\pi(a) = \lambda\,g : A \to X\ .\ g(a)$; split is given by $\bigwedge\ a \in A\ .\ g(a) = \lambda\,b \in B\ .\ \lambda\,a \in A\ .\ g(a)(b)$, that is, it simply swaps the two arguments of the curried function g.

The universal property (28) has three immediate consequences that are used repeatedly in the forthcoming calculations. If we substitute the left-hand side into the right-hand side, we obtain the *computation law*

$$\pi(\check{a}) \cdot (\bigwedge\ \hat{a} \in A\ .\ g(\hat{a})) = g(\check{a}) \ , \tag{29}$$

for all $\check{a} \in A$. Instantiating f in (28) to the identity $id_{\prod A \,.\, X}$ and substituting the right- into the left-hand side, yields the *reflection law*

$$id = (\bigwedge\ a \in A \,.\, \pi(a)) \ . \tag{30}$$

Finally, the *fusion law* allows us to fuse a split with an arrow to form another split (the proof is left as an exercise to the reader):

$$(\bigwedge\ a \in A \,.\, g(a)) \cdot k = (\bigwedge\ a \in A \,.\, g(a) \cdot k) \ . \tag{31}$$

The fusion law states that $\bigwedge : (A \to \mathscr{C}(B, X)) \to \mathscr{C}(B, \prod A \,.\, X)$ is natural in B.

If the power $\prod A \,.\, X$ exists for every set $A : \mathbf{Set}$, then there is a unique way to turn $\prod - \,.\, X$ into a functor of type $\mathbf{Set} \to \mathscr{C}^{\mathsf{op}}$ so that $\pi : A \to \mathscr{C}(\prod A \,.\, X, X)$ is natural in A. We calculate

$$\begin{array}{ll} & \mathscr{C}(\prod h \,.\, X, X) \cdot \pi = \pi \cdot h \\ \iff & \{\ \text{definition of hom-functor } \mathscr{C}(-, X)\ \} \\ & (\lambda\, a \in A \,.\, \pi(a) \cdot (\prod h \,.\, X)) = \pi \cdot h \\ \iff & \{\ \text{universal property (28)}\ \} \\ & \prod h \,.\, X = (\bigwedge\ a \in A \,.\, \pi(h(a))) \ , \end{array}$$

which suggests that the arrow part of $\prod - \,.\, X$ is defined

$$\prod h \,.\, X = (\bigwedge\ a \in A \,.\, \pi(h(a))) \ . \tag{32}$$

In other words, we have an adjoint situation: $\prod - \,.\, X \dashv \mathscr{C}(-, X)$.

$$\mathscr{C}^{\mathsf{op}} \underset{\mathscr{C}(-,X)}{\overset{\prod - \,.\, X}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbf{Set}$$

Since the hom-functor $\mathscr{C}(-, X)$ is contravariant, $\prod - \,.\, X$ is contravariant, as well. Moreover, $\prod - \,.\, X$ is a left adjoint, targeting the opposite category $\mathscr{C}^{\mathsf{op}}$.

$$\mathscr{C}^{\mathsf{op}}(\prod A \,.\, X, B) \cong \mathbf{Set}(A, \mathscr{C}(B, X)) \tag{33}$$

The units of the adjunction are given by $\epsilon\, B = \bigwedge\ a \in \mathscr{C}(B, Y) \,.\, a$ and $\eta\, A = \lambda\, a \in A \,.\, \pi(a)$, that is, $\eta = \pi$. A contravariant adjoint functor such as $\prod - \,.\, X$ gives rise to two monads: $\mathscr{C}(\prod - \,.\, X, X)$ is a monad in $\mathbf{Set}$ and $\prod \mathscr{C}(-, X) \,.\, X$ is a comonad in $\mathscr{C}^{\mathsf{op}}$ and consequently a monad in $\mathscr{C}$. Both monads can be seen as continuation monads. Let us spell out the details for the second monad: its unit is the counit (!) of the adjunction, for the multiplication we calculate

$$
\begin{aligned}
& \mu A \\
= & \quad \{ \text{ definition of } \mu \text{ (8c) } \} \\
& ((\Pi - \,.\, X) \circ \eta \circ (\mathscr{C}(-, X)))\, A \\
= & \quad \{ \text{ definition of horizontal composition } \circ \ \} \\
& \Pi\, \eta\, (\mathscr{C}(A, X)) \,.\, X \\
= & \quad \{ \text{ definition of } \Pi\, h \,.\, X \text{ (32) } \} \\
& \bigwedge a \in \mathscr{C}(A, X) \,.\, \pi(\eta\, (\mathscr{C}(A, X))\, (a)) \\
= & \quad \{ \text{ definition of } \eta, \text{ see above } \} \\
& \bigwedge a \in \mathscr{C}(A, X) \,.\, \pi(\pi(a)) \ .
\end{aligned}
$$

To summarise, the continuation monad $\mathsf{K} = (\Pi - \,.\, X) \circ (\mathscr{C}(-, X))$ is defined

$$
\begin{aligned}
\mathsf{K}\, A &= \Pi\, \mathscr{C}(A, X) \,.\, X \ , \\
\mathsf{K}\, f &= \bigwedge k \,.\, \pi(k \cdot f) \ , \\
\eta &= \bigwedge k \,.\, k \ , \\
\mu &= \bigwedge k \,.\, \pi(\pi(k)) \ .
\end{aligned}
$$

This is the abstract rendering of the monad C from the introduction with $\mathsf{M}\, A = X$ a constant functor. The correspondence can be seen more clearly, if we specialise the ambient category $\mathscr{C}$ to **Set**. We obtain

$$
\begin{aligned}
\mathsf{K}\, f &= \bigwedge k \,.\, \pi(k \cdot f) = \lambda\, m \,.\, \lambda\, k \,.\, \pi(k \cdot f)\, m = \lambda\, m \,.\, \lambda\, k \,.\, m\, (k \cdot f) \\
\eta &= \bigwedge k \,.\, k = \lambda\, a \,.\, \lambda\, k \,.\, k\, a \ , \\
\mu &= \bigwedge k \,.\, \pi(\pi(k)) = \lambda\, m \,.\, \lambda\, k \,.\, \pi(\pi(k))\, m = \lambda\, m \,.\, \lambda\, k \,.\, m\, (\lambda\, a \,.\, a\, k) \ ,
\end{aligned}
$$

which is exactly the Haskell code given in the introduction—recall that join and bind are related by $join\ m = m \gg\!= id$. For the full story—M an arbitrary functor, not necessarily constant—we need to model the universal quantifier in the type of C, which is what we do next in Section 6.2.

If the adjunction $\Pi - \,.\, X \dashv \mathscr{C}(-, X)$ exists for every $X : \mathscr{C}$, then there is a unique way to turn $\Pi - \,.\, = : \mathbf{Set}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{C}$ into a bifunctor so that (33) is also natural in X [27, Th. IV.7.3, p102]. The arrow part of the bifunctor is defined

$$
\Pi\, h \,.\, p = (\bigwedge a \in A \,.\, p \cdot \pi(h(a))) \ , \tag{34}
$$

for all $h : A \to B$ and for all $p : \mathscr{C}(X, Y)$.

6.2 Background: End

Ends capture polymorphic functions as objects. Before we can define the notion formally, we first need to introduce the concept of a dinatural transformation [27, p.218].

