Functional Programming with Relations*

Graham Hutton
Department of Computing Science
Glasgow University, Scotland

December 1990

Abstract

While programming in a relational framework has much to offer over the functional style in terms of expressiveness, computing with relations is less efficient, and more semantically troublesome. In this paper we propose a novel blend of the functional and relational styles. We identify a class of *causal relations*, which inherit some of the bi–directionality properties of relations, but retain the efficiency and semantic foundations of the functional style.

1 Introduction

In his ACM Turing Award Lecture, Backus presented a new style of programming, in which programs are built piecewise by combining smaller programs [Backus78]. In [Sheeran83], Sheeran showed how the same approach could be used to good effect in VLSI design. In keeping with the special constraints of hardware, many designs have a regular structure, with components communicating, often bi–directionally, only with their immediate neighbours. With function composition as the main combining form however, circuits with bi–directional data flow patterns tend to have rather contorted descriptions in the functional style. This problem lead Sheeran to use binary relations rather than functions as the underlying model of circuitry, thereby removing the distinction between input and output which causes the problem in the first place. An overview of the relational language *Ruby* is presented in Section 2.

Unfortunately, the many benefits of relations do not come for free. While relations have much to offer over functions for specification and refinement of a program (see Section 2.4), when it comes to execution time, functions are clear winners. In particular, with no inherent notion of data–flow, computing with relations is generally speaking much less efficient than computing with functions. Furthermore, relational languages are not so semantically well behaved as their functional counterparts. For example, it is known that the standard fixed–point approach to recursion does not naturally extend to the relational world.

In this paper, we propose a novel blend of the functional and relational styles. In Section 3, we identify a class of functional or *causal* relations. Informally speaking,

*Appears as [Hutton90]. Authors e–mail address: `graham@dcs.glasgow.ac.uk`.

a relation is causal if we can identify an "input" part of the relation, which uniquely determines the remaining "output" part. Unlike functions however, the input part of a causal relation is not restricted to its domain, nor output part to the range; indeed, inputs and outputs may be interleaved throughout the domain and range. Furthermore, a causal relation may have many such functional interpretations. The intention is that a causal language brings some of the expressive power of a truly relational language, without incurring semantic and implementation problems. It is clear that many hardware style programs fall naturally into the causal class.

When two causal relations are composed, it is reasonable to expect that inputs on one side join with outputs on the other, and vice–versa. In Section 3.2 we find that this (and one other) restriction is in fact necessary to ensure that causal relations have the expected properties. Finally, in Section 4 we describe a simple system in which we may capture the functional ways in which a causal program may be used.

2 Programming in Hardware

In this section, we present an overview of the Ruby style of relational programming. The language is developed incrementally, beginning with a functional framework in 2.1, introducing streams in 2.2, structural recursion in 2.3, and finally, moving to relations in 2.4. Ruby itself is explained more fully in [Jones90b], where it is used in the stepwise derivation of many interesting hardware style programs.

2.1 Constructive programming

In the Miranda[1] style of functional programming, functions are commonly defined using abstraction (e.g. $f\,x = x+1$ defines a function which increments a number). In the FP [Backus78] style however, functions are built indirectly by combining other functions in various ways. Examples of such constructions include composition, defined by $f \circ g \;\widehat{=}\; \lambda x.\; f\,(g\;x)$, and product, defined by $f \times g \;\widehat{=}\; \lambda(x,y).\; (f\;x, g\;y)$. Later on in this section, concerns with the shape of a program lead us to use backward composition "$;$", defined by $f\,;g \;\widehat{=}\; g \circ f$, rather than the more usual forward composition "$\circ$".

Languages in which larger programs are built piecewise from smaller ones are sometimes known as *constructive* or *combinatory* languages. Similarly, the higher–order functions from which programs are built are often referred to as *combining forms*, or *combinators*[2] for short. Appart from FP itself, perhaps the most well known example of the constructive paradigm is the Bird–Meertens formalism [Bird88], also known as the "theory of lists".

It is well known that programming in the constructive style is a great aid to formal manipulation. In particular, the combining forms from which programs are built satisfy many useful algebraic laws, which can be used to derive and prove properties of programs. For example, it is easy to show that $(A\,;B) \times (C\,;D) =$

[1]Miranda is a Trademark of Research Software Ltd.

[2]Our informal use of this term is consistent with the standard λ–calculus meaning.

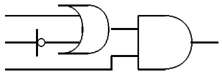

Figure 1: snd $(\mathsf{fst}\ \mathsf{not}\,;\mathsf{or})\,;\mathsf{and}$

$(A \times C)\,;(B \times D)$. In fact, many such theorems do not actually require an explicit proof, their validity following from the types of the combining forms, under the observation that (parametric) polymorphism corresponds precisely to the notion of *naturality* in Category Theory. The idea that polymorphic type inference derives "free theorems" is developed in [Wadler89], and applied in the specific context of Ruby in [Sheeran89].

In the constructive style, it is quite natural to consider the *shape* of a program; the combining forms have both a behavioural and a pictorial interpretation [Sheeran81]. An example picture is given in Figure 1, with respect to the standard abbreviations $\mathsf{fst}\ F \ \widehat{=}\ F \times \mathsf{id}$ and $\mathsf{snd}\ F \ \widehat{=}\ \mathsf{id} \times F$. The identity for composition "id" corresponds to a notional wire in pictorial terms. Because information flow is most often from left to right in pictures, forwards composition "$;$" is used rather than backwards composition "$\circ$" in Ruby. Generalising from pictures, it is often useful to view a constructive program as a description of a process network or data flow graph, with primitive components communicating over channels.

