JFP **35**, e7, 27 pages, 2025. © The Author(s), 2025. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.
doi:10.1017/S0956796825000012

FUNCTIONAL PEARL

How much is in a square? Calculating functional programs with squares

JOSE NUNO OLIVEIRA

HASLab - INESC TEC, University of Minho,
Gualtar Campus, Building E7, Braga, Portugal
(*e-mail:* jno@di.uminho.pt)

Abstract

Experience in teaching functional programming (FP) on a relational basis has led the author to focus on a graphical style of expression and reasoning in which a geometric construct shines: the (semi) commutative square. In the classroom this is termed the "magic square" (MS), since virtually everything that we do in logic, FP, database modeling, formal semantics and so on fits in some MS geometry. The sides of each magic square are binary relations and the square itself is a comparison of two paths, each involving two sides. MSs compose and have a number of useful properties. Among several examples given in the paper ranging over different application domains, *free-theorem* MSs are shown to be particularly elegant and productive. Helped by a little bit of Galois connections, a generic, induction-free theory for foldr and foldl is given, showing in particular that foldl s = foldr ($flip_ s$) holds under conditions milder than usually advocated.

1 Introduction

> *(...) A special feature of our approach is a general calculus of relations presented in part two. This calculus offers another, often more amenable framework for concepts and methods discussed in part one.*
>
> — Freyd & Ščedrov, *Categories, Allegories*, 1990.

Functions are mathematical objects, and functional programming (FP) has benefitted much from its strong mathematical foundations over the years. This began with LISP, a programming language born in the late 1950s that implemented Church's λ-calculus and included higher-order functions, that is, functions that manipulate other functions. This soon proved to be a strong competitive advantage.

Another significant advantage, originally also identified by Church, is *parametric polymorphism*, a powerful device of great practical and theoretical relevance

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

(Damas & Milner, 1982). Pragmatically, it leads to a programming style in which programs are *generic* and can be better *statically checked*, adding to programming productivity since one writes *less* and *better* code — *"programs for cheap!"* in the words of Hackett & Hutton (2015). This comes hand in hand with another advantage, evident when one needs to reason about programs: parametricity theory ensures structural properties of functional code that need not be proved case by case — cf. the *"theorem for free"* motto coined by Wadler (1989). Thus correctness-proving becomes more productive as well.

Curiously enough, parametricity theory found an effective formulation in terms of relations and not just functions (Reynolds, 1983). As Janis Voigtländer (2019) puts it, *the key to deriving free theorems is to interpret types as relations*. Indeed, it often happens in mathematics that, to better address a concept, it is convenient to invest into another one of wider scope. For example, the laws of basic trigonometry are best formulated if the domain of discourse is extended from real to complex numbers — recall Euler's formula $e^{ix} = \cos x + i \sin x$ and all that follows from it. The same happens between functions and relations, the latter extending the former. The quote by Freyd & Scedrov (1990) that opens this paper reveals exactly this, at the categorial level.

Building on this perspective, Bird & de Moor (1997) developed an *algebra of programming* in which specifications (relations) lead to implementations (functions) by calculation. However, the application of relational techniques to FP is actually wider, see e.g. the work by Backhouse & Backhouse (2004) among much other research reported in the literature. This paper will follow the same path of using relations to reason about functional programs, taking a rather pragmatic view inspired by experience in the classroom.

Teaching FP is not an easy task, all the more so when it comes late in the syllabus, after imperative and object-oriented programming. FP calls for a plan of the overall architecture before starting writing code, with particular emphasis on designing a suitable information flow.[1] It is therefore important to help programmers in identifying "big picture" design patterns.

It can be claimed that the higher-order combinators so much used in FP enable design patterns for free by inter-combination — see e.g. *map-filter*, *MapReduce* (Lämmel, 2008) and so on. In this paper we propose what we call "magic squares" (MS) to expand that view by expressing and composing quite common relational (and functional) patterns that arise in problem analysis and modeling.

Among several examples given in the paper, free-theorem MS prove to be elegant and productive. As observed by Voigtländer (2009), free theorems promise a lot but deriving them is not immediate. It is error-prone and can be tedious and result in intricate logical formulæ hard to simplify into something short and effective, even when using a mechanised generator. We claim that relational reasoning can be of great help in this respect. In this paper, free theorems are expressed in terms of magic squares, whose graphical nature makes it easier to perceive what is going on in the reasoning.

As an example of application, a generic, induction-free theory for foldr and foldl is given — understood in a class wider than just finite lists[2] — showing in particular that foldl s = foldr ($\mathit{flip_s}$) holds under conditions milder than usually advocated.

[1] In this respect, FP points to future programming paradigms, namely quantum programming (Neri *et al.*, 2022).
[2] Recall the *Foldable* class in Haskell.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

2 "Magic" squares

Experience in teaching FP using a relational approach has led the author to a graphical style of expression and reasoning in which a very simple, geometric construct shines: the (semi) *commutative square*.

In the classroom this is termed the "magic square" (MS), because much of what we do in logic, FP, database modeling, formal semantics, etc fits in some MS geometry. Each magic square has four sides, say R, P, S, Q, which are binary relations, and each square is a comparison between two paths, of two relations each:

$$\begin{array}{ccc} A & \xleftarrow{R} & B \\ {\scriptstyle P}\downarrow & \subseteq & \downarrow{\scriptstyle Q} \\ C & \xleftarrow[S]{} & D \end{array} \qquad P \cdot R \subseteq S \cdot Q \tag{2.1}$$

In more detail, each side is an arrow, say $A \xleftarrow{R} B$, declaring a binary relation R that relates objects of types A and B. Its meaning is as usual: given objects $a \in A$ and $b \in B$, the proposition $a\,R\,b$ tells whether or not a and b are related by R. Take for instance relation $\mathbb{N}_0 \xleftarrow{(\leqslant)} \mathbb{N}_0$. Clearly, $0 \leqslant 1$ holds (it is a true proposition) while $1 \leqslant 0$ does not.

Next, we need to say what a *path* means, say $P \cdot R$ in (2.1): given some $c \in C$ and some $b \in B$, $c\,(P \cdot R)\,b$ holds whenever there is some mediator $a \in A$ such that both $c\,P\,a$ and $a\,R\,b$ hold. We say that relation $P \cdot R$ is the *composition* of relations P and R. Relational composition is associative.

Finally, let us say what the comparison $R \subseteq S$ of two relations $B \xleftarrow{R,S} A$ means, in general. ($R \subseteq S$ should be read: "R is at most S" or "R is included in S".) It means that, for all $b \in B$ and $a \in A$, if $b\,R\,a$ holds then $b\,S\,a$ holds too. In summary, relation *inclusion* "hides" a *universal* quantifier, while relation *composition* hides an *existential* quantifier. In symbols, the logic interpretation of (2.1) is:

$$\langle \forall\, c, b \;::\; \langle \exists\, a \;::\; c\,P\,a \wedge a\,R\,b\rangle \Rightarrow \langle \exists\, d \;::\; c\,S\,d \wedge d\,Q\,b\rangle\rangle \tag{2.2}$$

The following version of (2.2) makes its connection with (2.1) more explicit:

$$\begin{array}{cccccccccc} \exists & & & a & & & & d & & \\ & & P & \cdot & R & \Rightarrow & S & \cdot & Q & \\ \forall & c & & & & b & c & & & b \end{array}$$

Comparisons $R \subseteq S$ form a partial order, and therefore are reflexive, transitive, and antisymmetric.

As will be shown shortly, one can express "a lot" using the "magic" square pattern (2.1), rendered *pointwise* in (2.2). Some terminology before giving examples: we shall refer to $R \subseteq S$ as a relational *inequality* in which R is the *pre* (or *lower*) side and S is the *post* (or *upper*) side. In a composition $R \cdot S$, relation S (resp. R) will be referred to as the *producer*

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

(resp. *consumer*) relation. This terminology bears some relationship to the way *information flows* in a magic square, cf.

Relation	Role
R	pre-producer
P	pre-consumer
Q	post-producer
S	post-consumer

Magic squares compose, not only horizontally

$$\begin{array}{ccccc} A & \xleftarrow{R} & C & \xleftarrow{R'} & C' \\ {\scriptstyle P}\downarrow & \subseteq & \downarrow{\scriptstyle Q} & \subseteq & \downarrow{\scriptstyle Q'} \\ B & \xleftarrow[S]{} & D & \xleftarrow[S']{} & D' \end{array} \quad \Longrightarrow \quad \begin{array}{ccc} A & \xleftarrow{R\cdot R'} & C' \\ {\scriptstyle P}\downarrow & \subseteq & \downarrow{\scriptstyle Q'} \\ B' & \xleftarrow[S\cdot S']{} & D' \end{array} \qquad (2.3)$$

but also vertically:

$$\begin{array}{ccc} A & \xleftarrow{R} & C \\ {\scriptstyle P}\downarrow & \subseteq & \downarrow{\scriptstyle Q} \\ B & \xleftarrow[S]{} & D \\ {\scriptstyle P'}\downarrow & \subseteq & \downarrow{\scriptstyle Q'} \\ B' & \xleftarrow[S']{} & D' \end{array} \quad \Longrightarrow \quad \begin{array}{ccc} A & \xleftarrow{R} & C \\ {\scriptstyle P'\cdot P}\downarrow & \subseteq & \downarrow{\scriptstyle Q'\cdot Q} \\ B' & \xleftarrow[S']{} & D' \end{array} \qquad (2.4)$$

Every type X has its own *identity* relation $X \xleftarrow{id} X$, which is such that $x' \, id \, x \Leftrightarrow x' = x$. Therefore, $R \cdot id = R = id \cdot R$ holds for any R and thus squares involving id degenerate into triangles or even "sides". So, squares in which both pre-consumer and post-producer are identities behave like units of square composition, cf.:

$$\begin{array}{ccc} A & \xleftarrow{R} & C \\ {\scriptstyle id}\downarrow & \subseteq & \downarrow{\scriptstyle id} \\ A & \xleftarrow{R} & C \\ {\scriptstyle P}\downarrow & \subseteq & \downarrow{\scriptstyle Q} \\ B & \xleftarrow[S]{} & D \\ {\scriptstyle id}\downarrow & \subseteq & \downarrow{\scriptstyle id} \\ B & \xleftarrow[S]{} & D \end{array} \quad \equiv \quad \begin{array}{ccc} A & \xleftarrow{R} & C \\ {\scriptstyle P}\downarrow & \subseteq & \downarrow{\scriptstyle Q} \\ B & \xleftarrow[S]{} & D \end{array} \qquad (2.5)$$

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

3 Functions

Statements like *John loves Mary* can be written as *Mary is loved by John* without changing meaning. The latter version is known as the *passive voice* of the former. Generalizing, any proposition $b\,R\,a$ can be converted to passive voice, written $a\,R^{\circ}\,b$. (In the example, $loves^{\circ} = is\ loved\ by$.) The $(_)^{\circ}$ operator, known as relational *converse*, will play a major role in our setting.