Let $\mathsf{S}, \mathsf{T} : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{D}$ be two parallel functors. A *dinatural transformation* $\delta : \mathsf{S} \ddot{\to} \mathsf{T}$ is a collection of arrows: for each object $A : \mathscr{C}$ there is an arrow $\delta\, A : \mathsf{S}\,(A, A) \to \mathsf{T}\,(A, A)$, such that

$$\mathsf{T}\,(id, h) \cdot \delta\, \hat{A} \cdot \mathsf{S}\,(h, id) = \mathsf{T}\,(h, id) \cdot \delta\, \check{A} \cdot \mathsf{S}\,(id, h) \ , \tag{35}$$

for all $h : \mathscr{C}(\hat{A}, \check{A})$. A component of a dinatural transformation instantiates both arguments of the bifunctors to the same object, which explains the term dinaturality, a contraction of the more unwieldy *diagonal naturality*.

A natural transformation $\alpha : \mathsf{S} \dot{\to} \mathsf{T}$ can be turned into a dinatural transformation $\delta : \mathsf{S} \ddot{\to} \mathsf{T}$ by setting $\delta\, A = \alpha\,(A, A)$. There is an identity dinatural transformation, but, unfortunately, dinatural transformations do not compose in general. However, they are closed under composition with a natural transformation: $(\alpha \cdot \delta)\, A = \alpha\,(A, A) \cdot \delta\, A$ where $\delta : \mathsf{S} \ddot{\to} \mathsf{T}$ and $\alpha : \mathsf{T} \dot{\to} \mathsf{U}$, and $(\delta \cdot \alpha)\, A = \delta\, A \cdot \alpha\,(A, A)$ where $\alpha : \mathsf{S} \dot{\to} \mathsf{T}$ and $\delta : \mathsf{T} \ddot{\to} \mathsf{U}$.

A dinatural transformation $\omega : \Delta A \ddot{\to} \mathsf{T}$ from a constant functor, $\Delta A\, X = A$, is called a *wedge*. For wedges, the dinaturality condition (35) simplifies to

$$\mathsf{T}\,(id, h) \cdot \omega\, \hat{A} = \mathsf{T}\,(h, id) \cdot \omega\, \check{A} \ , \tag{36}$$

for all $h : \mathscr{C}(\hat{A}, \check{A})$.

Now, the *end* [27, p.222] of a functor $\mathsf{T} : \mathscr{C}^{\mathsf{op}} \times \mathscr{C} \to \mathscr{D}$ consists of an object written $\mathsf{End}\,\mathsf{T} : \mathscr{D}$ and a wedge $App : \Delta(\mathsf{End}\,\mathsf{T}) \ddot{\to} \mathsf{T}$. These two things have to satisfy the following universal property: for each object A and for each wedge $\omega : \Delta A \ddot{\to} \mathsf{T}$, there exists an arrow $\Lambda\,\omega : \mathscr{D}(A, \mathsf{End}\,\mathsf{T})$ such that

$$\omega = App \cdot \Delta g \quad \Longleftrightarrow \quad \Lambda\,\omega = g \ , \tag{37}$$

for all $g : \mathscr{D}(A, \mathsf{End}\,\mathsf{T})$. Note that on the left a dinatural transformation, App, is composed with a natural transformation, Δg defined $\Delta g\, X = g$.

The end $\mathsf{End}\,\mathsf{T}$ is also written $\forall X : \mathscr{C} \ .\ \mathsf{T}\,(X, X)$, which supports the intuition that an end is a polymorphic type. The wedge App models type application: the component $App\, A : \mathsf{End}\,\mathsf{T} \to \mathsf{T}\,(A, A)$ instantiates a given end to the object A. Accordingly, Λ is type abstraction—to emphasise this point we also write Λ using a binder: $\Lambda\, Z \ .\ \omega\, Z$ serves as alternative notation for $\Lambda\,\omega$.

The universal property (37) has the usual three consequences. If we substitute the right-hand side into the left-hand side, we obtain the *computation law*:

$$\omega = App \cdot \Delta(\Lambda\,\omega) \ . \tag{38}$$

Instantiating g in (37) to the identity $id_{\mathsf{End}\,\mathsf{T}}$ and substituting the left- into the right-hand side, yields the *reflection law*:

$$\Lambda\, App = id \ . \tag{39}$$

Finally, the *fusion law* allows us to fuse a 'type abstraction' with an arrow to form another 'type abstraction' (the proof is left as an exercise to the reader):

$$\Lambda\,\omega \cdot h = \Lambda\,(\omega \cdot \Delta h) \ . \tag{40}$$

If all the necessary ends exist, we can turn End into a higher-order functor of type $\mathscr{D}^{\mathscr{C}^{\mathsf{op}} \times \mathscr{C}} \to \mathscr{D}$. The object part maps a bifunctor to its end; the arrow part maps a *natural transformation* $\alpha : \mathsf{S} \dot{\to} \mathsf{T}$ to an arrow $\mathsf{End}\,\alpha : \mathscr{D}(\mathsf{End}\,\mathsf{S}, \mathsf{End}\,\mathsf{T})$. There is a unique way to define this arrow so that type application $App : \Delta(\mathsf{End}\,\mathsf{T}) \ddot{\to} \mathsf{T}$ is natural in T:

$$\alpha \cdot App = App \cdot \Delta(\mathsf{End}\,\alpha) \ . \tag{41}$$

We simply appeal to the universal property (37)

$$\alpha \cdot App = App \cdot \Delta(\mathsf{End}\,\alpha) \quad \Longleftrightarrow \quad \mathsf{End}\,\alpha = \Lambda\,(\alpha \cdot App) \ ,$$

which suggests that the arrow part of End is defined

$$\mathsf{End}\,\alpha = \Lambda\,(\alpha \cdot App) \ . \tag{42}$$

The proof that End indeed preserves identity and composition is again left as an exercise to the reader.

6.3 End Formula

Equipped with the new vocabulary we can now scrutinise the end formula (27), $(\mathsf{G}/\mathsf{J})\,(A : \mathscr{D}) = \forall Z : \mathscr{C} \ .\ \Pi\ \mathscr{D}\,(A, \mathsf{J}\,Z) \ .\ \mathsf{G}\,Z$, more closely. This definition is shorthand for $\mathsf{G}/\mathsf{J} = \mathsf{End} \circ \mathsf{T}$ where

$$\mathsf{T}\,A\,(Z^-, Z^+) = \Pi\ \mathscr{D}\,(A, \mathsf{J}\,Z^-) \ .\ \mathsf{G}\,Z^+ \ , \tag{43}$$

is a higher-order functor of type $\mathscr{D} \to \mathscr{D}^{\mathscr{C}^{\mathsf{op}} \times \mathscr{C}}$. (As an aside, Z^- and Z^+ are identifiers ranging over objects. The superscripts indicate variance: Z^- is an object of $\mathscr{C}^{\mathsf{op}}$ and Z^+ is an object of $\mathscr{C}$.) Clearly, G/J thus defined is a functor. It is useful to explicate its action on arrows.

$$\begin{array}{ll} & \forall Z : \mathscr{C} \ .\ \prod \mathscr{D}(f, \mathsf{J}\,Z) \ .\ \mathsf{G}\,Z \\ = & \quad \{ \text{ definition of } \mathsf{End}\ (42) \ \} \\ & \Lambda\,Z : \mathscr{C} \ .\ (\prod \mathscr{D}(f, \mathsf{J}\,Z) \ .\ \mathsf{G}\,Z) \cdot App\,Z \\ = & \quad \{ \text{ definition of } \textstyle\prod - \ .\ Y \quad (32) \ \} \\ & \Lambda\,Z : \mathscr{C} \ .\ (\bigwedge\ c \in \mathscr{D}(\check{A}, \mathsf{J}\,Z) \ .\ \pi(c \cdot f)) \cdot App\,Z \\ = & \quad \{ \text{ fusion } (31) \ \} \\ & \Lambda\,Z : \mathscr{C} \ .\ \bigwedge\ c \in \mathscr{D}(\check{A}, \mathsf{J}\,Z) \ .\ \pi(c \cdot f) \cdot App\,Z \end{array}$$