2.2 Recursion and streams

Consider the factorial function. In a language like Miranda, where functions are most often built using abstraction, it may be defined recursively by:

$$\begin{array}{lcl} \mathit{fac}\quad 0 & = & 1 \\ \mathit{fac}\quad (n+1) & = & (n+1)\ *\ \mathit{fac}\ n \end{array}$$

In the constructive language FP, the corresponding program is:

$$\mathit{fac} \ = \ \mathit{eq}0 \rightarrow \overline{1}\ ;\ * \circ [\mathit{id}, \mathit{fac} \circ \mathit{sub}1]$$

where $\mathit{eq}0 = \mathit{eq} \circ [\mathit{id}, \overline{0}]$ and $\mathit{sub}1 = - \circ [\mathit{id}, \overline{1}]$. There is a however a problem with this style of recursion in our context. In FP, we may think of a program as the description of a *dynamic* network, in the sense that components may be freely created and destroyed at run–time. More formally (and in terms of pictures), the factorial program denotes an infinite network, corresponding to an infinite unwinding of the recursion.

While dynamic networks are fine for general purpose programming, in hardware we are restricted to entirely *static* networks. The reason is quite simple, the network is fixed once and for all when a circuit is fabricated on silicon. Components are not able to move around, duplicate or destroy themselves; explicit circuitry must be included for all eventualities which may arise.

In terms of programming style, the restriction to static networks means that *data dependent* recursion as used in the factorial example is not acceptable in our language. This restriction is the primary difference between "programming in hardware" and "programming in software".

Although infinite networks are not acceptable in a static language, there is nothing to stop us using *cyclic* networks. Consider a simple cyclic network – an and operator with its upper input driven from its own output. This may be cast as $\mathsf{loop}\ (\mathsf{and}\ ;\ \mathsf{split})$ in Ruby notation. Assuming that all primitives are functional, it is clear that this program will deadlock, since the output is directly dependent upon itself. More formally, the constructive expression is equivalent to the recursive function $f = \lambda x.\,(x \wedge fx)$ in λ–style, which under least fixed-point semantics (and assuming strict conjunction) is in turn equivalent to $\lambda x.\bot$.

To allow programs involving cyclic networks to terminate, a notation of time is introduced into the language, through the use of *streams*. A stream has the same behaviour as a lazy list (i.e. only a prefix need be evaluated at any instant to allow computation to proceed), except that its elements are normally accessed by subscript, rather than by structural decomposition. A stream may be formally viewed as a mapping, with the natural numbers (representing time) as the domain.

To avoid deadlock in feedback programs, the first step is to lift all the primitives to the stream level, so that they operate pointwise over the components. For example, an and operator will now take a stream of booleans to a stream of their conjunctions. Now all we need do is ensure that at any moment, the output of feedback programs depends only upon their own value at strictly earlier instances in time. This is achieved by the introduction of a single sequential primitive, a unit delay, which returns its input value at time t as its output at time $t + 1$. Rather than filling the gap in the output at $t = 0$ with an undefined or fixed value such as $\bot$, we write D_s for a delay element whose first output is s. This gives us a direct form of control over the start–up phase in feedback programs.

Introducing a delay into our feedback example results in little change in the text of the constructive program, which now takes the form $\mathsf{loop}\ (\mathsf{snd}\ \mathsf{D}_\mathsf{T}\ ;\ \mathsf{and}\ ;\ \mathsf{split})$, where T denotes boolean *true*. However, the jump to streams results in a marked change to the analogous λ–style expression:

$$f\ x = \lambda t. \begin{cases} x_0 & \mathsf{if}\ t = 0 \\ x_t \wedge f\ x\ (t-1) & \mathsf{otherwise} \end{cases}$$

where x and the result of the function are streams of (i.e. mappings to) booleans. Note that T was carefully chosen as the starting value for the delay, such that $x_0 \wedge \mathsf{T}$ simplifies to x_0 in the base case. Assuming strict conjunction again, this function clearly has least fixed point $\lambda x.\,(\lambda t.\ \bigwedge_{i=0}^{t} x_i)$. Thus, through the introduction of streams, and hence the delay primitive, the feedback program which previously denoted a non–terminating function now denotes a well defined boolean function which holds T until its input drops to F, after which it remains F for ever more, independent of subsequent input values.

From the programmer's point of view, the presence of streams means that combinational (time independent) and sequential (clocked) design is cast in a uniform framework. In particular, we can use the sequential primitive D as a buffer between combinational components, thereby introducing pipeline parallelism into our programs, in addition to the explicit parallelism of the "$\times$" combining form.

2.3 Generic primitives

Many interesting hardware style (i.e. static) algorithms have a regular structure. For example, correlation (one of the most important signal processing algorithms) may be cast as a 2–dimensional array of simple binary components [Jones90b]. Since our language at present is itself entirely static, we would have to choose a particular size, and build such a grid explicitly using combining forms which stack components beside and above one another. While this approach would allow us to experiment with regular programs, it is far from satisfactory. Having multiple occurrences of the same combining forms and primitive cells would certainly hinder transformation and proofs. Furthermore, there is a danger in reasoning about fixed sized arrays; a theorem which holds for a particular instance (e.g. 3×3) may extend to some cases (e.g. odd–sized arrays), but not to the general case.