Converse is involutive ($(R^{\circ})^{\circ} = R$) and contravariant: $(R \cdot S)^{\circ} = S^{\circ} \cdot R^{\circ}$. So one can take the converse of a square, obtaining an equivalent square where producer and consumer relations swap roles, once conversed:

$$\begin{array}{ccc} A & \xleftarrow{R} & C \\ {\scriptstyle P}\downarrow & \subseteq & \downarrow{\scriptstyle Q} \\ B & \xleftarrow[S]{} & D \end{array} \quad \Leftrightarrow \quad \begin{array}{ccc} A & \xleftarrow{P^{\circ}} & B \\ {\scriptstyle R^{\circ}}\downarrow & \subseteq & \downarrow{\scriptstyle S^{\circ}} \\ C & \xleftarrow[Q^{\circ}]{} & D \end{array}$$

Some relations, termed *functions* and denoted in lowercase — see e.g. f in (3.1) below — are special in the sense of fitting in two particular squares involving them and their converses:

$$\begin{array}{ccccc} A & \xleftarrow{id} & A & \xleftarrow{f^{\circ}} & B \\ {\scriptstyle id}\downarrow & \subseteq & \downarrow{\scriptstyle f} & \subseteq & \downarrow{\scriptstyle id} \\ A & \xleftarrow[f^{\circ}]{} & B & \xleftarrow[id]{} & B \end{array} \tag{3.1}$$

By (2.2), the square on the left ($id \subseteq f^{\circ} \cdot f$) converts to:

$$\langle \forall\, a\ ::\ \langle \exists\, b\ ::\ b\,f\,a \rangle \rangle$$

In words, f reacts to every possible input (it is "total"). Similarly, the square on the right ($f \cdot f^{\circ} \subseteq id$) converts to:

$$\langle \forall\, b, b'\ ::\ \langle \exists\, a\ ::\ b\,f\,a \wedge b'\,f\,a \rangle \Rightarrow (b = b') \rangle$$

In words: f is univocal. Because f in (3.1) always yields some output for any possible input, and such an output is unique, one can interpret the proposition $b\,f\,a$ as $b = f\,a$, that is, b is unequivocally *the* output of f once a is passed as input. This saves one from the two existential quantifiers of a very frequent and useful pattern involving functions and function converses:[3]

$$b\,(f^{\circ} \cdot R \cdot g)\,a \ \Leftrightarrow\ (f\,b)\,R\,(g\,a) \qquad B \xrightarrow{f} C \xleftarrow{R} D \xleftarrow{g} A, \quad B \xleftarrow{f^{\circ} \cdot R \cdot g} A \tag{3.2}$$

[3] To improve readership in pointwise formulæ, we follow the convention of assuming ∀ by default wherever no quantifier is specified.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

We shall refer to (3.2) as the *"nice" rule*, where f and g are functions and R is not constrained.[4]

It can be shown that functions — i.e. relations f fitting in (3.1) — are precisely those that satisfy the following square equivalences,

$$f \cdot R \subseteq Q \quad \equiv \quad R \subseteq f^{\circ} \cdot Q \tag{3.3}$$
$$R \cdot f^{\circ} \subseteq Q \quad \equiv \quad R \subseteq Q \cdot f \tag{3.4}$$

cf.

$$\begin{array}{ccc} A & \xleftarrow{R} & C \\ {\scriptstyle f}\downarrow & \subseteq & \downarrow{\scriptstyle Q} \\ B & \xleftarrow[id]{} & B \end{array} \quad \Leftrightarrow \quad \begin{array}{ccc} A & \xleftarrow{R} & C \\ {\scriptstyle id}\downarrow & \subseteq & \downarrow{\scriptstyle Q} \\ A & \xleftarrow[f^{\circ}]{} & B \end{array}$$

— similarly for (3.4). These equivalences, known as "shunting rules," are very useful for reasoning about functions. One particular follow-up is that comparing functions amounts to equating them,

$$f \subseteq g \Leftrightarrow f = g \tag{3.5}$$

as can be easily shown:

$$\begin{array}{ll} & f = g \\ \equiv & \quad \{ \ \subseteq\text{-antisymmetry} \ \} \\ & f \subseteq g \wedge g \subseteq f \\ \equiv & \quad \{ \ (3.3) \text{ (resp. } (3.4)) \text{ for function } g \text{ (resp. } f) \ \} \\ & f \subseteq g \wedge f^{\circ} \subseteq g^{\circ} \\ \equiv & \quad \{ \ \text{converse is an isomorphism} \ \} \\ & f \subseteq g \wedge f \subseteq g \\ \equiv & \quad \{ \ \text{trivial} \ \} \\ & f \subseteq g \end{array}$$

4 Reynolds squares

In the sequel, we shall be particularly interested in squares (2.1) in which both pre-consumers and post-producers are functions, say f and g in (4.1) given next:

[4] As already mentioned, for easy reference functions will be written in lowercase (e.g. f, g, h, ...). Arbitrary relations will be written uppercase, as e.g. R, S, ... above.

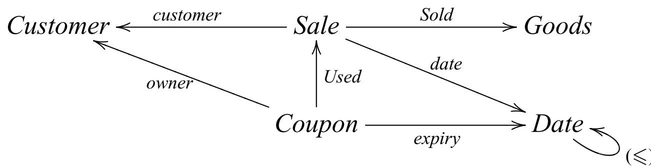
Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

$$\begin{array}{ccc} A & \xleftarrow{R} & B \\ {\scriptstyle f}\downarrow & \subseteq & \downarrow{\scriptstyle g} \\ C & \xleftarrow[S]{} & D \end{array} \qquad f \cdot R \subseteq S \cdot g \tag{4.1}$$

By (3.3) followed by the "nice rule" (3.2), square (4.1) captures the situation in which, for R-related inputs, the two functions f and g always yield S-related outputs: if $a\,R\,b$ holds then $(f\ a)\,S\,(g\ b)$ holds, for any a and b.

We shall refer to squares of pattern (4.1) as *Reynolds squares* because of the role they will play in expressing *free theorems* of parametric polymorphic functions, due to John Reynolds (1983) and made popular by Philip Wadler (1989). Note however that pattern (4.1) can be found in far more mundane settings, for instance relational database modeling.

Take for instance the very simple data model below of a small grocery in which customers can use coupons to obtain discounts when purchasing goods:



The "polygons" of the data model can be regarded as "opportunities" to dig up business rules, i.e. data invariants. Indeed, the triangle on the right can be oriented as the square

$$\begin{array}{ccc} \mathit{Sale} & \xleftarrow{\mathit{Used}} & \mathit{Coupon} \\ {\scriptstyle \mathit{date}}\downarrow & \subseteq & \downarrow{\scriptstyle \mathit{expiry}} \\ \mathit{Date} & \xleftarrow[(\leqslant)]{} & \mathit{Date} \end{array} \tag{4.2}$$

which is indeed meaningful: *coupons cannot be used beyond their expiry date*. And the lefthand triangle can be oriented as

$$\begin{array}{ccc} \mathit{Sale} & \xleftarrow{\mathit{Used}} & \mathit{Coupon} \\ {\scriptstyle \mathit{customer}}\downarrow & \subseteq & \downarrow{\scriptstyle \mathit{owner}} \\ \mathit{Customer} & \xleftarrow[\mathit{id}]{} & \mathit{Customer} \end{array} \tag{4.3}$$

bearing in it another relevant business rule: *coupons can only be used by the customers who own them.*[5]

[5] Note that (4.2,4.3) are Reynolds squares because (mandatory) attributes *date*, *expiry*, and so on can be regarded as functions.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

Squares in general (2.1) arise naturally in formal modeling in various guises, involving data relations (e.g. key-value maps), object attributes etc, and capturing not only domain-specific invariants such as those illustrated above but also structural ones. Particularly frequent squares are those that capture *referential integrity*,

$$\begin{array}{ccc} A & \xleftarrow{R} & K \\ {\scriptstyle f}\downarrow & \subseteq & \downarrow{\scriptstyle \top} \\ K' & \xleftarrow[S^\circ]{} & B \end{array} \qquad a\,R\,k \Rightarrow \langle \exists\, b\ ::\ b\,S\,(f\ a)\rangle \tag{4.4}$$

where $\top$ is the largest relation of its type, that is, $b \top k = \text{TRUE}$ for all b and k. Think of R and S as key-value stores with keys K and K', respectively, the latter playing the role of *foreign key* referred to by attribute f of A (a function).[6]

Reynolds arrow. Back to the genericity of Reynolds squares, fix the pre-producer R and the post-consumer S of (4.1). Then (4.1) expresses a (higher-order) relation on functions $f\,(R \to S)\,g$ defined by:

$$f\,(R \to S)\,g \quad\equiv\quad f \cdot R \subseteq S \cdot g \tag{4.5}$$

Using the exponential notation Y^X to denote the type of all functions from type X to type Y, and extending it to relations, $S^R = (R \to S)$, one has:

$$\frac{A \xleftarrow{R} B \qquad C \xleftarrow{S} D}{C^A \xleftarrow{S^R} D^B}$$

Thus we obtain the *Reynolds arrow*, a higher-order relational operator that builds, from relations R and S, the (higher-order) relation S^R on functions identified by Backhouse (1990) and defined above by (4.5). We will use the notations S^R, $R \to S$ and $S \leftarrow R$ interchangeably, depending on how convenient they are. (E.g. we avoid $(S^R)^Q$ and write $Q \to S^R$ instead.)