Let us record the definition.

$$(\mathsf{G}/\mathsf{J})\,f = \Lambda\,Z : \mathscr{C} \ .\ \bigwedge\ c \in \mathscr{D}(\check{A}, \mathsf{J}\,Z) \ .\ \pi(c \cdot f) \cdot App\,Z \tag{44}$$

The unit $run : \mathscr{E}^{\mathscr{C}}((\mathsf{G}/\mathsf{J}) \circ \mathsf{J}, \mathsf{G})$ of the right Kan extension is defined

$$run\,A = \pi(id_{\mathsf{J}\,A}) \cdot App\,A \ . \tag{45}$$

The end is instantiated to A and then the component whose index is the identity $id_{\mathsf{J}\,A}$ is selected. A number of proof obligations arise. We have to show that *run* is a natural transformation and that it satisfies the universal property (10) of right Kan extensions. For the calculations, the following property of *run*, a simple program optimisation, proves to be useful. Let $f : \mathscr{D}(A, \mathsf{J}\,B)$, then

$$run\ B \cdot (\mathsf{G}/\mathsf{J})\,f = \pi(f) \cdot App\ B \ . \tag{46}$$

We reason

$$\begin{array}{ll} & run\ B \cdot (\mathsf{G}/\mathsf{J})\,f \\ = & \quad \{\ \text{definition of } run\ (45)\ \} \\ & \pi(id_{\mathsf{J}\,B}) \cdot App\ B \cdot (\mathsf{G}/\mathsf{J})\,f \\ = & \quad \{\ \text{definition of } \mathsf{G}/\mathsf{J}\ (44)\ \} \\ & \pi(id_{\mathsf{J}\,B}) \cdot App\ B \cdot (\Lambda\, Z : \mathscr{C}\ .\ \bigwedge\ c \in \mathscr{D}(\mathsf{J}\,B, \mathsf{J}\,Z)\ .\ \pi(c \cdot f) \cdot App\ Z) \\ = & \quad \{\ \text{computation } (38)\ \} \\ & \pi(id_{\mathsf{J}\,B}) \cdot (\bigwedge\ c \in \mathscr{D}(\mathsf{J}\,B, \mathsf{J}\,B)\ .\ \pi(c \cdot f) \cdot App\ B) \\ = & \quad \{\ \text{computation } (29)\ \} \\ & \pi(f) \cdot App\ B \ . \end{array}$$

The naturality of *run* follows from the dinaturality of *App*.

$$\begin{array}{ll} & run\ \check{A} \cdot (\mathsf{G}/\mathsf{J})\,(\mathsf{J}\,h) \\ = & \quad \{\ \text{property of } run\ (46)\ \} \\ & \pi(\mathsf{J}\,h) \cdot App\ \check{A} \\ = & \quad \{\ App \text{ is dinatural, see below}\ \} \\ & \mathsf{G}\,h \cdot \pi(id_{\mathsf{J}\,\hat{A}}) \cdot App\ \hat{A} \\ = & \quad \{\ \text{definition of } run\ (45)\ \} \\ & \mathsf{G}\,h \cdot run\ \hat{A} \end{array}$$

To comprehend the second step let us instantiate the dinaturality condition (36) to $App : \Delta(\mathsf{End}\,(\mathsf{T}\,A)) \ddot{\to} \mathsf{T}\,A$. Let $h : \mathscr{C}(\hat{Z}, \check{Z})$, then

$$\begin{array}{ll} & \mathsf{T}\,A\,(id, h) \cdot App\ \hat{Z} = \mathsf{T}\,A\,(h, id) \cdot App\ \check{Z} \\ \Longleftrightarrow & \quad \{\ \text{definition of } \mathsf{T}\ (43)\ \} \\ & (\bigwedge\ a \in A\ .\ \mathsf{G}\,h \cdot \pi(a)) \cdot App\ \hat{Z} = (\bigwedge\ a \in A\ .\ \pi(\mathsf{J}\,h \cdot a)) \cdot App\ \check{Z} \\ \Longleftrightarrow & \quad \{\ \text{fusion } (31)\ \} \\ & (\bigwedge\ a \in A\ .\ \mathsf{G}\,h \cdot \pi(a) \cdot App\ \hat{Z}) = (\bigwedge\ a \in A\ .\ \pi(\mathsf{J}\,h \cdot a) \cdot App\ \check{Z}) \\ \Longleftarrow & \quad \{\ \text{left-compose with } \pi(id_{\mathsf{J}\,A}) \text{ and computation } (29)\ \} \\ & \mathsf{G}\,h \cdot \pi(id_{\mathsf{J}\,A}) \cdot App\ \hat{Z} = \pi(\mathsf{J}\,h) \cdot App\ \check{Z} \ . \end{array}$$

Next we show that *run* satisfies the universal property (10) of right Kan extensions. Along the way we derive the definition of shift. Let $\alpha : \mathscr{E}^{\mathscr{C}}(\mathsf{F} \circ \mathsf{J}, \mathsf{G})$ and $\beta : \mathscr{E}^{\mathscr{D}}(\mathsf{F}, \mathsf{G}/\mathsf{J})$, then

$$
\begin{array}{ll}
 & \alpha = \mathit{run} \cdot \beta \circ \mathsf{J} \\
\iff & \{ \text{ equality of natural transformations } \} \\
 & \forall A : \mathscr{C} \;.\; \alpha\, A = \mathit{run}\, A \cdot \beta\, (\mathsf{J}\, A) \\
\iff & \{ \text{ Yoneda Lemma: } \mathscr{E}(\mathsf{F}\,(\mathsf{J}\, A), \mathsf{G}\, A) \cong \mathscr{D}(-, \mathsf{J}\, A) \dot{\to} \mathscr{E}(\mathsf{F} -, \mathsf{G}\, A) \; \} \\
 & \forall A : \mathscr{C}, B : \mathscr{D} \;.\; \forall c : \mathscr{D}(B, \mathsf{J}\, A) \;.\; \alpha\, A \cdot \mathsf{F}\, c = \mathit{run}\, A \cdot \beta\, (\mathsf{J}\, A) \cdot \mathsf{F}\, c \\
\iff & \{ \; \beta \text{ is natural } \} \\
 & \forall A : \mathscr{C}, B : \mathscr{D} \;.\; \forall c : \mathscr{D}(B, \mathsf{J}\, A) \;.\; \alpha\, A \cdot \mathsf{F}\, c = \mathit{run}\, A \cdot (\mathsf{G}/\mathsf{J})\, c \cdot \beta\, B \\
\iff & \{ \text{ property of } \mathit{run} \text{ (46) } \} \\
 & \forall A : \mathscr{C}, B : \mathscr{D} \;.\; \forall c : \mathscr{D}(B, \mathsf{J}\, A) \;.\; \alpha\, A \cdot \mathsf{F}\, c = \pi(c) \cdot \mathit{App}\, A \cdot \beta\, B \\
\iff & \{ \text{ universal property of powers (28) } \} \\
 & \forall A : \mathscr{C}, B : \mathscr{D} \;.\; (\bigwedge\, c \in \mathscr{D}(B, \mathsf{J}\, A) \;.\; \alpha\, A \cdot \mathsf{F}\, c) = \mathit{App}\, A \cdot \beta\, B \\
\iff & \{ \text{ universal property of ends (37) } \} \\
 & \forall B : \mathscr{D} \;.\; (\bigwedge A : \mathscr{C} \;.\; \bigwedge\, c \in \mathscr{D}(B, \mathsf{J}\, A) \;.\; \alpha\, A \cdot \mathsf{F}\, c) = \beta\, B \\
\iff & \{ \text{ define } [\alpha]\, B = \bigwedge A : \mathscr{C} \;.\; \bigwedge\, c \in \mathscr{D}(B, \mathsf{J}\, A) \;.\; \alpha\, A \cdot \mathsf{F}\, c \; \} \\
 & \forall B : \mathscr{D} \;.\; [\alpha]\, B = \beta\, B \\
\iff & \{ \text{ equality of natural transformations } \} \\
 & [\alpha] = \beta \;\; .
\end{array}
$$