If the constructive style is to be practical, we must introduce some means to capture and manipulate regular structures, without reference to any particular size. How is this possible if all our programs must denote static networks? The key is to note that they need only be static at run–time, when the program is actually executed. It is perfectly acceptable to have dynamic primitives at compile–time, so long as we ensure that all programs are indeed of a fixed size when they come to be executed. At present, all our primitives are entirely static at compile–time.

A *generic* combining form denotes a family of fixed–sized networks, one for each instance of a regular pattern. For example, map in Ruby is a generalisation of the product construction "$\times$", such that ($\mathsf{map}\ R$) denotes the infinite famililly $\{R^n \mid n \in \mathsf{N}\}$, where R^n stands for the n–way product of R with itself. Generic plumbing such as zip (which interleaves two lists) are used to route data between generic components. In fact, inverse plumbing such as unzip is also useful. While generic plumbing and their inverse would normally have to be defined separately, in a relational language such as Ruby (see next section), one may be defined directly in terms of the other using relational inverse.

To ensure that generic programs are static at run–time, we introduce a new stage between compilation and execution. At *silicon–time*, the programmer will choose a particular size for the program, such that it may then be physically expanded to a fixed sized network. Thus, while $R \times R$ is static at compile–time, ($\mathsf{map}\ R$) cannot be guaranteed to be static until run–time.

Since the compiler must now work with generic programs, the analysis phases during compilation have the opportunity to produce more general information. For example, we have devised a simple type system in which size information about generic programs is included in their types. Consider the program ($\mathsf{tail}\ ;\ \mathsf{halve}$), which knocks off the first element of a list, and splits the remainder into two equal length parts. In our system, the derived type $*^{2n+1} \rightarrow (*^n, *^n)$ captures precisely that this program only works properly with odd length lists.

Not surprisingly, generic primitives are defined recursively, on the structure (i.e. shape rather than content) of streams. Proofs involving generics naturally proceed by induction. For example, it is easy to show that $(\mathsf{map}\ R\,;\mathsf{map}\ S) = \mathsf{map}\ (R\,;S)$, a generalisation of the example theorem in Section 2.1. Just as for the simple static primitives, the validity of many theorems involving generics follows directly from their polymorphic types [Sheeran89].

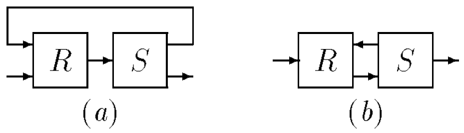

Figure 2: Bi–directional communication between R and S

2.4 Relations

In keeping with the special constraints of hardware, many regular circuits make extensive use of bi–directional communication between components. For example, the systolic correlator in [Jones90b] has data flowing both rightwards and leftwards. While bi–directional communication can be achieved in the manner of Figure 2a, this breaks with the convention that constructive programs capture both shape and behaviour. In this sense, it would clearly be preferable to have components communicating directly, as in Figure 2b. Working in a functional language however, this is not directly possible, due to the uni–directional flow of data inherent in function composition.

While the data flow problem could be worked around by introducing a few special combining forms to capture bi–directional communication, a much more acceptable solution is to weaken the normal functional constraint, and allow inputs and outputs to be distributed throughout the domain and range of a program. In this manner, the standard composition operator "$;$" may be used to combine any two components, regardless of whether they communicate bi–directionally or not. It is this observation which originally led Sheeran to consider using relations rather than functions, thereby removing the distinction between input and output entirely.

Not surprisingly, the jump to relations brings much more than bi–directionality properties. Since they are not biased towards a particular direction of data flow, relational combining forms tend to be more symmetric, and hence a single Ruby law often replaces a number of μFP laws. Furthermore, unlike in the (total) functional world, where only bijective functions may be inverted, every relation R has an inverse R^{-1}, defined[3] by $x \; R^{-1} \; y \;\widehat{=}\; y \; R \; x$. In terms of pictures, relational inverse corresponds to reflection of a program about the vertical axis.

The ability to invert programs means that many constructions which would normally have to be defined inductively may be defined quite naturally in terms of other related primitives. For example, the generic combining forms row and col tile components beside and above one another respectively; using inverse, one may be defined in terms of the other: $\mathsf{col}\ R \;\widehat{=}\; (\mathsf{row}\ R^{-1})^{-1}$. Defining components in this way also reduces the burden of proof. For example, any row theorem may be transformed into an analogous col theorem, without repeating the steps of the proof. It is interesting to note the similarity to the powerful notion of "duality" in Category Theory [Barr90], under which one proof yields two theorems.

Relational inverse has also proved useful in capturing abstraction and refinement steps in program derivation [Jones90a]. For example, given an initial word–level design, we can formally move down to a bit–level version by pushing the "refinement

[3]In relational notation, $x \; \mathsf{R} \; y$ is simply a shorthand for $(x, y) \in \mathsf{R}$.

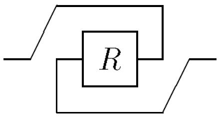

Figure 3: $R^{-1} \;\widehat{=}\; \mathsf{left} \,;\, \mathsf{fst}\ (\mathsf{snd}\ R) \,;\, \mathsf{right}$

relation" bits (which relates numeric values to bit vectors) in from the left hand side of the program, and the "abstraction relation" bits^{-1} in from the right.