For instance, the $Used \to (\leqslant)$ higher-order relation relates attributes *date* and *expiry* in (4.2), expressing the semantic constraint that the *use* of a coupon can only occur on a date *prior* $(\leqslant)$ to its expiry date.

Reynolds arrows are central to the inference of *free theorems*, as will be shown later, in particular in the following situation, in which R is the converse of a function ($R := h^\circ$) and S is a function ($S := k$):

$$\begin{array}{ccc} A & \xleftarrow{h^\circ} & B \\ {\scriptstyle f}\downarrow & \subseteq & \downarrow{\scriptstyle g} \\ C & \xleftarrow[k]{} & D \end{array} \qquad f\,(h^\circ \to k)\,g \Leftrightarrow f \cdot h^\circ \subseteq k \cdot g \tag{4.6}$$

[6] See e.g. Oliveira (2009) and Oliveira & Ferreira (2013) for concrete examples arising in modeling a flash file system. Note that (4.4) is not a Reynolds square because $\top$ is not a function.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

By shunting h° in (4.6) via (3.4), $h^\circ \to k$ becomes the higher-order function

$$(h^\circ \to k)\, g \quad = \quad k \cdot g \cdot h \tag{4.7}$$

as can be easily checked:

$$\begin{array}{ll} & f\,(h^\circ \to k)\, g \\ \equiv & \qquad \{\ \text{Reynolds (4.1)}\ \} \\ & f \cdot h^\circ \subseteq k \cdot g \\ \equiv & \qquad \{\ \text{nice rule (3.2)}\ \} \\ & f \subseteq k \cdot g \cdot h \\ \equiv & \qquad \{\ \text{equality of functions (3.5)}\ \} \\ & f = k \cdot g \cdot h \end{array}$$

The special cases of (4.7),

$$(id \to k)\, g = k \cdot g \tag{4.8}$$

(for $A = B$) and

$$(h^\circ \to id)\, g = g \cdot h \tag{4.9}$$

(for $C = D$) capture post-and pre-composition, that is:

$$(id \to k) = (k\cdot) \quad \text{— cf. covariant exponentials} \tag{4.10}$$
$$(h^\circ \to id) = (\cdot h) \quad \text{— cf. contravariant exponentials} \tag{4.11}$$

By taking converses of both sides of (4.5) and swapping f and g via shunting rules (3.3,3.4), one gets

$$(S^R)^\circ \quad = \quad (S^\circ)^{(R^\circ)} \tag{4.12}$$

Thus:

$$id^h = (\cdot h)^\circ \tag{4.13}$$
$$id^{id} = id \tag{4.14}$$

Higher-order Reynolds squares. Exponential relations S^R can involve other exponentials, for instance $(S^Q)^R$ i.e. $R \to S^Q$. This happens frequently with functional programmers, who tend to use functions curried rather than uncurried. Such is the case in

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

$$\begin{array}{ccc} A & \xleftarrow{R} & B \\ {\scriptstyle f}\downarrow & \subseteq & \downarrow{\scriptstyle g} \\ X^C & \xleftarrow[S^Q]{} & Y^D \end{array} \qquad f\,(R \to S^Q)\,g$$

Let us unfold this, assuming all fresh variables universally quantified:

$$\begin{array}{ll} & f\,(R \to S^Q)\,g \\ \equiv & \quad \{\ (4.5)\ \} \\ & f \cdot R \subseteq S^Q \cdot g \\ \equiv & \quad \{\ (3.3) \text{ followed by } (3.2)\ \} \\ & a\,R\,b \Rightarrow (f\ a)\,S^Q\,(g\ b) \\ \equiv & \quad \{\ (4.5) \text{ again }\ \} \\ & a\,R\,b \Rightarrow ((f\ a) \cdot Q \subseteq S \cdot (g\ b)) \\ \equiv & \quad \{\ \text{again } (3.3) \text{ followed by } (3.2)\ \} \\ & a\,R\,b \Rightarrow c\,Q\,d \Rightarrow (f\ a\ c)\,S\,(g\ b\ d) \end{array} \tag{4.15, 4.16}$$

Perfect squares. All relations in square (4.6) but the pre-producer are functions. Suppose now that *all* relations are functions in a *magic* square (2.1). Then, by (3.5) path comparison becomes path equality and producer/consumer paths become interchangeable. In this case, we drop the $\subseteq$ symbol from the square:

$$\begin{array}{ccc} A & \xleftarrow{h} & B \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ C & \xleftarrow[k]{} & D \end{array} \qquad f \cdot h = k \cdot g$$

Clearly:

$$f\,k^h\,g \quad \equiv \quad f \cdot h = k \cdot g \tag{4.17}$$

5 More on Reynolds square expressiveness

Let the pre-producer R of a Reynolds square (4.1) be the identity:

$$\begin{array}{ccc} A & \xleftarrow{id} & A \\ {\scriptstyle f}\downarrow & \subseteq & \downarrow{\scriptstyle g} \\ C & \xleftarrow[S]{} & D \end{array} \qquad f \subseteq S \cdot g \tag{5.1}$$

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

In this case, f and g have the same input type and are said to be *pointwise S-related*. Using the dotted notation $\dot{S}$ for S^{id}, one may write $f\ \dot{S}\ g$ to mean square (5.1), that is: $\langle \forall\ a\ ::\ (f\ a)\ S\ (g\ a) \rangle$. Thus, the following (useful) "lifting" rule:

$$f\ S^{id}\ g \Leftrightarrow f\ \dot{S}\ g \Leftrightarrow (f\ x)\ S\ (g\ x) \tag{5.2}$$

A typical situation arises in (5.1) when the post-consumer S is an ordering $(\leqslant)$. Then $f\ \dot{\leqslant}\ g$ (i.e. $f \subseteq (\leqslant) \cdot g$) means that $f\ a \leqslant g\ a$ for every input a, i.e. f is *pointwise-smaller* than g with respect to $(\leqslant)$.

Relational types. The intersection of S^R and (higher-order) *id* captures all Reynolds squares (4.1) in which $f = g$:

$$\begin{array}{ccc} A & \xleftarrow{R} & A \\ {\scriptstyle f}\downarrow & \subseteq & \downarrow{\scriptstyle f} \\ C & \xleftarrow[S]{} & C \end{array} \qquad f \cdot R \subseteq S \cdot f \tag{5.3}$$

In this case, we often abbreviate $f\ (R \to S)\ f$ to $f : R \to S$, meaning that f has *relational type* $R \to S$.[7] As seen before, this means that f maps R-related inputs to S-related outputs.

We can also write $R \xrightarrow{f} S$ to express (5.3), stressing the fact that f can be regarded as a *morphism* of a category in which relations are the objects, mapped by functions that preserve them in the sense of (5.3). Indeed, by vertical composition (2.4) such morphisms compose and composition has identities (2.5).[8]

For R and S instantiated to preorders, say $R, S := (\sqsubseteq), (\leqslant)$, the square $f : (\sqsubseteq) \to (\leqslant)$ is a concise way of saying that f is a *monotonic* function: $\langle \forall\ a, a'\ ::\ a \sqsubseteq a' \Rightarrow (f\ a) \leqslant (f\ a') \rangle$.

Predicates as types. A special case of square (5.3) pops up wherever R and S are *partial identities*, also called *coreflexive* relations ($R \subseteq id$ and $S \subseteq id$). These relations are one-to-one correspondent to predicates: given a predicate $p : A \to \mathbb{B}$, its associated coreflexive is the relation $p? : A \to A$ defined by

$$a'\ (p?)\ a \Leftrightarrow p\ a \wedge a' = a$$

Let f have relational type $p? \to q?$, as shown next:

$$\begin{array}{ccc} A & \xleftarrow{p?} & A \\ {\scriptstyle f}\downarrow & \subseteq & \downarrow{\scriptstyle f} \\ C & \xleftarrow[q?]{} & C \end{array} \qquad f \cdot p? \subseteq q? \cdot f \tag{5.4}$$

[7] Note how *type variables* A and C in $f : A \to C$ are straightforwardly replaced by *relations* R and S in $f : R \to S$, respectively. Thus types are easily interpreted as relations (Voigtländer, 2019) in our relational setting.