Each of the steps is fairly compelling, except perhaps the second one, which rests on the Yoneda Lemma [27, p.61]. Its purpose is to introduce the functor application $\mathsf{F}\, c$ so that the naturality of β can be utilised. Thus, shift is defined

$$[\alpha]\, A = \bigwedge Z : \mathscr{C} \;.\; \bigwedge\, c \in \mathscr{D}(A, \mathsf{J}\, Z) \;.\; \alpha\, Z \cdot \mathsf{F}\, c \;\; . \tag{47}$$

Two remarks are in order. First, the body of the type abstraction, that is $\bigwedge\, c \in \mathscr{D}(A, \mathsf{J}\, Z) \;.\; \alpha\, Z \cdot \mathsf{F}\, c$ is a *dinatural* transformation because it equals $\mathit{App}\, Z \cdot \beta\, A = (\mathit{App} \cdot \Delta(\beta\, A))\, Z$—see derivation above—which is dinatural in Z. Second, $[\alpha]$ itself is a *natural* transformation because β is one by assumption.

Let us now turn our attention to the implementation of the codensity monad of a functor J. Combining (20a) with the end formula (27) gives

$$\mathsf{C}\, A = \forall Z : \mathscr{C} \;.\; \Pi\; \mathscr{D}\,(A, \mathsf{J}\, Z) \;.\; \mathsf{J}\, Z \;\; .$$

The instance of the end formula on the right is commonly regarded as *the* codensity monad. This view is partially justified since the end formula provides a general implementation of right Kan extensions, subject to the existence of the necessary powers and ends. It confuses, however, an implementation with an abstract concept. (This confusion is not uncommon in computer science.) The codensity monad is also regarded as the 'real' continuation monad. To see the

relation to continuation-passing style, let us unroll the definitions of return and join. For $\eta = [id]$ (20b), we obtain

$$\begin{aligned}
& [id]\, A \\
= \quad & \{ \text{ definition of } [-] \ (47) \ \} \\
& \Lambda\, Z : \mathscr{C} \ .\ \bigwedge k \in \mathscr{D}(A, \mathsf{J}\, Z) \ .\ id\ Z \cdot \mathsf{Id}\, k \\
= \quad & \{ \text{ identity } \} \\
& \Lambda\, Z : \mathscr{C} \ .\ \bigwedge k \in \mathscr{D}(A, \mathsf{J}\, Z) \ .\ k \ \ ,
\end{aligned}$$

and for $\mu = [run \cdot \mathsf{M} \circ run]$ (20c) we calculate

$$\begin{aligned}
& [run \cdot \mathsf{M} \circ run]\, A \\
= \quad & \{ \text{ definition of } [-] \ (47) \ \} \\
& \Lambda\, Z : \mathscr{C} \ .\ \bigwedge k \in \mathscr{D}(A, \mathsf{J}\, Z) \ .\ run\ Z \cdot \mathsf{M}\,(run\ Z) \cdot \mathsf{M}\,(\mathsf{M}\, k) \\
= \quad & \{ \ \mathsf{M} \text{ functor } \} \\
& \Lambda\, Z : \mathscr{C} \ .\ \bigwedge k \in \mathscr{D}(A, \mathsf{J}\, Z) \ .\ run\ Z \cdot \mathsf{M}\,(run\ Z \cdot \mathsf{M}\, k) \\
= \quad & \{ \text{ property of } run \ (46) \ \} \\
& \Lambda\, Z : \mathscr{C} \ .\ \bigwedge k \in \mathscr{D}(A, \mathsf{J}\, Z) \ .\ run\ Z \cdot \mathsf{M}\,(\pi(k) \cdot App\ Z) \\
= \quad & \{ \text{ property of } run \ (46) \ \} \\
& \Lambda\, Z : \mathscr{C} \ .\ \bigwedge k \in \mathscr{D}(A, \mathsf{J}\, Z) \ .\ \pi(\pi(k) \cdot App\ Z) \cdot App\ Z \ \ .
\end{aligned}$$

To summarise, the codensity monad implemented in terms of powers and ends is given by

$$\begin{aligned}
\mathsf{C}\, A &= \forall Z \ .\ \Pi\ \mathscr{D}\,(A, \mathsf{J}\, Z) \ .\ \mathsf{J}\, Z \ \ , \\
\mathsf{C} f &= \Lambda\, Z \ .\ \bigwedge k \ .\ \pi(k \cdot f) \cdot App\ Z \ \ , \\
\eta &= \Lambda\, Z \ .\ \bigwedge k \ .\ k \ \ , \\
\mu &= \Lambda\, Z \ .\ \bigwedge k \ .\ \pi(\pi(k) \cdot App\ Z) \cdot App\ Z \ \ ,
\end{aligned}$$

which is similar to the continuation monad K of Section 6.1, except for occurrences of type abstraction and type application. This is the abstract rendering of the Haskell code for C from the introduction—note that in Haskell type abstraction and type application are implicit.

7 Examples

Let $\mathsf{L} \dashv \mathsf{R}$ be an adjunction.

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

We have encountered two implementations of the codensity monad of R: the standard implementation $\mathsf{R} \circ \mathsf{L}$ and the implementation induced by the end formula (27). Since right Kan extensions are unique up to isomorphism (see Section 3), we have

$$\mathsf{R} \circ \mathsf{L} = \mathsf{R}/_1\mathsf{R} \cong \mathsf{R}/_2\mathsf{R} = \lambda\, A : \mathscr{D} \,.\, \forall Z : \mathscr{C} \,.\, \Pi\, \mathscr{D}\,(A, \mathsf{R}\, Z) \,.\, \mathsf{R}\, Z \quad . \tag{48}$$

The isomorphisms are $[run_1]_2$ and $[run_2]_1$ (again see Section 3).

In the following sections we look at a few instances of this isomorphism. The list is by no means exhaustive, but it is indicative of the wide range of applications—the reader is invited to explore further adjunctions.

7.1 Identity Monad

The simplest example of an adjunction is $\mathsf{Id} \dashv \mathsf{Id}$, which induces the identity monad.

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

The units of the adjunction are identities: $\epsilon_1 = id$ and $\eta_1 = id$. Furthermore, run and shift are defined $run_1 = id$ and $[\alpha]_1 = \alpha$.

Instantiating (48) to $\mathsf{Id} \dashv \mathsf{Id}$ yields

$$\mathsf{Id} \cong \lambda\, A : \mathscr{D} \,.\, \forall Z : \mathscr{C} \,.\, \Pi\, \mathscr{D}\,(A, Z) \,.\, Z \quad , \tag{49}$$

which generalises one of the main examples in Wadler's famous paper "Theorems for free!" [34]. Wadler shows $A \cong \forall Z : \mathscr{C} \,.\, \Pi\, \mathscr{D}\,(A, Z) \,.\, Z$. Equation (49) tells us that this isomorphism is also natural in A. The isomorphisms $[run_1]_2$ and $[run_2]_1$ specialise to

$$[run_1]_2 = [id]_2 = \eta_2 \quad ,$$
$$[run_2]_1 = run_2 \quad .$$

One direction is given by the unit of the 'continuation monad'; for the other direction we simply run the continuation monad.