While it is clear that many simple combining forms like fst may be defined in terms of others, it is perhaps surprising to find that in a relational framework, even such powerful constructs as loop and relational inverse may be defined in terms of simpler components. For example, Figure 3 shows how inverse may be defined using the plumbing relations[4] given by $x\ \mathsf{left}\ ((y,y),x)$ and $((x,y),y)\ \mathsf{right}\ x$. In fact, all (non–generic) components can be defined in terms of 4 basic constructs — composition ";", product "$\times$", the delay element D, and a "spread" construction which allows us to represent combinational primitives and plumbing relations. This approach is developed in more detail in [Rossen90].

Since a function may be viewed as a restricted kind of relation, Ruby naturally admits a larger class of programs than μFP. For example, ignoring streams for simplicity, the program $\mathsf{loop}\ (\mathsf{and} \,;\, \mathsf{split})$ which was not acceptable in the functional framework of Section 2.2, has a perfectly well defined meaning as $\{(x,y) \mid y \equiv x \wedge y\}$ in Ruby, which simplifies to the relation $\{(\mathsf{F},\mathsf{F}),(\mathsf{T},\mathsf{F}),(\mathsf{T},\mathsf{T})\}$. Such programs are sometimes referred to as having non–deterministic behaviour, in the sense that a set of results may be produced for a given input value. In this case for example, T in the domain relates to both F and T in the range.

3 Causal Relations

Moving to relations is perhaps the most natural way to allow bi–directional communication over composition, but causes implementation and semantic problems. In particular, while relations are useful for specification and refinement of a design, the end product is normally functional, even though inputs and outputs may be distributed throughout the domain and range. With no inherent notion of data–flow, computing with relations is generally speaking much less efficient than computing with functions, even though most programs will in fact be used functionally. Furthermore, relational languages are not so semantically well behaved as their functional counterparts. For example, it is known that the standard fixed–point approach to recursion does not naturally extend to the relational world.

In this section, we consider how to get some of the expressive power of relations, without incurring the implementation and semantic problems. Our solution lies with what we shall call *causal relations*, a novel blend of the functional and relational styles. The intent is that we may use the full power of relations during program derivation, with the satisfaction of knowing that a final causal design has a functional style semantics, and may be implemented in an efficient manner.

[4]As shown in $\mathsf{left}/\mathsf{right}$, it is usual to omit "$\widehat{=}\ \mathsf{T}$" from plumbing definitions.

3.1 Causality

We define a relation to be *causal* if we may identify an 'input part' of the relation, which totally and uniquely determines the remaining 'output part'. Unlike functions however, the input part is not restricted to the domain (left side) of a causal relation, nor output part to the range (right side); indeed, inputs and outputs may be interleaved throughout the domain and range.

For example, $\mathsf{not} = \{(\mathsf{F},\mathsf{T}),(\mathsf{T},\mathsf{F})\}$ is a causal relation, since the first component of each pair uniquely determines the second. Moreover, the second component also determines the first. This is perfectly acceptable; a causal relation may have many such functional interpretations. Conversely, $(\mathsf{or}^{-1}\,;\,\mathsf{and}) = \{(\mathsf{F},\mathsf{F}),\ (\mathsf{T},\mathsf{F}),\ (\mathsf{T},\mathsf{T})\}$ is not causal, since no part of the relation uniquely determines the remainder. In particular, T in the domain relates to both F and T in the range; similarly for F in the reverse direction.

To capture precisely what me mean by causality, we use a slight modification of the "equivalence of spans" construction of binary relations [deMoor90]. We start by reviewing the standard construction of relations in terms of binary products.

Given sets A and B, their *cartesian product* is the set

$$A\times B \;\hat{=}\; \{(a,b) \mid a \in A \,\wedge\, b \in B\}$$

together with *projection functions* $\pi_A : A\times B \to A$ and $\pi_B : A\times B \to B$, defined by

$$\begin{array}{lcl} \pi_A & \hat{=} & \lambda\,(a,b)\,.\,a \\ \pi_B & \hat{=} & \lambda\,(a,b)\,.\,b \end{array}$$

The construction is universal, in that given any other set W, together with total functions (we assume from now on that all functions are total) $f : W \to A$ and $g : W \to B$, there exists a unique *span*[5] $\langle f,g\rangle : W \to A\times B$, defined by

$$\langle f,g\rangle \;\hat{=}\; \lambda\,x\,.\,(f\,x, g\,x)$$

such that $\langle f,g\rangle\,;\,\pi_A = f$ and $\langle f,g\rangle\,;\,\pi_B = g$, as shown in the diagram below.

$$\begin{tikzcd} & W \arrow[dl, "f"'] \arrow[d, dotted, "\langle f,g\rangle"] \arrow[dr, "g"] & \\ A & A\times B \arrow[l, "\pi_A"'] \arrow[r, "\pi_B"] & B \end{tikzcd}$$

From the universality of the product, we deduce the useful law

$$h\,;\langle f,g\rangle = \langle h\,;f, h\,;g\rangle$$

which we shall use without comment to simplify expressions involving spans. In this framework, a binary relation of type $A \leftrightarrow B$ is normally defined as a subset of $A\times B$.