[8] This category is named Rel_2 in Plotkin *et al*. (2000). Hence, what is termed *relational type* $R \to S$ in this paper corresponds to the homset $Rel_2\ (R, S)$. Rel_2 is cartesian closed, meaning that homset $R \to Q^S$ is, by currying, isomorphic to $R \times S \to Q$, where the "tensor" product of two relations is defined in the expected way: $(y, x)\ (R \times S)\ (b, a) \Leftrightarrow y\ R\ b \wedge x\ S\ a$.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

It is easy to see that (5.4) converts to the pointwise $\langle \forall\ a\ ::\ p\ a \Rightarrow q\ (f\ a)\rangle$, meaning that p is a *sufficient* condition on the input of f (*pre-condition*) for q to hold on the output (*post-condition*). By a not so dangerous abuse of notation one might abbreviate $f : p? \rightarrow q?$ to $f : p \rightarrow q$ and regard *predicates* p and q as *types*.[9]

Another perspective is to look at (5.4) as a *Hoare-triple* $p\ \{f\}\ q$ where f is a functional program that satisfies such pre/post conditions.[10]

Algebraic squares. A quite interesting situation arises in a Reynolds square (4.1) when $f, g := \alpha, \beta$, where α and β are *algebras* of a *relator* F:[11]

$$\begin{array}{ccc} \mathsf{F}\,A & \xleftarrow{\mathsf{F}\,R} & \mathsf{F}\,B \\ {\scriptstyle\alpha}\downarrow & \subseteq & \downarrow{\scriptstyle\beta} \\ A & \xleftarrow[R]{} & B \end{array} \qquad \alpha \cdot \mathsf{F}\,R \subseteq R \cdot \beta \tag{5.5}$$

The pointwise equivalent of (5.5) is $x\,(\mathsf{F}\,R)\,y \Rightarrow (\alpha\ x)\,R\,(\beta\ y)$, for all x and y. For $\alpha = \beta$ and endo-relation R, $\alpha : \mathsf{F}\,R \rightarrow R$ says that R is *compatible* with algebra α. For R an *equivalence relation*, it further says that R is a *congruence relation* with respect to α.[12] In case R is a *function*, say $R := h$ in (5.5), the square becomes perfect

$$\begin{array}{ccc} \mathsf{F}\,A & \xleftarrow{\mathsf{F}\,h} & \mathsf{F}\,B \\ {\scriptstyle\alpha}\downarrow & & \downarrow{\scriptstyle\beta} \\ A & \xleftarrow[h]{} & B \end{array} \qquad \alpha \cdot \mathsf{F}\,h = h \cdot \beta$$

meaning that h is an F-*homomorphism* (Bird & de Moor, 1997).

Coalgebraic squares. Let $f, g := \gamma, \phi$ in a Reynolds square, where γ and ϕ are F-**coalgebras**:

$$\begin{array}{ccc} A & \xleftarrow{R} & B \\ {\scriptstyle\gamma}\downarrow & \subseteq & \downarrow{\scriptstyle\phi} \\ \mathsf{F}\,A & \xleftarrow[\mathsf{F}\,R]{} & \mathsf{F}\,B \end{array} \qquad \gamma \cdot R \subseteq \mathsf{F}\,R \cdot \phi \tag{5.6}$$

[9] Such arrows $f : p \rightarrow q$ form a sub-category of Rel_2 (mentioned in footnote 8) whose objects are coreflexives (i.e. predicates). Thus one gets *Curry-Howard* (Wadler, 2015) in a relational setting.

[10] See e.g. Oliveira (2009) for a treatment of Hoare logic in this way, an approach that has long been known to extend to relations (non-deterministic programs), see e.g. the pioneering work of de Bakker & de Roever (1972).

[11] Recall that a *functor* is a structural map $\mathsf{F} : (A \rightarrow B) \rightarrow (\mathsf{F}\,A \rightarrow \mathsf{F}\,B)$ respecting the identity ($\mathsf{F}\,id = id$) and composition, $\mathsf{F}\,(f \cdot g) = \mathsf{F}\,f \cdot \mathsf{F}\,g$. *Relators* extend functors to relations, $y\,(\mathsf{F}\,R)\,x$ meaning that every b in F-structure y is related to the corresponding a in F-structure x via R, that is, $b\,R\,a$ holds. Furthermore, relators are monotonic and preserve converses: $\mathsf{F}\,(R^\circ) = (\mathsf{F}\,R)^\circ$ (Backhouse & Backhouse, 2004).

[12] Note also that, in a rather succinct way, the square also says that R is a *logical relation* between α and β (Plotkin *et al.*, 2000).

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

Then R is said to be a *bisimulation* (Sangiorgi, 2004) between the two coalgebras, meaning:

$$\langle \forall\, a, b \;::\; a\, R\, b \Rightarrow (\gamma\; a)\, \mathsf{F}\, R\, (\phi\; b) \rangle$$

In case R is coreflexive, it is known as a *coalgebraic invariant* (Barbosa *et al.*, 2008). Squares (5.5) and (5.6) show that, in a sense, a bisimulation is a kind of co-logical-relation.

Galois connections. To complete this review of concepts that *magic* squares are able to easily capture, let R and S in (4.1) be two preorders, $(\sqsubseteq)$ and $(\leqslant)$ respectively. Moreover, reverse the direction of f and let the square be perfect:

$$\begin{array}{ccc} A & \xleftarrow{(\sqsubseteq)} & A \\ {\scriptstyle f^{\circ}}\downarrow & & \downarrow{\scriptstyle g} \\ B & \xleftarrow[(\leqslant)]{} & B \end{array} \qquad f^{\circ} \cdot (\sqsubseteq) = (\leqslant) \cdot g \tag{5.7}$$

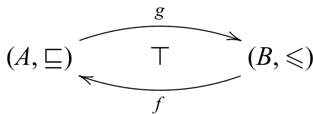
This very special kind of square is known as a *Galois connection* (GC) between two so-called *adjoint* functions, f (the *lower* or *left* adjoint) and g (the *upper* or *right* adjoint). Going pointwise in (5.7) via (2.2,3.2) etc one gets:

$$f\, b \sqsubseteq a \;\Leftrightarrow\; b \leqslant g\, a \tag{5.8}$$

Two cancellation laws are easily obtained from (5.8):

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

These inequalities tell that f and g are *inverses* of each other in an *imperfect* way: the round-trip $f \cdot g$ loses information while $g \cdot f$ yields an over-approximation of its input. This is captured by the diagram



and often abbreviated by simply writing $f \dashv g$.

Examples of the usefulness of GCs in both mathematics and computing are abundant, see e.g. von Karger (1998), Backhouse & Backhouse (2004), Mu & Oliveira (2011). The following GC between multiplication and whole division in the natural numbers

$$a \times y \leqslant x \quad \Leftrightarrow \quad a \leqslant x \div y \tag{5.9}$$

— $(\times y) \dashv (\div y)$ in short, for $y \neq 0$ — will be of interest in the sequel.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

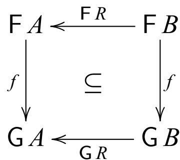
6 Free theorem squares

In his landmark paper *Theorems for free!*, Philip Wadler (1989) wrote: "*From the type of a polymorphic function we can derive a theorem that it satisfies.* (...) *How useful are the theorems so generated? Only time and experience will tell (...)*". Under this *"theorems for free"* catchy phrase, Wadler made popular an important result on parametric polymorphism due to John Reynolds (1983). Four decades later there is ample evidence that such free theorems are indeed very useful, see for instance the work by Backhouse & Backhouse (2004), Voigtländer (2009) and Hackett & Hutton (2015).

The rest of this paper will be devoted to one particular application of such free theorems. Before that, the question arises: what is a *free theorem* and how does one derive it (for free)? It turns out that free theorems are magic squares. To put it simply, let a parametric function $f : \mathsf{F}\,X \to \mathsf{G}\,X$ be given. Then its free theorem states that f has *relational type*

$$f : \mathsf{F}\,R \to \mathsf{G}\,R \tag{6.1}$$

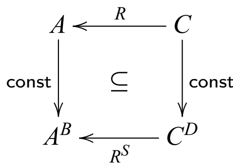
for any R relating its parameters, as shown in the corresponding square:



This extends to multiparametric functions, as shown next with a simple example: consider the Haskell constant function $\mathsf{const} : a \to b \to a$. In exponential notation (and following our uppercase notation for type parameters), we may write $\mathsf{const} : A \to A^B$. Then, by (6.1), const has relational type $R \to R^S$, that is,

$$\mathsf{const} \cdot R \subseteq R^S \cdot \mathsf{const} \tag{6.2}$$

holds, cf.



As already mentioned, this square is the free theorem of const. Recalling (4.15,4.16), its pointwise version is, for all a, b, c, d:

$$a\,R\,c \Rightarrow b\,S\,d \Rightarrow (\mathsf{const}\,a\,b)\,R\,(\mathsf{const}\,c\,d)$$

As a foretaste of what is to come, let us see how productive (6.2) is. Select, for instance, $R, S := id, k$. Then (6.2) becomes $\mathsf{const} \subseteq id^k \cdot \mathsf{const}$ that, by (4.13,3.3) and (3.5) becomes $(\cdot k) \cdot \mathsf{const} = \mathsf{const}$, which is equivalent to $\mathsf{const}\,a \cdot k = \mathsf{const}\,a$ — a well-known *fusion*

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

law for constant functions. Swapping the roles of R and S to $R, S := k, id$, one gets $\mathsf{const} \cdot k \subseteq k^{id} \cdot \mathsf{const}$ which, by (4.8) etc, rewrites to $\mathsf{const} \cdot k = (k\cdot) \cdot \mathsf{const}$ yielding another well-known law: $k \cdot \mathsf{const}\ a = \mathsf{const}\ (k\ a)$.