7.2 State Monad

The Haskell programmer's favourite adjunction is currying: $- \times X \dashv (-)^X$.

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

In **Set**, a function of two arguments can be treated as a function of the first argument whose values are functions of the second argument. In general, we are seeking the right adjoint of pairing with a fixed object $X : \mathscr{C}$.

$$\mathscr{C}(A \times X, B) \cong \mathscr{C}(A, B^X) \ .$$

The object B^X is called the *exponential* of X and B. That this adjunction exists is one of the requirements for *cartesian closure* [25]. In **Set**, B^X is the set of total functions from X to B.

The curry adjunction induces the state monad, where $X : \mathscr{C}$ serves as the state space. This instance of (48) reads

$$(- \times X)^X \cong \lambda\, A : \mathscr{C} \ . \ \forall Z : \mathscr{C} \ . \ \Pi\, \mathscr{C}\,(A, Z^X) \ . \ Z^X \tag{50}$$

On the left we have the standard implementation of the state monad using state transformers. The end formula yields an implementation in continuation-passing style. The continuation of type $\mathscr{C}(A, Z^X) \cong \mathscr{C}(A \times X, Z)$ takes an element of the return type A and an element of the state type X, the final state. The initial state is passed to the exponential in the body of the power.

The Haskell rendering of the two implementations is fairly straightforward. Here is the standard implementation

$$\mathbf{newtype}\, \mathsf{State}_1\ a = \mathit{In}\, \{\, \mathit{out} : X \to (a, X) \}\ ,$$

and here is the CPS-based one

$$\mathbf{newtype}\, \mathsf{State}_2\ a = \mathit{CPS}\, \{\, \mathit{call} : \forall\, z\ .\ (a \to (X \to z)) \to (X \to z) \}\ .$$

7.3 Free Monad of a Functor

One of the most important adjunctions for the algebra of programming is $\mathsf{Free} \dashv \mathsf{U}$, which induces the so-called free monad of a functor. This adjunction makes a particularly interesting example as it involves two different categories. Here are the gory details:

Let $\mathsf{F} : \mathscr{C} \to \mathscr{C}$ be an endofunctor. An F-*algebra* is a pair $\langle A,\ a \rangle$ consisting of an object $A : \mathscr{C}$ (the carrier of the algebra) and an arrow $a : \mathscr{C}(\mathsf{F}\, A, A)$ (the action of the algebra). An F-*algebra homomorphism* between algebras $\langle A,\ a \rangle$ and $\langle B,\ b \rangle$ is an arrow $h : \mathscr{C}(A, B)$ such that $h \cdot a = b \cdot \mathsf{F}\, h$. Identity is an F-algebra homomorphism and homomorphisms compose. Thus, the data defines a category, called F-$\mathbf{Alg}(\mathscr{C})$.

The category F-$\mathbf{Alg}(\mathscr{C})$ has more structure than $\mathscr{C}$. The forgetful or underlying functor $\mathsf{U} : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \to \mathscr{C}$ forgets about the additional structure: $\mathsf{U}\, \langle A,\ a \rangle = A$ and $\mathsf{U}\, h = h$. While the definition of the forgetful functor is deceptively simple, it gives rise to an interesting concept via an adjunction.

$$\mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \ \underset{\mathsf{U}}{\overset{\mathsf{Free}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\ \mathscr{C}$$

The left adjoint Free maps an object A to the *free* F-*algebra* over A, written $\langle \mathsf{F}^*\, A, \mathit{com}\rangle$. In **Set**, the elements of $\mathsf{F}^*\, A$ are terms built from constructors determined by F and variables drawn from A. Think of the functor F as a grammar describing the syntax of a language. The action $\mathit{com} : \mathscr{C}(\mathsf{F}\,(\mathsf{F}^*\, A), \mathsf{F}^*\, A)$ constructs a *com*posite term from an F-structure of subterms. There is also an operation $\mathit{var} : \mathscr{C}(A, \mathsf{F}^*\, A)$ for embedding a *var*iable into a term. This operation is a further example of a universal arrow: for each F-algebra B and for each arrow $g : \mathscr{C}(A, \mathsf{U}\, B)$ there exists an F-algebra homomorphism $\mathit{eval}\, g : \mathsf{F}\text{-}\mathbf{Alg}(\mathsf{Free}\, A, B)$ (pronounce "*eval*uate with g") such that

$$f = \mathit{eval}\, g \iff \mathsf{U} f \cdot \mathit{var} = g \ , \tag{51}$$

for all $f : \mathsf{F}\text{-}\mathbf{Alg}(\mathsf{Free}\, A, B)$. In words, the meaning of a term is uniquely determined by the meaning of the variables. The fact that $\mathit{eval}\, g$ is a homomorphism entails that the meaning function is compositional: the meaning of a composite term is defined in terms of the meanings of its constituent parts.

The adjunction $\mathsf{Free} \dashv \mathsf{U}$ induces the free monad F^* of the functor F. The isomorphism (48) gives two implementations of the free monad.

$$\mathsf{F}^* \cong \lambda\, A : \mathscr{C} \,.\, \forall Z : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \,.\, \Pi\, \mathscr{C}\,(A, \mathsf{U}\, Z) \,.\, \mathsf{U}\, Z \tag{52}$$

The standard implementation represents terms as finite trees: the free algebra $\mathsf{F}^*\, A$ is isomorphic to $\mu \mathsf{F}_A$ where $\mathsf{F}_A\, X = A + \mathsf{F}\, X$ [1]. The implementation based on Kan extensions can be seen as the *Church representation* [26,5] of terms. Note that the variable Z ranges over F-algebras. The continuation of type $\mathscr{C}(A, \mathsf{U}\, Z)$ specifies the meaning of variables. Given such a meaning function a term can be evaluated to an element of type $\mathsf{U}\, Z$. (It is debatable whether the term 'continuation' makes sense here—U/U is certainly a *generalised* continuation type.)

It is instructive to consider how the definitions translate into Haskell. The implementation using trees is straightforward: the constructors *var* and *com* are turned into constructors of a datatype (we are building on the isomorphism $\mathsf{F}^*\, A \cong \mu \mathsf{F}_A$ here).

$$\mathbf{data}\, \mathsf{Term}_1\, a = \mathit{Var}\, a \mid \mathit{Com}\, (\mathsf{F}\, (\mathsf{Term}_1\, a))$$

The Church representation is more interesting as we have to deal with the question of how to model the variable Z, which ranges over F-algebras. One way to achieve this is to constrain Z by a class context.

$$\begin{aligned}&\mathbf{class}\, \mathit{Algebra}\, a\, \mathbf{where}\\ &\quad \mathit{algebra} : \mathsf{F}\, a \to a\end{aligned}$$

The Church representation then reads

$$\mathbf{newtype}\, \mathsf{Term}_2\, a = \mathit{Abstr}\, \{\, \mathit{apply} : \forall\, z \,.\, (\mathit{Algebra}\, z) \Rightarrow (a \to z) \to z \,\}\ .$$

The two implementations represent two extremes. For terms as trees, constructing a term is easy, whereas evaluating a term is hard work. For the Church

representation, it is the other way round. Evaluating a term is a breeze as, in a sense, a term *is* an evaluator. Constructing a term is slightly harder, but not that much. The reader is invited to spell out the details.