[5]Normally, a span is defined as a pair of functions (f,g) with common domain, while $\langle f,g\rangle$ is called a *product function*. In this paper, for reasons of brevity, we prefer to call each $\langle f,g\rangle$ a span, citing the one–to–one correspondence between product functions and spans in our defense.

Since we are interested in functional relations however, we take a different route, modelling relations in terms of spans. The connection between the two is that the image of a span $\langle f, g\rangle : W \to A\times B$ is a relation of type $A \leftrightarrow B$, defined by

$$\{f, g\} \ \hat{=}\ \{(f\,x, g\,x) \mid x \in W\}$$

Think of each $x \in W$ as a *witness* that $f\,x \in A$ is related by $\{f, g\}$ to $g\,x \in B$. While spans allow us to model relations in terms of functions, the representation is clearly not unique. We now proceed to define an equivalence relation "$\equiv$" on spans, such that if $\langle f, g\rangle : W \to A\times B$ and $\langle f', g'\rangle : W' \to A\times B$, then

$$\langle f, g\rangle \equiv \langle f', g'\rangle \quad \Leftrightarrow \quad \{f, g\} = \{f', g'\}$$

Given $R \ \hat{=}\ \langle f : W \to A, g : W \to B\rangle$ and $R' \ \hat{=}\ \langle f' : W' \to A, g' : W' \to B\rangle$, a *span morphism* $m : R \to R'$ is a function $m : W \to W'$, such that $\langle f, g\rangle = m;\langle f', g'\rangle$.

$$\begin{tikzcd} & W \arrow[dl, "f"'] \arrow[dr, "g"] \arrow[dd, "m"] & \\ A & & B \\ & W' \arrow[ul, "f'"] \arrow[ur, "g'"'] & \end{tikzcd}$$

Span morphisms induce a pre–ordering "$\preceq$" on spans, defined by

$$R \preceq R' \ \hat{=}\ \exists m : R \to R'$$

It is easy to see that $\{f, g\} \subseteq \{f', g'\}$ follows from $\langle f, g\rangle \preceq \langle f', g'\rangle$. Because "$\preceq$" is a pre–order, it can be extended to an equivalence relation on spans, defined by

$$R \equiv R' \ \hat{=}\ R \preceq R' \ \wedge\ R' \preceq R$$

It follows immediately that two spans generate the same relation precisely when they are equivalent under "$\equiv$". For this reason, we model a relation not by a single span $\langle f, g\rangle$, but by its entire equivalence class, which we shall denote $[f, g]$.

Before considering causal relations in full, let us start with a simple case. Given a function $f : A \to B$, the span $\langle id_A, f\rangle$ determines the corresponding relation, $\{(a, f\,a) \mid a \in A\}$. In this manner, a relation $[f, g] : A \leftrightarrow B$ is functional from A to B precisely when there exists an $h : A \to B$, such that $\langle f, g\rangle \equiv \langle id_A, h\rangle$.

Causal relations are a generalisation of these functional relations, in that inputs are not restricted to the domain, nor outputs to the range. We start by defining what it means for a relation to be generated by a function.

Definition: Let $\langle s,t\rangle : I \rightarrow A\times B$ and $[f,g] : A \leftrightarrow B$, such that $\langle s,t\rangle \in [f,g]$. The span $\langle s,t\rangle$ is called a *functional interpretation* of the relation $[f,g]$ precisely when $\langle s,t\rangle = \langle id_I, h\rangle \mathbin{;} \iota^{-1}$, where $h : I \rightarrow O$ is a total function, and $\iota : A\times B \rightarrow I\times O$ is a unique isomorphism.

In diagramatic form, everything fits as follows.

$$\begin{tikzcd}
& W \arrow[dl, "f"'] \arrow[dr, "g"] \arrow[d, dotted, "\langle f,g\rangle"] & \\
A & A\times B \arrow[l, "\pi_A"] \arrow[r, "\pi_B"'] \arrow[d, dotted, shift right, "\iota"'] & B \\
I & I\times O \arrow[l, "\pi_I"'] \arrow[r, "\pi_O"] \arrow[u, dotted, shift right, "\iota^{-1}"'] & O \\
& I \arrow[ul, "id_I"] \arrow[ur, "h"'] \arrow[u, dotted, "\langle id_I, h\rangle"] &
\end{tikzcd}$$

While the projections π_A and π_B decompose the product $A\times B$ into domain and range parts, the functions $\iota \mathbin{;} \pi_I$ and $\iota \mathbin{;} \pi_O$ allow us to decompose the relation into input and output parts, such that the first uniquely determines the second, as witnessed by the function $h : I \rightarrow O$. More precisely, it follows from definition of the span $\langle s,t\rangle$ and the universality of the product $I\times O$ that

$$\langle s,t\rangle \mathbin{;} \iota \mathbin{;} \pi_I \mathbin{;} h = \langle s,t\rangle \mathbin{;} \iota \mathbin{;} \pi_O$$

The restriction to $I\times O$ being uniquely isomorphic to $A\times B$ ensures that the decomposition is purely structural. In particular, the familiar isomorphisms

$$\begin{array}{rcl} X\times Y & \cong & Y\times X \\ X\times(Y\times Z) & \cong & (X\times Y)\times Z \end{array}$$

confirm our intuition about how a relation may be permuted such that all the input parts are in the domain, and output parts in the range. The less well known $X \cong X\times 1$, where 1 is any singleton set, tells us that the unique $h : A\times B \rightarrow 1$ forms part of a functional interpretation of the full relation on $A\times B$.