Flipping. As another example, consider the well-known function in Haskell that swaps the arguments of a curried function:

$$flip_ :: (a \to b \to c) \to b \to a \to c \tag{6.3}$$

Its free theorem states that *flip_* has relational type $Q^{S^R} \to Q^{R^S}$ meaning

$$g\,(R \to Q^S)\,f \quad \Rightarrow \quad (flip_\,g)\,(S \to Q^R)\,(flip_\,f)$$

that is,

$$\begin{array}{ccc} A \xleftarrow{\ R\ } X \\ g \downarrow \quad \subseteq \quad \downarrow f \\ C^B \xleftarrow[Q^S]{} Z^Y \end{array} \quad \Rightarrow \quad \begin{array}{ccc} B \xleftarrow{\ S\ } Y \\ \widetilde{g} \downarrow \quad \subseteq \quad \downarrow \widetilde{f} \\ C^A \xleftarrow[Q^R]{} Z^X \end{array} \tag{6.4}$$

where (as in the sequel) we use the tilde notation $\widetilde{f}$ to abbreviate *flip_f*, as in the right hand square above. For $Q := id, S := id$ *and* $R := r$ (a function), one gets:

$$\begin{array}{ll} & g\,(r \to id)\,f \Rightarrow \widetilde{g}\,(id \to id^r)\,\widetilde{f} \\ \equiv & \quad \{\ (4.17)\,;\,(5.1)\ \} \\ & g \cdot r = f \Rightarrow \widetilde{g} \subseteq id^r \cdot \widetilde{f} \\ \equiv & \quad \{\ id^r = (\cdot r)^\circ\ (4.13)\,;\ \text{substitution of } f;\ \text{shunting } (3.3)\ \} \\ & (\cdot r) \cdot \widetilde{g} = \widetilde{g \cdot r} \end{array}$$

This is known as the fusion-law of *flipping* (Oliveira, 2020).

Flipping will be relevant in the rest of the paper, in which we shall investigate the theory that stems from the free theorems of two very popular functional programming combinators, foldl and foldr, heading toward discussing under what conditions they have the same behaviour.

7 foldl and foldr

In Haskell, there is a class *Foldable* that, according to the standard documentation, "represents data structures that can be reduced to a summary value one element at a time".[13] The class offers, in particular, two standard ways of accessing *foldable* data structures:

$$\begin{array}{l} \mathsf{foldl} :: Foldable\ t \Rightarrow (b \to a \to b) \to b \to t\ a \to b \\ \mathsf{foldr} :: Foldable\ t \Rightarrow (a \to b \to b) \to b \to t\ a \to b \end{array} \tag{7.1}$$

[13] Quoted from Hackage's Data.Foldable, consulted December 1, 2023.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

Lists are a prominent instance of the class — and, indeed, its main inspiration — whose standard definitions are:

$$\begin{array}{l}
\mathsf{foldl}\,f\;z\,[\,] = z \\
\mathsf{foldl}\,f\;z\,(x:xs) = \mathsf{foldl}\,f\;(f\;z\;x)\;xs \\
\mathsf{foldr}\,f\;z\,[\,] = z \\
\mathsf{foldr}\,f\;z\,(x:xs) = f\;x\,(\mathsf{foldr}\,f\;z\;xs)
\end{array}$$

The free theorems of foldl and foldr — valid for any instance T of class *Foldable* — can be written in the form of relational types:[14]

$$\mathsf{foldl} : (S \to S^R) \to (S \to S^{\mathsf{T}\,R}) \tag{7.2}$$

$$\mathsf{foldr} : (R \to S^S) \to (S \to S^{\mathsf{T}\,R}) \tag{7.3}$$

There are two squares involved in each type. The one on the left will play the role of a side-condition for the one on the right to hold.

foldl. Let us see (7.2) first:

$$\begin{array}{ccc}
B & \xleftarrow{\;S\;} & Y \\
{\scriptstyle g}\downarrow & \subseteq & \downarrow{\scriptstyle f} \\
B^A & \xleftarrow[\;S^R\;]{} & Y^X
\end{array}
\quad\Rightarrow\quad
\begin{array}{ccc}
B & \xleftarrow{\;S\;} & Y \\
{\scriptstyle \mathsf{foldl}\,g}\downarrow & \subseteq & \downarrow{\scriptstyle \mathsf{foldl}\,f} \\
B^{\mathsf{T}\,A} & \xleftarrow[\;S^{\mathsf{T}\,R}\;]{} & Y^{\mathsf{T}\,X}
\end{array} \tag{7.4}$$

For $R, S := id, h$ (hence $A := X$), both $S^{\mathsf{T}\,R}$ and S^R reduce to $(h\cdot)$ by $\mathsf{T}\;id = id$[15] and (4.8). So the squares become perfect and one has:

$$\begin{array}{ccc}
B & \xleftarrow{\;h\;} & Y \\
{\scriptstyle g}\downarrow & & \downarrow{\scriptstyle f} \\
B^X & \xleftarrow[\;(h\cdot)\;]{} & Y^X
\end{array}
\quad\Rightarrow\quad
\begin{array}{ccc}
B & \xleftarrow{\;h\;} & Y \\
{\scriptstyle \mathsf{foldl}\,g}\downarrow & & \downarrow{\scriptstyle \mathsf{foldl}\,f} \\
B^{\mathsf{T}\,X} & \xleftarrow[\;(h\cdot)\;]{} & Y^{\mathsf{T}\,X}
\end{array} \tag{7.5}$$

Going pointwise via the usual laws, from (7.5) one gets:

$$h\,(f\;y\,x) = g\,(h\,y)\,x \quad\Rightarrow\quad h\,(\mathsf{foldl}\,f\;y\,xs) = \mathsf{foldl}\;g\;(h\,y)\,xs$$

This is the *fusion law* of foldl given by Bird & Gibbons (2020), who prove it (for lists) by finite-list induction. Note however that, as a corollary of a free theorem, it needs no dedicated proof and it holds for all instances of class *Foldable*, not just for lists. Moreover, the scope of (7.4) is far wider than perfect squares involving only functions, as shown next.

To give a simple example, let S be an ordering $(\leqslant)$ and keep $R := id$. Since $\mathsf{T}\;id = id$ both squares will feature the relational type $(\leqslant) \to (\leqslant)^{id}$. Then, if the left square holds, i.e. for all b, y

[14] Mentally associate relation R (resp. S) to type a (resp. b) and use exponentials where convenient.

[15] Recall footnote 11.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

$$b \leqslant y \Rightarrow (g\, b) \mathrel{\dot{\leqslant}} (f\, y)$$

then the right square

$$b \leqslant y \Rightarrow \mathsf{foldl}\; g\; b \mathrel{\dot{\leqslant}} \mathsf{foldl}\; f\; y$$

will hold too.

Perhaps more interesting is to observe that, if types a and b are replaced by predicates p and q, under the interpretation given by (5.4), then

$$\mathsf{foldl} :: \mathit{Foldable}\; t \Rightarrow (q \to p \to q) \to q \to t\, p \to q$$

is a corollary of the free theorem, saying:

$$\langle \forall\; a, b \;::\; (q\, b \wedge p\, a \Rightarrow q\, (f\; a\; b)) \Rightarrow \langle \forall\; b, x \;::\; q\, b \wedge \mathsf{all}\, p\, x \Rightarrow q\, (\mathsf{foldl}\, f\; b\, x)\rangle\rangle$$

NB: $\mathsf{all}\, p$, available for all instances of class *Foldable*, is the predicate associated to the coreflexive $\mathsf{T}\,(p?)$ that relates a T-structure to itself if and only if all data in it satisfy predicate p.

foldr. Repeating the above exercise for (7.3), on the right we get the same square as in (7.4), but the side condition on the left is different:

$$\begin{array}{ccc} A & \xleftarrow{\;R\;} & X \\ {\scriptstyle g}\big\downarrow & \subseteq & \big\downarrow{\scriptstyle f} \\ B^B & \xleftarrow[\;S^S\;]{} & Y^Y \end{array} \quad \Rightarrow \quad \begin{array}{ccc} B & \xleftarrow{\;S\;} & Y \\ {\scriptstyle \mathsf{foldr}\; g}\big\downarrow & \subseteq & \big\downarrow{\scriptstyle \mathsf{foldr}\; f} \\ B^{\mathsf{T}\,A} & \xleftarrow[\;S^{\mathsf{T}\,R}\;]{} & Y^{\mathsf{T}\,X} \end{array} \tag{7.6}$$

For $R, S := id, h$, such a side-condition square unfolds to:

$$\begin{array}{ll} & g\,(id \to h^h)\,f \\ \equiv & \quad \{\; (5.2) \;\} \\ & (g\, x)\; h^h\; (f\, x) \\ \equiv & \quad \{\; (4.17) \;\} \\ & (g\, x) \cdot h = h \cdot (f\, x) \end{array}$$

Altogether, one gets:

$$(g\, x) \cdot h = h \cdot (f\, x) \quad \Longrightarrow \quad \mathsf{foldr}\; g \cdot h = (h\cdot) \cdot \mathsf{foldr}\; f \tag{7.7}$$

Going fully pointwise,

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

$$g\,x\,(h\,y) = h\,(f\,x\,y) \quad\Rightarrow\quad h\,(\mathsf{foldr}\,f\;e\,xs) = \mathsf{foldr}\;g\;(h\;e)\,xs \tag{7.8}$$

one obtains the *fusion law* of foldr proved for finite lists (by induction) by Bird & Gibbons (2020). Again we stress that (7.7) is a corollary of a free theorem and needs no proof, holding for all instances of class *Foldable*.