Initial and free algebras are closely related (see above). The so-called extended initial algebra semantics [11] is a simple consequence of (52).

$$
\begin{aligned}
& \mu\mathsf{F} \\
\cong \quad & \{ \text{ left adjoint preserve colimits: } \mu\mathsf{F} = \mathsf{U}\,0 \cong \mathsf{U}\,(\mathsf{Free}\,0) = \mathsf{F}^*\,0 \,\} \\
& \mathsf{F}^*\,0 \\
\cong \quad & \{ \text{ (52) } \} \\
& \forall Z : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \,.\, \Pi\, \mathscr{C}\,(0, \mathsf{U}\,Z) \,.\, \mathsf{U}\,Z \\
\cong \quad & \{ \text{ 0 is initial } \} \\
& \forall Z : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \,.\, \Pi\, 1 \,.\, \mathsf{U}\,Z \\
\cong \quad & \{ \Pi\, 1 \,.\, X \cong X \,\} \\
& \forall Z : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \,.\, \mathsf{U}\,Z \\
\cong \quad & \{ \text{ relation between ends and limits [27, Prop. IX.5.3] } \} \\
& \mathsf{Lim}\,\mathsf{U}
\end{aligned}
$$

The calculation shows that a *colimit*, the initial algebra $\mu\mathsf{F}$, is isomorphic to a *limit*, the limit of the forgetful functor. (This is familiar from lattice theory: the least element of a lattice is both the supremum of the empty set and the infimum of the entire lattice.)

Remark 9. Arrows of type $A \to \mathsf{Lim}\,\mathsf{U}$ and natural transformations of type $\Delta A \dot{\to} \mathsf{U}$ are in one-to-one correspondence. (In other words, we have an adjunction $\Delta \dashv \mathsf{Lim}$.) Using the concept of a *strong* dinatural transformation, the naturality property can be captured solely in terms of the underlying category $\mathscr{C}$ [11]. Whether a similar construction is also possible for ends is left for future work. □

7.4 List Monad

The Haskell programmer's favourite data structure, the type of parametric lists, arises out of an adjunction between **Mon**, the category of monoids and monoid homomorphisms, and **Set**, the category of sets and total functions.

$$\mathbf{Mon} \;\underset{\mathsf{U}}{\overset{\mathsf{Free}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}}\; \mathbf{Set}$$

Now U is the underlying functor that forgets about the monoidal structure, mapping a monoid to its carrier set. Its left adjoint Free maps a set A to the *free monoid on* A, whose elements are finite sequences of elements of A.

The adjunction $\mathsf{Free} \dashv \mathsf{U}$ induces the list monad List. For this instance, the isomorphism (48) can be simplified to

$$\mathsf{List} \cong \lambda\, A : \mathbf{Set} \,.\, \forall Z : \mathbf{Mon} \,.\, (A \to \mathsf{U}\,Z) \to \mathsf{U}\,Z \quad . \tag{53}$$

The variable Z now ranges over monoids. An element of the end can be seen as an evaluator: given a function of type $A \to \mathsf{U}\, Z$, which determines the meaning of singleton lists, the list represented can be homomorphically evaluated to an element of type $\mathsf{U}\, Z$.

Turning to Haskell, the standard implementation corresponds to the familiar datatype of lists.

$$\mathbf{data}\, \mathsf{List}_1\ a = \mathit{Nil} \mid \mathit{Cons}\, (a, \mathsf{List}_1\ a)$$

For the Church representation we take a similar approach as in the previous section: we introduce a class *Monoid* and constrain the universally quantified variable by a class context.

$$\begin{aligned}
&\mathbf{class}\, \mathit{Monoid}\ a\, \mathbf{where}\\
&\quad \epsilon \quad : a\\
&\quad (\bullet) : a \to a \to a\\
&\mathbf{newtype}\, \mathsf{List}_2\ a = \mathit{Abstr}\, \{\, \mathit{apply} : \forall\, z\ .\ (\mathit{Monoid}\ z) \Rightarrow (a \to z) \to z \,\}
\end{aligned}$$

Of course, in Haskell there is no guarantee that an instance of *Monoid* is actually a monoid—this is a proof obligation for the programmer. We can turn the free constructions into monoids as follows:

$$\begin{aligned}
&\mathbf{instance}\, \mathit{Monoid}\, (\mathsf{List}_1\ a)\, \mathbf{where}\\
&\quad \epsilon \qquad\qquad = \mathit{Nil}\\
&\quad \mathit{Nil} \bullet y \qquad = y\\
&\quad \mathit{Cons}\, (a, x) \bullet y = \mathit{Cons}\, (a, x \bullet y)\\
&\mathbf{instance}\, \mathit{Monoid}\, (\mathsf{List}_2\ a)\, \mathbf{where}\\
&\quad \epsilon \quad = \mathit{Abstr}\, (\lambda k \to \epsilon)\\
&\quad x \bullet y = \mathit{Abstr}\, (\lambda k \to \mathit{apply}\ x\ k \bullet \mathit{apply}\ y\ k)\ .
\end{aligned}$$

The second instance is closely related to the Haskell code from the introduction: the implementation of backtracking using a success and a failure continuation simply specialises z to the monoid of endofunctions.

$$\begin{aligned}
&\mathbf{instance}\, \mathit{Monoid}\, (a \to a)\, \mathbf{where}\\
&\quad \epsilon \quad = \mathit{id}\\
&\quad x \bullet y = x \cdot y
\end{aligned}$$

We can instantiate z to this monoid without loss of generality as every monoid is isomorphic to a monoid of endofunctions, the so-called Cayley representation, named after Arthur Cayley:

$$(A, \epsilon, \bullet) \cong (\{\, (a \bullet -) : A \to A \mid a \in A \,\}, \mathit{id}, \cdot)\ .$$

This isomorphism is also the gist of Hughes' efficient representation of lists [19].

To summarise, the CPS variant of the list monad combines Kan extensions and Cayley representations—Dan explains the success and Art the failure continuation.

8 Related Work

As mentioned in the introduction, all of the core results appear either explicitly or implicitly in Mac Lane's textbook [27]. Specifically, Section X.7 of the textbook introduces the notion of absolute Kan extensions—that $\mathsf{L} \dashv \mathsf{R}$ induces $- \circ \mathsf{R} \dashv - \circ \mathsf{L}$ is implicit in the proof of Theorem X.7.2. Overall, our paper solves Exercise X.7.3, which asks the reader to show that J/J is a monad and that $\mathsf{R} \circ \mathsf{L}$ is isomorphic to R/R.

Kan Extension. Kan extensions are named after Daniel Kan, who constructed **Set**-valued extensions using limits and colimits in 1960. Kan extensions have found a few applications in computer science: right Kan extensions have been used to give a semantics to generalised folds for nested datatypes [4]; left Kan extensions have been used to provide an initial algebra semantics for certain "generalised algebraic datatypes" [22].

Codensity Monad. The origins of the 'trick', wrapping a CPS transformation around a monad, seem to be unknown. The trick captured as a monad transformer was introduced by the author in 1996 [12]. Much later, Jaskelioff noted that the transformer corresponds to a construction in category theory, the codensity monad [21]. None of the papers that utilise the trick [13,8,33,20,28], however, employ the isomorphism $\mathsf{R} \circ \mathsf{L} \cong \mathsf{R}/\mathsf{R}$—all of them work with M/M instead. In more detail:

Building on the work of Hughes [18], the author showed how to derive backtracking monad transformers that support computational effects such as the Prolog cut and an operation for delimiting the effect of a cut [13]. Wand et al [35] later identified a problem with our derivations, which built on fold-unfold transformations [6]. Roughly speaking, the culprit is the lack of a sound induction principle for local universal quantification. Wand et al proposed an alternative approach based on logical relations. Their proof, however, uses a different CPS monad with a fixed type O of observations, $\mathsf{B}\, a = (a \to (O \to O)) \to (O \to O)$, which is somewhat unsatisfactory.