Returning to our original motivation for functional interpretations, each such $\langle s,t\rangle$ allows us to generate the relation, just by looking at the input part, in that

$$\langle s,t\rangle \equiv \iota \mathbin{;} \pi_I \mathbin{;} \langle s,t\rangle \qquad (1)$$

where $\iota;\pi_I$ picks out the input part of any pair in $A\times B$. The span morphisms which verify the equivalence are $\langle id_I, h\rangle \mathbin{;} \iota^{-1}$ in the $I \rightarrow A\times B$ direction, and $\iota \mathbin{;} \pi_I$ in the reverse direction. Naturally, $\langle s,t\rangle$ preserves the input part we supply, in that

$$\iota \mathbin{;} \pi_I = \iota \mathbin{;} \pi_I \mathbin{;} \langle s,t\rangle \mathbin{;} \iota \mathbin{;} \pi_I$$

which follows immediately from the definition of $\langle s, t \rangle$. It is a useful exercise at this point to show that the $h : I \to O$ part of a functional interpretation is uniquely determined by the isomorphism ι. In other words, for each input/output decomposition, there is at most one function which precisely generates the relation.

While general relations may be modelled by an equivalence of spans, we will model a causal relation by a *non–empty equivalence of functional interpretations*. The fact that there may be many such interpretations just means that there may be many ways to generate the complete relation just by looking at some part of it; many ways to "drive the relation". Non–causal relations have no such functional interpretations, and hence do not fit into our model of causality.

To clarify our construction of causal relations, let us consider a simple example. Suppose we assert that the relation $[f, g] : X \leftrightarrow Y \times Z$ is really a function $h : X \times Z \to Y$ in disguise. If we take $I \;\widehat{=}\; X \times Z$, $O \;\widehat{=}\; Y$, and $\iota \;\widehat{=}\; \lambda\,(x, (y, z))\,.\,((x, z), y)$, our informal statement of the underlying functionality of the relation may be properly cast as the following equivalence of spans, where $\langle s, t \rangle \;\widehat{=}\; \langle id_I, h \rangle \,;\, \iota^{-1}$.

$$\langle f, g \rangle \equiv \langle s, t \rangle$$

Since $\langle s, t \rangle$ is a functional interpretation, it follows from property (1) that

$$\langle f, g \rangle \equiv \iota \,;\, \pi_I \,;\, \langle s, t \rangle$$

Using this equivalence, we can generate elements of the relation using the function h; given any $(x, (\perp, z))$ in $X \times (Y \times Z)$, where the value of $\perp$ is not important, we may produce an $(x, (y, z)) \in \{f, g\}$ as follows, where $y \;\widehat{=}\; h\,(x, z)$.

$$(x, (\perp, z)) \stackrel{\iota}{\longmapsto} ((x, z), \perp) \stackrel{\pi_I}{\longmapsto} (x, z) \stackrel{\langle id_I, h \rangle}{\longmapsto} ((x, z), y) \stackrel{\iota^{-1}}{\longmapsto} ((x, y), z)$$

3.2 Causality is not enough

While causality ensures that we only work with "functional relations", it is not the end of the story in itself. First of all, causal relations are not closed under composition. For example, both or^{-1} and and are causal, but $(\mathsf{or}^{-1} \,;\, \mathsf{and})$ is not, as we saw at the start of Section 3.1. Secondly, there are causal programs which involve non–functional data flow. For example, $(\mathsf{and}^{-1} \,;\, \mathsf{and})$ is equivalent to the identity relation on booleans (certainly a causal program), but operationally has non–functional flow of information between the two primitives. To ensure that causal programs are closed under composition, and have a functional semantics, we must restrict the way in which they are built. In particular, we require that programs are *well–directed*, and have no *unbroken loops*.

The composition $(R \,;\, S)$ is well–directed if all information flow is functional, in the sense that outputs in the range of R match with inputs in the domain of S, and vice–versa. For example, $(\mathsf{and} \,;\, \mathsf{not})$ is well–directed, since the output from the first component matches with the input of the second. Conversely, both $(\mathsf{or}^{-1} \,;\, \mathsf{and})$ and $(\mathsf{and} \,;\, \mathsf{and}^{-1})$ are ill–directed, due to a clash of inputs in the first case, and outputs in the second. In other words, both components in a composition must be used functionally in their own right, in addition to the functionality of the composition as a whole. In the next section, we describe a simple system in which it is possible to capture all the well–directed ways in which a causal program may be used.

While the direction constraint filters out most non–causal programs, some programs involving feedback slip through the net. For example, $\mathsf{loop}\ (\mathsf{and}\ ;\mathsf{split})$ is well–directed, but as mentioned in Section 2.4, corresponds to the non–causal relation $\{(\mathsf{F},\mathsf{F}),\ (\mathsf{T},\mathsf{F}),\ (\mathsf{T},\mathsf{T})\}$. This problem is solved by insisting that all feedback loops must be broken by a delay element. For example, as we saw in Section 2.2, $\mathsf{loop}\ (\mathsf{snd}\ \mathsf{D}_\mathsf{T}\ ;\mathsf{and}\ ;\mathsf{split})$ is a perfectly valid causal program.