Furthermore, by swapping the roles of R and S and making $S, R := id, h$, the side-condition square of (7.6) becomes $g\,(id^h)\,f$ which, by (4.17) is $f = g \cdot h$. Replacing f in the right square we get

$$\begin{array}{ll}
 & \mathsf{foldr}\;g \subseteq id^{\mathsf{T}\,h} \cdot \mathsf{foldr}\;(g \cdot h) \\
\equiv & \qquad \{\ (4.13)\ \} \\
 & \mathsf{foldr}\;g \subseteq (\cdot\mathsf{T}\;h)^{\circ} \cdot \mathsf{foldr}\;(g \cdot h) \\
\equiv & \qquad \{\ \text{shunting } (3.3);\ (3.5)\ \} \\
 & (\cdot\mathsf{T}\;h) \cdot \mathsf{foldr}\;g = \mathsf{foldr}\;(g \cdot h) \\
\equiv & \qquad \{\ \text{go pointwise on } e\ \} \\
 & \mathsf{foldr}\;g\;e \cdot \mathsf{T}\;h = \mathsf{foldr}\;(g \cdot h)\;e
\end{array} \tag{7.9}$$

Law (7.9) is often referred to as the *absorption law* of “fmap” ($\mathsf{T}\;h$) by foldr. It is another corollary of free theorem (7.6) and therefore none of its instances in the *Foldable* class needs a proof. The default implementation of *foldMap* in Data.Foldable arises from (7.9).

Maybe. Just to give an example of (7.8) holding for instances of class *Foldable* other than lists, let us check the *Maybe* instance defined in *Data.Foldable*, with respect to (7.7):

```
instance Foldable Maybe where
  foldMap = maybe mempty
  foldr f z Nothing = z
  foldr f z (Just x) = f x z
  foldl f z Nothing = z
  foldl f z (Just x) = f z x
```

We have two cases: for $xs := \mathsf{Nothing}$, the right square trivially reduces to $h\,e = h\,e$ and we are done. In the case $xs := \mathsf{Just}\,x$, we have $\mathsf{foldr}\;f\;\;e\;(\mathsf{Just}\,x) = f\,x\,e$ in (7.8) and $\mathsf{foldr}\;\;g\;\;(h\,e)\,(\mathsf{Just}\,x) = g\,x\,(h\,e)$. Then: $g\,x\,(h\,y) = h\,(f\,x\,y) \Rightarrow h\,(f\,x\,e) = g\,x\,(h\,e)$ trivially holds too.

Permutativity. Let f and g be the same function in (7.7), say s, and $h := s\,a$:

$$(s\,x) \cdot (s\,a) = (s\,a) \cdot (s\,x) \quad\Rightarrow\quad \mathsf{foldr}\;s \cdot (s\,a) = (s\,a\cdot) \cdot \mathsf{foldr}\;s \tag{7.10}$$

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

Property $(s\,x)\cdot(s\,a) = (s\,a)\cdot(s\,x)$ — i.e. the fully pointwise $s\,x\,(s\,a\,y) = s\,a\,(s\,x\,y)$ — is called (left) *permutativity* by Danvy (2023).[16] It can be easily shown that if s is *associative* and *commutative* then it is *permutative*:

$$
\begin{aligned}
& s\,a\,(s\,x\,y) \\
= & \quad \{\ \text{associative}\ \} \\
& s\,(s\,a\,x)\,y \\
= & \quad \{\ \text{commutative}\ \} \\
& s\,(s\,x\,a)\,y \\
= & \quad \{\ \text{associative}\ \} \\
& s\,x\,(s\,a\,y)
\end{aligned}
$$

That is:

$$s\,a\,(s\,x\,y) = s\,x\,(s\,a\,y) \quad \Leftarrow \quad s \text{ is associative and commutative} \tag{7.11}$$

On the other hand, if s is *permutative* and has unit e, then s is *commutative*: $s\,x\,(s\,a\,e) = s\,a\,(s\,x\,e)$ and thus $s\,x\,a = s\,a\,x$.

8 Universal properties

Suppose a particular instance T of *Foldable* such that

$$\mathsf{foldr}\ \alpha\ \gamma = id \tag{8.1}$$

holds, for some α and γ. By (7.6) the types are $\alpha : A \to \mathsf{T}\,A \to \mathsf{T}\,A$ and $\gamma : \mathsf{T}\,A$. That is, α and γ are constructors of type $\mathsf{T}\,A$. Such is the case of lists, in which $\mathsf{foldr}\ (:)\ [\,] = id$, that is, $\alpha = (:)$ and $\gamma = [\,]$ in (8.1). Then one has the following corollary of (7.8):

$$g\,x\,(h\,y) = h\,(\alpha\,x\,y) \quad \Rightarrow \quad h\,xs = \mathsf{foldr}\ g\ (h\,\gamma)\ xs$$

which, by introducing $z = h\,\gamma$ and dropping xs, becomes:

$$\begin{cases} h\,\gamma = z \\ h\,(\alpha\,x\,y) = g\,x\,(h\,y) \end{cases} \quad \Rightarrow \quad h = \mathsf{foldr}\ g\ z \tag{8.2}$$

This means that, if γ and α exist such that (8.1) holds, by the free-theorem of foldr the system of equations (in h)

$$\begin{cases} h\,\gamma = z \\ h\,(\alpha\,x\,y) = g\,x\,(h\,y) \end{cases}$$

[16] Left permutativity is also called *left-commutativity* in the literature, see e.g. Schropp & Popescu (2013).

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

has a unique solution, $h = \mathsf{foldr}\ g\ z$. Since $\mathsf{foldr}\ g\ z$ satisfies the equations for which it is a solution, then

$$\begin{cases} \mathsf{foldr}\ g\ z\ \gamma = z \\ \mathsf{foldr}\ g\ z\ (\alpha\ x\ y) = g\ x\ (\mathsf{foldr}\ g\ z\ y) \end{cases} \tag{8.3}$$

hold, unveiling the definition of foldr itself. Moreover, this definition is mathematically equivalent to (just replace h by $\mathsf{foldr}\ g\ z$ and simplify):

$$h = \mathsf{foldr}\ g\ z \quad \Rightarrow \quad \begin{cases} h\ \gamma = z \\ h\ (\alpha\ x\ y) = g\ x\ (h\ y) \end{cases} \tag{8.4}$$

Putting (8.2,8.4) together, we get the **universal property** of foldr,

$$h = \mathsf{foldr}\ g\ z \quad \equiv \quad \begin{cases} h\ \gamma = z \\ h\ (\alpha\ x\ xs) = g\ x\ (h\ xs) \end{cases} \tag{8.5}$$

which, for lists, is

$$h = \mathsf{foldr}\ g\ z \quad \equiv \quad \begin{cases} h\ [\,] = z \\ h\ (x : xs) = g\ x\ (h\ xs) \end{cases} \tag{8.6}$$

cf. e.g. Hutton (1999).

9 Is foldl equal to foldr?

Looking at (7.1), the *type-wise distance* between foldr and foldl is the flip (6.3) of the first parameter.[17] So the "best fit" one can aim at here is

$$\mathsf{foldl}\ f \ \stackrel{?}{=}\ \mathsf{foldr}\ \widetilde{f} \tag{9.1}$$

possibly valid for (as wide as possible) a class of functions f and instances of class *Foldable*.

Our first step is to conjecture a definition for foldl that is type-wise consistent (over the *Foldable* class) with (8.3) which — recall — was derived from the free-theorem of foldr. A possibility is to involve foldr itself in the definition. Looking at the flipped f in (9.1), which is of type $A \to B^B$, one can think of expressing $\widetilde{\mathsf{foldl}\ f} : \mathsf{T}\ A \to B^B$ by a higher-order fold:

$$\begin{aligned} &\widetilde{\mathsf{foldl}\ f} = \mathsf{foldr}\ (\theta\ f)\ id \\ &\qquad \textbf{where}\ (\theta\ f)\ a\ g = g \cdot (\widetilde{f}\ a) \end{aligned} \tag{9.2}$$

Note the type $\theta\ f : A \to (B^B)^{(B^B)}$ for $f : B \to B^A$.

[17] Danvy (2023), who gives a brief history of folding left and right over lists, is somewhat critical about this mismatch of types between the two fold-combinators.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

Next, we unfold (9.2) via universal property (8.5). For easier reference, we instantiate (with no loss of generality) γ and α for lists, as these are very well known to functional programmers:

$$
\begin{array}{cl}
 & \widetilde{\mathsf{foldl}\, f} = \mathsf{foldr}\;(\theta\, f)\; id \\
\equiv & \qquad \{\ (8.5)\ \} \\
 & \left\{ \begin{array}{l} \widetilde{\mathsf{foldl}\, f}\;[\,] = id \\ \widetilde{\mathsf{foldl}\, f}\;(x : xs) = (\theta\, f)\; x\; (\widetilde{\mathsf{foldl}\, f}\; xs) \end{array} \right. \\
\equiv & \qquad \{\ (9.2)\ \} \\
 & \left\{ \begin{array}{l} \widetilde{\mathsf{foldl}\, f}\;[\,] = id \\ \widetilde{\mathsf{foldl}\, f}\;(x : xs) = (\widetilde{\mathsf{foldl}\, f}\; xs) \cdot (\widetilde{f}\; x) \end{array} \right. \\
\equiv & \qquad \{\ \text{go pointwise on } z \text{ and unfold the flips}\ \} \\
 & \left\{ \begin{array}{l} \mathsf{foldl}\, f\; z\; [\,] = z \\ \mathsf{foldl}\, f\; z\; (x : xs) = \mathsf{foldl}\, f\; (f\; z\; x)\; xs \end{array} \right.
\end{array}
$$

This confirms the standard definition of foldl for lists.