Claessen [8] applied the trick to speed up his parallel parsing combinators. Voigtländer [33] showed that the trick gives an asymptotic improvement for free algebras and the operation of substitution. He sketched a proof of correctness and conjectured that a formal proof might require sophisticated techniques involving free theorems. This gap was later filled by Hutton et al [20] who proved correctness by framing it as an instance of the so-called worker/wrapper transformation. Their proof, however, is an indirect one as it only establishes the equivalence of two functions from a common source into the two monads. Finally, a more advanced application involving indexed monads was recently given by McBride [28].

Kan extensions and the codensity monad are also popular topics for blog posts. Piponi (`blog.sigfpe.com`) expands on the codensity monad as "the mother of all monads", a catchy phrase due to Peter Hancock. Kmett (`comonad.com/reader`) has a series of posts on both topics, including a wealth of Haskell code.

String Diagram. String diagrams were introduced by Penrose [31] as an alternative notation for "abstract tensor systems". These diagrams are widely used for monoidal categories—Selinger [32] surveys graphical languages for different types of monoidal structures. (A monoidal category is a special case of a 2-category, namely, one that has only a single object. Consequently, there is no need to label or colour regions.)

9 Conclusion

Monads can be seen as abstract datatypes for computational effects. (This view is admittedly a bit limited—consider the free monad of a functor, would you want to regard substitution as a computational effect?) The take-home message of this paper is that the right Kan extension of the functor R along R is a drop-in replacement for the monad induced by the adjunction $\mathsf{L} \dashv \mathsf{R}$. The paper stresses the importance of adjunctions, a point already emphasised in a previous paper by the author [14]. The bad news is that the construction, the implementation of the codensity monad using ends and powers, does not lend itself easily to a library implementation, at least not in today's programming languages. A generic implementation would require support for abstraction over categories.

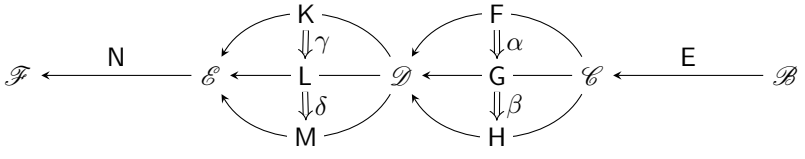
For quite a while I have been experimenting with two-dimensional notation for calculational proofs. I first tried traditional notation, the Poincaré dual of string diagrams, but I never used it in anger as I found the diagrams difficult to compose and to manipulate. The reason is that natural transformations, the main focus of interest, are represented by regions, which are often difficult to lay out in an aesthetically pleasing way. By contrast, string diagrams are fairly easy to draw. Furthermore, the least interesting piece of information—which are the categories involved?— can be easily suppressed. I hope to see string diagrams more widely used for program calculation in the future.

Everything we have said nicely dualises: right Kan extensions dualise to left Kan extensions and the codensity monad dualises to the density comonad. A left Kan extension can be seen as a generalised existential type and the density comonad corresponds to an abstract datatype or a simple object type—the type of parametric streams is one of the prime examples. Quite clearly, the dual story is interesting in its own right but this is a story to be told elsewhere.

Acknowledgements. I owe a particular debt of gratitude to Steven Vickers for pointing me to wire diagrams, which inspired me to explore two-dimensional notation. A big thank you is furthermore due to Daniel James and Nicolas Wu for artistically typesetting the diagrams. Nicolas also suggested various presentational improvements. Thanks are furthermore due to Roland Backhouse, Jeremy Gibbons and the other (anonymous) referees of MPC 2012 for finding several typographical errors, glitches of language, and for forcing me to be precise about the contributions of the paper. In particular, I would like to thank Roland for proposing the notation G/J instead of the more traditional $\mathsf{Ran}_\mathsf{J}\ \mathsf{G}$ and for pointing me to regular algebras. I have added Remarks 1, 3, 5, 6 and 7 in response to his review.

A Composition of Functors and Natural Transformations

This appendix contains supplementary material. It is intended primarily as a reference, so that the reader can re-familiarise themselves with the category theory that is utilised in this paper. Specifically, we introduce composition of functors and natural transformations. We shall use the following entities to frame the discussion ($\mathsf{F}, \mathsf{G} : \mathscr{C} \to \mathscr{D}$ are parallel functors, $\alpha : \mathsf{F} \to \mathsf{G}$ is a natural transformation between them etc).



Here we use traditional two-dimensional notation. The reader is invited to turn the diagram into its Poincaré dual, a string diagram.

Functors can be composed, written $\mathsf{K}\circ\mathsf{F}$. Rather intriguingly, the operation $\mathsf{K}\circ-$, post-composing a functor K, is itself functorial: the higher-order functor $\mathsf{K}\circ- : \mathscr{D}^{\mathscr{C}} \to \mathscr{E}^{\mathscr{C}}$ maps the functor F to the functor $\mathsf{K}\circ\mathsf{F}$ and the natural transformation α to the natural transformation $\mathsf{K}\circ\alpha$ defined $(\mathsf{K}\circ\alpha)\,A = \mathsf{K}\,(\alpha\,A)$. Post-composition dualises to pre-composition: the higher-order functor $-\circ\mathsf{E} : \mathscr{D}^{\mathscr{C}} \to \mathscr{D}^{\mathscr{B}}$ maps the functor F to the functor $\mathsf{F}\circ\mathsf{E}$ and the natural transformation α to the natural transformation $\alpha\circ\mathsf{E}$ defined $(\alpha\circ\mathsf{E})\,A = \alpha\,(\mathsf{E}\,A)$. (The reader should convince themselves that $\mathsf{K}\circ\alpha : \mathsf{K}\circ\mathsf{F} \dot{\to} \mathsf{K}\circ\mathsf{G}$ and $\alpha\circ\mathsf{E} : \mathsf{F}\circ\mathsf{E} \dot{\to} \mathsf{G}\circ\mathsf{E}$ are again natural transformations.) Here are the functor laws spelled out.

$$\mathsf{K}\circ id_{\mathsf{F}} = id_{\mathsf{K}\circ\mathsf{F}} \qquad (54a) \qquad\qquad id_{\mathsf{F}}\circ\mathsf{E} = id_{\mathsf{F}\circ\mathsf{E}} \qquad (54c)$$

$$\mathsf{K}\circ(\beta\cdot\alpha) = (\mathsf{K}\circ\beta)\cdot(\mathsf{K}\circ\alpha) \qquad (54b) \qquad\qquad (\beta\cdot\alpha)\circ\mathsf{E} = (\beta\circ\mathsf{E})\cdot(\alpha\circ\mathsf{E}) \qquad (54d)$$

Altogether, we have three different forms of composition: $\mathsf{K}\circ\mathsf{F}$, $\gamma\circ\mathsf{F}$ and $\mathsf{K}\circ\alpha$. They are 'pseudo-associative' and have the functor Id as their neutral element.

$$\gamma\circ(\mathsf{F}\circ\mathsf{E}) = (\gamma\circ\mathsf{F})\circ\mathsf{E} \qquad (55a)$$

$$\mathsf{K}\circ(\beta\circ\mathsf{E}) = (\mathsf{K}\circ\beta)\circ\mathsf{E} \qquad (55b)$$

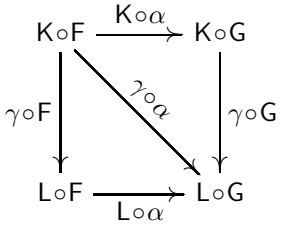
$$\mathsf{N}\circ(\mathsf{M}\circ\alpha) = (\mathsf{N}\circ\mathsf{M})\circ\alpha \qquad (55c)$$

$$\mathsf{Id}\circ\alpha = \alpha \qquad (55d)$$

$$\alpha\circ\mathsf{Id} = \alpha \qquad (55e)$$

This means that we can freely drop parentheses when composing compositions.