4 Directions

A relational program is causal if we can identify input and output parts. If internally, inputs and outputs match over composition, it is also well–directed. In the functional style, programs are automatically well–directed, since inputs are restricted to the domain, and outputs to the range. In moving to causal relations, we have removed the normal contextual distinction between input and output, so we now have an obligation to check that programs are indeed well–directed. In this section we describe a system in which it is possible to capture all the functional ways in which a causal program may be used. Examining directions provides considerable insight into the expressive power of causal relations.

4.1 Notation

Whereas types tell us what kind of data are expected, *directions* specify which parts of the data are inputs, and which are outputs. In general, since a causal relation R may have many functional interpretations, it has a *set* of directions, which we shall denote R^*. For example, we write $\mathsf{not}^* = \{(\mathsf{in},\mathsf{out}),(\mathsf{out},\mathsf{in})\}$ to mean that the not relation is functional from domain to range, and range to domain.

In this setting, ill–directed programs such as $(\mathsf{and}\,;\mathsf{or}^{-1})$ correspond to the empty set of directions $\emptyset$. Following the terminology for types, a program with more than one direction will be called *polydirectional*.

4.2 What are directions?

While the directions for simple cases like and and not are intuitively obvious, it is important to understand precisely where such sets come from, and what they actually mean. We use the simple order–theoretic notion of a *lattice*; a set A, equipped with a partial–ordering "$\sqsubseteq$", least/greatest elements $\bot_A$ and $\top_A$, and binary meet and join operations, denoted by "$\sqcap$" and "$\sqcup$" respectively.

Under the ordering $\mathsf{out} \sqsubseteq \mathsf{in}$, the primitive directions $\{\mathsf{out},\mathsf{in}\}$ form a 2–point lattice, which we shall call D. It is easy to see that the lattice structure is closed under the product construction. For example, $D \times D$ is a lattice, with $\bot_{D\times D} = (\mathsf{out},\mathsf{out})$ and $\top_{D\times D} = (\mathsf{in},\mathsf{in})$, under the ordering $(a,b) \sqsubseteq (c,d)\ \hat{=}\ a \sqsubseteq c\ \wedge\ b \sqsubseteq d$.

Just as $\perp$ and $\top$ are often used without subscript to denote the least and greatest elements in any lattice, so we will sometimes use out and in to denote the least and greatest elements of any product lattice constructed from D.

Recall from Section 3.1 that a causal relation $[f, g] : A \leftrightarrow B$ is modelled as an equivalence of functional interpretations $\langle s, t\rangle \in [f, g]$, such that each $\langle s, t\rangle$ has the form $\langle id_I, h\rangle \,;\, \iota^{-1}$, where $h : I \to O$ is a total function, and $\iota : A \times B \to I \times O$ a unique isomorphism. In this context, it is clear that directions have the same shape as the type of a causal relation, in that

$$[f, g]^* \subseteq [\![A \times B]\!]$$

where $[\![\cdot]\!]$ is a function which converts arbitrary products into direction lattices, such that for example $[\![X \times (Y \times Z)]\!] = D \times (D \times D)$. If we extend each ι to an isomorphism $[\![\iota]\!] : [\![A \times B]\!] \to [\![I \times O]\!]$ in the natural manner, it is clear that each functional interpretation $\langle s, t\rangle$ corresponds to a single direction, given by

$$\langle s, t\rangle^* \;\widehat{=}\; [\![\iota]\!]^{-1}\,(\top_{[\![I]\!]}, \perp_{[\![O]\!]})$$

Returning to our example at the end of Section 3.1, where $\langle s, t\rangle$ is a functional interpretation of a relation of type $X \leftrightarrow Y \times Z$, with $\iota = \lambda\,(x, (y, z))\,.\,((x, z), y)$, we obtain a direction corresponding to the span $\langle s, t\rangle$ as follows.

$$\begin{array}{rcl}
\langle s, t\rangle^* & = & [\![\iota]\!]^{-1}\,(\top_{[\![I]\!]}, \perp_{[\![O]\!]}) \\
 & = & [\![\iota]\!]^{-1}\,(\top_{[\![X\times Z]\!]}, \perp_{[\![Y]\!]}) \\
 & = & [\![\iota]\!]^{-1}\,(\top_{D\times D}, \perp_{D}) \\
 & = & [\![\iota]\!]^{-1}\,((\mathsf{in}, \mathsf{in}), \mathsf{out}) \\
 & = & ((\mathsf{in}, \mathsf{out}), \mathsf{in})
\end{array}$$

4.3 Composition

In lattice terminology, two elements a and b are said to be *complementary* if $a \sqcup b = \top$ and $a \sqcap b = \perp$. For example, $(\mathsf{in}, \mathsf{out})$ is the complement of $(\mathsf{out}, \mathsf{in})$ in $D \times D$. It is not hard to see that every direction has a unique complement. In Section 3.2 we said that for the composition $(R\,; S)$ to be *well–directed*, outputs in range of R must match with inputs in the domain of S, and vice–versa. Clearly then, two directions are only compatible over composition if they are complementary. Using this insight, we may now state precisely all the well–directed ways in which $(R\,; S)$ may be used:

$$a\,(R\,; S)^*\,d \;\widehat{=}\; (a\,R^*\,b) \;\wedge\; (c\,S^*\,d) \;\wedge\; (b \sqcup c = \mathsf{in}) \;\wedge\; (b \sqcap c = \mathsf{out})$$

For example, although not has two possible directions in isolation, only the left–to–right orientation is acceptable in $(\mathsf{and}\,;\,\mathsf{not})^* = \{((\mathsf{in}, \mathsf{in}), \mathsf{out})\}$. In keeping with the fact that not is its own inverse, we have $(\mathsf{not}\,;\,\mathsf{not})^* = \{(\mathsf{in}, \mathsf{out}), (\mathsf{out}, \mathsf{in})\} = \mathsf{not}^*$.