Universal-foldl. An advantage of defining foldl *"as a* foldr*"* (9.2) is that the universal property of the latter induces the universal property of the former:[18]

$$
\begin{array}{cl}
 & k = \mathsf{foldl}\, f \\
\equiv & \qquad \{\ (9.2)\ ;\ \text{isomorphism}\ \mathit{flip_}\ \} \\
 & \widetilde{k} = \mathsf{foldr}\;(\theta\, f)\; id \\
\equiv & \qquad \{\ \text{universal-}\mathsf{foldr}\ (8.6)\ \} \\
 & \left\{ \begin{array}{l} \widetilde{k}\;[\,] = id \\ \widetilde{k}\;(h : t) = (\theta\, f)\; h\; (\widetilde{k}\; t) \end{array} \right. \\
\equiv & \qquad \{\ \text{introduce } z \text{ and flip}\ \} \\
 & \left\{ \begin{array}{l} k\; z\; [\,] = z \\ k\; z\; (h : t) = (\theta\, f)\; h\; (\widetilde{k}\; t)\; z \end{array} \right. \\
\equiv & \qquad \{\ (\theta\, f)\, x\, g = g \cdot (\widetilde{f}\; x)\ (9.2)\ \} \\
 & \left\{ \begin{array}{l} k\; z\; [\,] = z \\ k\; z\; (h : t) = \widetilde{k}\; t\; (f\; z\; h) \end{array} \right. \\
\equiv & \qquad \{\ \text{flipping}\ \} \\
 & \left\{ \begin{array}{l} k\; z\; [\,] = z \\ k\; z\; (h : t) = k\; (f\; z\; h)\; t \end{array} \right.
\end{array}
$$

[18] Again we stay with lists for easier reference.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

Thus we get the universal-property of foldl:

$$k = \mathsf{foldl}\, f \quad \equiv \quad \left\{ \begin{array}{l} k\, z\, [\,] = z \\ k\, z\, (h : t) = k\, (f\, z\, h)\, t \end{array} \right. \tag{9.3}$$

Equating foldl **and** foldr. Next we address the question (9.1) that opened this section: under what conditions does $\mathsf{foldl}\, f = \mathsf{foldr}\, \widetilde{f}$ hold? We can use foldl-universal (9.3) to find an answer:

$$\begin{array}{ll}
& \mathsf{foldl}\, f = \mathsf{foldr}\, \widetilde{f} \\
\equiv & \qquad \{\ (9.3)\ \} \\
& \left\{ \begin{array}{l} \mathsf{foldr}\, \widetilde{f}\, z\, [\,] = z \\ \mathsf{foldr}\, \widetilde{f}\, z\, (h : t) = \mathsf{foldr}\, \widetilde{f}\, (f\, z\, h)\, t \end{array} \right. \\
\equiv & \qquad \{\ \text{flipping}\ f\, z\, h\ \} \\
& \left\{ \begin{array}{l} \mathsf{foldr}\, \widetilde{f}\, z\, [\,] = z \\ \mathsf{foldr}\, \widetilde{f}\, z\, (h : t) = \mathsf{foldr}\, \widetilde{f}\, (\widetilde{f}\, h\, z)\, t \end{array} \right. \\
\equiv & \qquad \{\ \text{resort to (7.10) by assuming permutativity: } (\widetilde{f}\, x) \cdot (\widetilde{f}\, a) = (\widetilde{f}\, a) \cdot (\widetilde{f}\, x)\ \} \\
& \left\{ \begin{array}{l} \mathsf{foldr}\, \widetilde{f}\, z\, [\,] = z \\ \mathsf{foldr}\, \widetilde{f}\, z\, (h : t) = \widetilde{f}\, h\, (\mathsf{foldr}\, \widetilde{f}\, z\, t) \end{array} \right. \\
\equiv & \qquad \{\ (8.3)\ \} \\
& \text{TRUE}
\end{array}$$

We conclude that $\mathsf{foldl}\, f = \mathsf{foldr}\, \widetilde{f}$ holds for the instances of class *Foldable* such that (8.1) holds, provided that $\widetilde{f}$ is a *permutative* operation (recall Section 7).

The usual assumption that $\mathsf{foldl}\, f\ e$ and $\mathsf{foldr}\, \widetilde{f}\ e$ are the same for f associative and e its unit[19] is therefore too strong. By (7.11) we know that associativity and commutativity ensure permutativity.[20] However, the converse implication does not hold, take e.g. $f = (\div)$ (5.9) as counter-example: neither $(\div)$ nor $\widetilde{(\div)}$ are associative or commutative, and yet $\widetilde{(\div)}$ is permutative. Why? Let us check it by indirect equality (Dijkstra, 2001):

$$\begin{array}{ll}
& y \leqslant \widetilde{f}\, a\, (\widetilde{f}\, b\, x) \\
\equiv & \qquad \{\ f\, x\, y = x \div y \text{ in this case }\ \} \\
& y \leqslant (x \div b) \div a \\
\equiv & \qquad \{\ \text{Galois connection (5.9) twice }\ \}
\end{array}$$

[19] See e.g. exercise 1.10 of Bird & Gibbons (2020).
[20] Also easy to show is that flipping preserves commutativity and associativity.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

$$
\begin{array}{ll}
& (y \times a) \times b \leqslant x \\
\equiv & \quad \{ \ (\times) \text{ is associative and commutative } \} \\
& (y \times b) \times a \leqslant x \\
\equiv & \quad \{ \ \text{Galois connection (5.9) twice in the opposite direction } \} \\
& y \leqslant (x \div a) \div b \\
\equiv & \quad \{ \ f\,x\,y = x \div y \text{ in this case } \} \\
& y \leqslant \widetilde{f}\,b\,(\widetilde{f}\,a\,x) \\
:: & \quad \{ \ \text{by indirect equality (Dijkstra, 2001) } \} \\
& \widetilde{f}\,a\,(\widetilde{f}\,b\,x) = \widetilde{f}\,b\,(\widetilde{f}\,a\,x)
\end{array}
$$

Also to be noted, the second parameter e remains unconstrained in both folds. For example:

$$
\begin{array}{l}
\mathsf{foldl}\ (\div)\ 100,000\ [99,2,7] = 72 = \mathsf{foldr}\ (\widetilde{\div})\ 100,000\ [99,2,7] \\
\mathsf{foldl}\ (\div)\ 10,000\ [99,2,7] = 7 = \mathsf{foldr}\ (\widetilde{\div})\ 10,000\ [99,2,7]
\end{array}
$$

Of course, the previous calculation can generalize to any Galois connection $f \dashv g$. Altogether, we have that $\mathsf{foldr}\ \widetilde{g} = \mathsf{foldl}\ g$ is granted for those g that, being neither associative nor commutative, participate in an adjunction $f \dashv g$ whose other adjoint is so.

10 Summary and discussion

This article addresses the use of relational, pointfree techniques in reasoning about functional programs. In particular, it shows the role of relational semi-commutative squares (suggestively referred to as *"magic"* squares) in expressing (and reasoning about) formal concepts relevant to computing and, in particular, to FP. By switching to *relational types* one gets a quite rich setting in which programming and expressing properties of programs blend naturally.[21] The calculation of "theorems-for-free" is given as an application of this, in particular used to infer conditions for the combinators $\mathsf{foldl}\ f\ e$ and $\mathsf{foldr}\ \widetilde{f}\ e$ to compute the same output for any e.

This research shares much with the work of Backhouse & Backhouse (2004), who give perhaps the most impressive account of the power of free-theorems of all literature on the subject, contributing with sharp results on the role of Galois connections in (higher-order) *abstract interpretation*. Such a paper is, however, not an easy read for the average functional programmer. The (less ambitious) approach proposed in the current paper bears in mind the need to make such fantastic results more and more accessible to the programming community.

There is a sharp contrast between the effectiveness of relational reasoning in the style of e.g. Bird & de Moor (1997) and Backhouse & Backhouse (2004) and the low popularity of

[21] More examples could have been given of such squares and relational types, for instance order-preserving multifunctions (Smithson, 1971), i.e. isotone relations (Walker, 1984), and metamorphic relations, which are at the core of metamorphic testing (Zhou *et al*., 2020).

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

the methodology. This is rather unfortunate. Having decided to teach such relational methods to computer science students two decades ago, the author has invested into widening scope and applying such a reasoning style to areas such as database programming, formal modeling and, of course, functional programming (Oliveira, 2024). In this process, diagrams have proved very effective as a device enhancing the perception of the information flow and of how the overall reasoning is conducted. Thus *magic squares* (MS) emerged as a simple, yet very powerful *design unit* of a pointfree, relational approach to formal reasoning.

Relational exponentiation S^R plays a major role in the way higher-order functions are handled. By MS vertical composition (2.4), one immediately infers that, as expected, it is monotone on the base and anti-monotonic on the exponent:

$$\begin{cases} R' \subseteq R \\ S \subseteq S' \end{cases} \implies S^R \subseteq S'^{R'}$$

We also know that $id^{id} = id$ (4.14). By horizontal composition (2.3), we get

$$S^R \cdot S'^{R'} \subseteq (S \cdot S')^{(R \cdot R')} \tag{10.1}$$

However, the converse inclusion does not hold and so relational exponentiation is not in general a (bi)relator.[22] Backhouse & Backhouse (2004) give conditions for strengthening (10.1) to an equality that include the cases involving functions and converses of functions used in this paper.