Given two natural transformations $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ and $\gamma : \mathsf{K} \dot{\to} \mathsf{L}$, there are two ways to turn a $\mathsf{K}\circ\mathsf{F}$ into an $\mathsf{L}\circ\mathsf{G}$ structure.



The diagram commutes since γ is natural:

$$\begin{aligned}
& ((\gamma \circ \mathsf{G}) \cdot (\mathsf{K} \circ \alpha))\, X \\
= & \quad \{ \text{ definition of compositions } \} \\
& \gamma\,(\mathsf{G}\, X) \cdot \mathsf{K}\,(\alpha\, X) \\
= & \quad \{ \ \gamma \text{ is natural: } \mathsf{L}\, h \cdot \gamma\, A = \gamma\, B \cdot \mathsf{K}\, h \ \} \\
& \mathsf{L}\,(\alpha\, X) \cdot \gamma\,(\mathsf{F}\, X) \\
= & \quad \{ \text{ definition of compositions } \} \\
& ((\mathsf{L} \circ \alpha) \cdot (\gamma \circ \mathsf{F}))\, X \quad .
\end{aligned}$$

The diagonal is called the *horizontal composition* of natural transformations, denoted $\gamma \circ \alpha$.

$$(\gamma \circ \mathsf{G}) \cdot (\mathsf{K} \circ \alpha) = \gamma \circ \alpha = (\mathsf{L} \circ \alpha) \cdot (\gamma \circ \mathsf{F}) \tag{56}$$

The definition witnesses the fact that functor composition $\mathscr{E}^{\mathscr{D}} \times \mathscr{D}^{\mathscr{C}} \to \mathscr{E}^{\mathscr{C}}$ is a bi-functor: (56) defines its action on arrows.

References

1. Awodey, S.: Category Theory, 2nd edn. Oxford University Press (2010)
2. Backhouse, R., Bijsterveld, M., van Geldrop, R., van der Woude, J.: Category theory as coherently constructive lattice theory (1994), http://www.cs.nott.ac.uk/~rcb/MPC/CatTheory.ps.gz
3. Bird, R., de Moor, O.: Algebra of Programming. Prentice Hall Europe, London (1997)
4. Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects of Computing 11(2), 200–222 (1999)
5. Böhm, C., Berarducci, A.: Automatic synthesis of typed λ-programs on term algebras. Theoretical Computer Science 39(2-3), 135–154 (1985)
6. Burstall, R., Darlington, J.: A tranformation system for developing recursive programs. Journal of the ACM 24(1), 44–67 (1977)
7. Cáccamo, M., Winskel, G.: A Higher-Order Calculus for Categories. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 136–153. Springer, Heidelberg (2001), http://dx.doi.org/10.1007/3-540-44755-5_11
8. Claessen, K.: Functional pearl: Parallel parsing processes. Journal of Functional Programming 14(6), 741–757 (2004), http://dx.doi.org/10.1017/S0956796804005192
9. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, pp. 151–160. ACM Press (June 1990)
10. Eilenberg, S., Moore, J.C.: Adjoint functors and triples. Illinois J. Math 9(3), 381–398 (1965)
11. Ghani, N., Uustalu, T., Vene, V.: Build, Augment and Destroy, Universally. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 327–347. Springer, Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-30477-7_22

12. Hinze, R.: Efficient monadic-style backtracking. Tech. Rep. IAI-TR-96-9, Institut für Informatik III, Universität Bonn (October 1996)
13. Hinze, R.: Deriving backtracking monad transformers. In: Wadler, P. (ed.) Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), pp. 186–197. ACM, New York (2000)
14. Hinze, R.: Adjoint folds and unfolds—an extended study. Science of Computer Programming (2011) (to appear)
15. Hinze, R., James, D.W.H.: Reason isomorphically! In: Oliveira, B.C., Zalewski, M. (eds.) Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming (WGP 2010), pp. 85–96. ACM, New York (2010)
16. Hoare, C.A.R., He, J.: The weakest prespecification. Inf. Process. Lett. 24(2), 127–132 (1987)
17. Huber, P.J.: Homotopy theory in general categories. Mathematische Annalen 144, 361–385 (1961), http://dx.doi.org/10.1007/BF01396534, doi:10.1007/BF01396534
18. Hughes, J.: The Design of a Pretty-Printing Library. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 53–96. Springer, Heidelberg (1995)
19. Hughes, R.J.M.: A novel representation of lists and its application to the function reverse. Information Processing Letters 22(3), 141–144 (1986)
20. Hutton, G., Jaskelioff, M., Gill, A.: Factorising folds for faster functions. Journal of Functional Programming 20(Special Issue 3-4), 353–373 (2010), http://dx.doi.org/10.1017/S0956796810000122
21. Jaskelioff, M.: Modular Monad Transformers. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 64–79. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-00590-9_6
22. Johann, P., Ghani, N.: Foundations for structured programming with GADTs. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 297–308. ACM, New York (2008), http://doi.acm.org/10.1145/1328438.1328475
23. Kan, D.M.: Adjoint functors. Transactions of the American Mathematical Society 87(2), 294–329 (1958)
24. Kleisli, H.: Every standard construction is induced by a pair of adjoint functors. Proceedings of the American Mathematical Society 16(3), 544–546 (1965), http://www.jstor.org/stable/2034693
25. Lambek, J.: From lambda-calculus to cartesian closed categories. In: Seldin, J., Hindley, J. (eds.) To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 376–402. Academic Press (1980)
26. Leivant, D.: Reasoning about functional programs and complexity classes associated with type disciplines. In: Proceedings 24th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1983, Tucson, AZ, USA, pp. 460–469. IEEE Computer Society Press, Los Alamitos (1983)
27. Mac Lane, S.: Categories for the Working Mathematician, 2nd edn. Graduate Texts in Mathematics. Springer, Berlin (1998)
28. McBride, C.: Functional pearl: Kleisli arrows of outrageous fortune. Journal of Functional Programming (to appear)
29. Mellish, C., Hardy, S.: Integrating Prolog into the Poplog environment. In: Campbell, J. (ed.) Implementations of Prolog, pp. 533–535. Ellis Horwood Limited (1984)
30. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)

31. Penrose, R.: Applications of negative dimensional tensors. In: Welsh, D. (ed.) Combinatorial Mathematics and its Applications: Proceedings of a Conference held at the Mathematical Institute, Oxford, July 7-10, 1969, pp. 221–244. Academic Press (1971)
32. Selinger, P.: A survey of graphical languages for monoidal categories. In: Coecke, B. (ed.) New Structures for Physics. Springer Lecture Notes in Physics, vol. 813, pp. 289–355. Springer, Heidelberg (2011)
33. Voigtländer, J.: Asymptotic Improvement of Computations over Free Monads. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 388–403. Springer, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-70594-9_20
34. Wadler, P.: Theorems for free! In: The Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA 1989), London, UK, pp. 347–359. Addison-Wesley Publishing Company (September 1989)
35. Wand, M., Vaillancourt, D.: Relating models of backtracking. In: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, pp. 54–65. ACM, New York (2004), http://doi.acm.org/10.1145/1016850.1016861