Functional programmers have realized that many abstract concepts they love have a categorical nature. This has led to a strong focus on explaining the whole paradigm in categorial terms, see e.g. Hinze *et al.* (2015) — a trend that of course includes the today so important concept of a *monad* (Wadler, 1990; Moggi, 1991; Gibbons & Hinze, 2011). It all happened as if, in a sense, FP eventually came to rescue category theory from being considered by the programming community as mere *abstract nonsense*, with no practical application. In this setting, theorems-for-free have been formalized via (lax) *dinatural transformations* (Hackett & Hutton, 2015; Voigtländer, 2019) in order-enriched categories. We believe that the category of relations, which is "naturally" ordered by relation inclusion, provides an overall simpler approach, much in the spirit of the quote by Freyd & Scedrov (1990) that opens this paper.

Framed in this trend, the foldl / foldr case study could have been given as a pointfree calculation carried out via the *adjoint-fold theorem* (Hinze, 2013; Oliveira, 2023) and enabled by the self-adjunction witnessed by *flip*[23] — as in a similar calculation concerning left and right-iteration (Oliveira, 2020). It should be noted that knowing that permutativity is enough for foldr/foldl "equality" is not new — see e.g. Danvy (2023). Danvy's reasoning is, however, quite different: he postulates permutativity as a side condition and then proves it in Coq by induction on lists. In this paper, permutativity arises

[22] In a sense, relational exponentiation can be regarded as a "lax (bi)relator".
[23] It must be said that this is where (9.2) comes from.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

by free-theorem calculation. Moreover, it shows that Danvy's result can be extended via Galois connections.[24]

The generic approach to this case study given in the current paper throws attention to the *Foldable* class of the Haskell standard library system. The permutativity of f that grants the $\mathsf{foldl}\, f = \mathsf{foldr}\, \widetilde{f}$ outcome "for-free" is not alone: a *reflection constraint* (8.1) is also needed. This constraint is quite strong in the sense of identifying the type constructors γ and α of the particular *Foldable* instance in hands. The types of α and γ in (8.1) clearly point to lists as being the *prototypical* instance of the class.

It is not difficult to find instances of class *Foldable* that do not meet reflection constraint (8.1) and for which the foldl / foldr result cannot be obtained as in the current paper. Interestingly enough, $\mathsf{foldl}\, f\ e = \mathsf{foldr}\, \widetilde{f}\ e$ seems to be granted across the class as an axiom, that is, $\mathsf{foldl}\, f\ e = \mathsf{foldr}\, \widetilde{f}\ e$ is the *default* definition of foldl, assuming foldr defined. Studying this better and applying the same kind of reasoning to other classes of the Haskell standard libraries is a topic for future research.

Acknowledgments

This work was supported by National Funds through the *FCT – Fundação para a Ciência e a Tecnologia*, I.P. (Portuguese Foundation for Science and Technology) within the IBEX project, with reference PTDC/CCI-COM/4280/2021.

Competing interests

The author reports no conflict of interest.

References

Backhouse, K. & Backhouse, R. C. (2004) Safety of abstract interpretations for free, via logical relations and Galois connections. *SCP* **15**(1–2), 153–196.

Backhouse, R. C. (1990) On a relation on functions. In *Beauty is our Business: A Birthday Salute to Edsger W. Dijkstra*. New York, NY, USA: Springer-Verlag, pp. 7–18.

Baquero, C., Almeida, P. S. & Shoker, A. (2014) Making operation-based CRDTs operation-based. In *Distributed Applications and Interoperable Systems*, Magoutis, K. & Pietzuch, P. (eds), Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 126–140.

Barbosa, L. S., Oliveira, J. N. & Silva, A. M. (2008) Calculating invariants as coreflexive bisimulations. In AMAST'08, LNCS, vol. 5140. Springer-Verlag, pp. 83–99.

Bird, R. & de Moor, O. (1997) *Algebra of Programming*. Prentice-Hall. ISBN: 978-0-13-507245-5.

Bird, R. & Gibbons, J. (2020) *Algorithm Design with Haskell*. Cambridge University.

Damas, L. & Milner, R. (1982) Principal type-schemes for functional programs. In Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'82. New York, NY, USA: ACM, pp. 207–212. ISBN 0-89791-065-6.

Danvy, O. (2023) Folding left and right matters: Direct style, accumulators, and continuations. *J. Funct. Program.* **33**, e2.

[24] It is worth studying permutativity in its own right, as it seems to play a role also in other research areas, for instance replicated datatypes (CRDTs) (Baquero *et al.*, 2014) and protocols such as the Diffie-Hellman key exchange (Merkle, 1978).

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

de Bakker, J. & de Roever, W. P. *A Calculus for Recursive Program Schemes*. Stichting Mathematisch Centrum Tec. Report 131/72, Amsterdam. https://ir.cwi.nl/pub/9145, January 1972.

Dijkstra, E. W. (2001) *Indirect Equality Enriched*. Technical note EWD 1315-0.

Freyd, P. J. & Scedrov, A. (1990) *Categories, Allegories*, Mathematical Library, vol. 39. North-Holland. ISBN: 9780444703682.

Gibbons, J. & Hinze, R. (2011) Just do it: Simple monadic equational reasoning. In Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP'11. New York, NY, USA: ACM, pp. 2–14.

Hackett, J. & Hutton, G. (2015) Programs for cheap! In LICS 2015. IEEE Computer Society, pp. 115–126.

Hinze, R. (2013) Adjoint folds and unfolds — an extended study. *SCP* **78**(11), 2108–2159.

Hinze, R., Wu, N. & Gibbons, J. (2015) Conjugate hylomorphisms – or: The mother of all structured recursion schemes. In POPL'15. New York, NY, USA: ACM, pp. 527–538.

Hutton, G. (1999) A tutorial on the universality and expressiveness of fold. *J. Funct. Program.* **9**(4), 355–372.

Lämmel, R. (2008) Google's MapReduce programming model - Revisited. *Sci. Comput. Program.* **70**(1), 1–30.

Merkle, R. C. (1978) Secure communications over insecure channels. *Commun. ACM* **21**(4), 294–299.

Moggi, E. (1991) Notions of computation and monads. *Inf. Comput.* **93**(1), 55–92.

Mu, S.-C. & Oliveira, J. N. (2011) Programming from Galois connections. In *RAMiCS*, de Swart, H. (ed.), LNCS, vol. 6663, pp. 294–313.

Neri, A., Barbosa, R. S. & Oliveira, J. N. (2022) Compiling quantamorphisms for the IBM Q experience. *IEEE Trans. Software Eng.* **48**(11), 4339–4356.

Oliveira, J. N. (2009) Extended static checking by calculation using the pointfree transform. LNCS, vol. 5520. Springer-Verlag, pp. 195–251.

Oliveira, J. N. (2020) *A Note on the Under-Appreciated for-Loop*. Technical Report TR-HASLab:01:2020 (PDF), HASLab/U.Minho and INESC TEC.

Oliveira, J. N. (2023) Why adjunctions matter—a functional programmer perspective. In WADT'22, Madeira, A. & Martins, M. A. (eds), LNCS, pp. 25–59.

Oliveira, J. N. (2024) Program Design by Calculation. Unpublished book draft, Sep. 2024. Informatics Dept., U.Minho (PDF).

Oliveira, J. N. and Ferreira, M. A. (2013) Alloy meets the algebra of programming: A case study. *IEEE Trans. Soft. Eng.* **39**(3), 305–326.

Plotkin, G., Power, J., Sannella, D. & Tennent, R. (2000) Lax logical relations. In *Automata, Languages and Programming*, Montanari, U., Rolim, J. D. P. & Welzl, E. (eds). Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 85–102.

Reynolds, J. C. (1983) Types, abstraction and parametric polymorphism. *Inf. Process.*, **83**, 513–523.

Sangiorgi, D. (2004) Bisimulation: From the origins to today. In LICS. IEEE Computer Society, pp. 298–302.

Schropp, A. & Popescu, A. (2013) Nonfree datatypes in Isabelle/HOL. In *Certified Programs and Proofs*, Gonthier, G. & Norrish, M. (eds). Springer International Publishing, pp. 114–130. ISBN 978-3-319-03545-1.

Smithson, R. E. (1971) Fixed points of order preserving multifunctions. *Proc. Am. Math. Soc.* **28** (1), 304–310.

Voigtländer, J. (2009) Bidirectionalization for free! (Pearl). In POPL 2009, Shao, Z. & Pierce, B. C. (eds). ACM, pp. 165–176.

Voigtländer, J. (2019) Free theorems simply, via dinaturality. arXiv cs.PL 1908.07776.

von Karger, B. (1998) Temporal algebra. *Math. Struct. Comput. Sci.* **8**(3), 277–320.

Wadler, P. L. (1989) Theorems for free! In 4th International Symposium on Functional Programming Languages and Computer Architecture, London. ACM, pp. 347–359.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012

Wadler, P. L. (1990) Comprehending monads. In Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, Nice, France.

Wadler, P. L. (2015) Propositions as types. *Commun. ACM* **58**(12), 75–84.

Walker, J. W. (1984) Isotone relations and the fixed point property for posets. *Discrete Math.* **48**(2), 275–288. ISSN 0012-365X.

Zhou, Z. Q., Sun, L., Chen, T. Y. & Towey, D. (2020) Metamorphic relations for enhancing system understanding and use. *IEEE Trans. Softw. Eng.* **46**(10), 1120–1154.

Downloaded from https://www.cambridge.org/core. IP address: 3.17.179.240, on 03 Mar 2025 at 23:25:27, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/S0956796825000012