4.4 Plumbing relations

Plumbing relations are (parametrically) polymorphic, in that they may be viewed as a collection of monomorphic instances which, in some sense, behave in the same way.

Not surprisingly, plumbing relations are also polydirectional. By way of example, let us consider id, the identity element for composition. We start with the simplest instance, id_A, where A is an atomic type such as the booleans or integers. While it may appear that $\mathsf{id}_A{}^* = D \times D$, this is not in fact the case. Since id is the identity for composition, we have that $(\mathsf{id} \,;\, \mathsf{id}) = \mathsf{id}$. Clearly this property should also hold for directions. However, working with $\mathsf{id}_A{}^* = D \times D$, we find that $(\mathsf{id}_A \,;\, \mathsf{id}_A)^* = \{(\mathsf{out}, \mathsf{in}), (\mathsf{in}, \mathsf{out})\} \neq \mathsf{id}_A{}^*$. This situation arises because the rule for (;) insists that inputs match with outputs. Thus, we conclude that $\mathsf{id}_A{}^* = \{(\mathsf{out}, \mathsf{in}), (\mathsf{in}, \mathsf{out})\}$.

Consider a more complicated instance, $\mathsf{id}_{A\times A}$. A simple calculation shows that the directions for this expression may be given in terms of those for its components:

$$\begin{array}{rcll} \mathsf{id}_{A\times A}{}^* & = & (\mathsf{id}_A \times \mathsf{id}_A)^* & [\times \text{ is a functor}] \\ & = & \mathsf{id}_A{}^* \times \mathsf{id}_A^* & [* \text{ distributes over } \times] \end{array}$$

Using this equivalence, and recalling the definition of product on relations, $(a, b)\ R{\times}S\ (c, d) \mathrel{\widehat{=}} a\,R\,c \wedge b\,S\,d$, we find that there are 4 directions for $\mathsf{id}_{A\times A}$. While $(\mathsf{out}, \mathsf{out}) \leftrightarrow (\mathsf{in}, \mathsf{in})$ and its complement are perfectly intuitive, $(\mathsf{in}, \mathsf{out}) \leftrightarrow (\mathsf{out}, \mathsf{in})$ and its complement may seem a little strange, since they allow data flow in opposite directions over id at the same time. In a relational setting however, this kind of behaviour is quite natural. For example, we could imagine $(\mathsf{T}, \perp)\ \mathsf{id}\ (\perp, \mathsf{F})$ resolving to $((\mathsf{T}, \mathsf{F}), (\mathsf{T}, \mathsf{F}))$. In general then, it is not hard to see that id^* is precisely the pairs of complementary directions:

$$\mathsf{id}^* \mathrel{\widehat{=}} \{(a, b) \mid a \sqcup b = \mathsf{in} \ \wedge \ a \sqcap b = \mathsf{out}\}$$

In terms of hardware, the polymorphic identity relation id may be viewed as an arbitrary bus of wires. Under this interpretation, $a \sqcup b = \mathsf{in}$ means that each component wire is driven at least once, $a \sqcap b = \mathsf{out}$ means that each wire is driven at most once; together they ensure that each wire has precisely one value.

5 Further Developments

In this paper, we presented causal relations as a new programming paradigm, particularly well suited to the bi–directionality demands of "programming in hardware". We have given a model of causality, in terms the equivalence of spans construction of relations, and presented a simple system in which we may examine various functional interpretations of a causal relation. At the present moment however, it is not clear how to incorporate the direction and feedback constraints into our model, and hence give a proof of closure under composition. Although the mathematical aspects of causality are not yet complete, we have taken some steps towards a causal implementation of the Ruby language.

In Prolog, bi–directional communication may be achieved using logic variables. In particular, if two processes A and B wish to communicate in both directions, A may pass a stream of pairs to B, with one component of each pair being a message from A, and the other being an uninstantiated variable, in which B may reply. Just as direction inference is important for causal languages, so mode annotations (and to a lesser extent mode inference) is important in logic languages. It is clearly important to investigate the use of bi–directionality in logic programming, and bring out the differences and similarities to our causal relational approach.

Acknowledgements

This work was completed under the SERC *Relational Programming* project.

References

[Backus78] John Backus. *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs.* CACM vol. 9, August 1978.

[Bird88] Richard Bird. *Lectures on constructive functional programming.* Oxford University, 1988. (PRG–69)

[Barr90] Michael Barr and Charles Wells. *Category Theory for Computing Science.* Prentice Hall, 1990.

[deMoor90] Oege de Moor. *Categories, Relations and Dynamic Programming.* Programming Research Group, Oxford University 1990. (PRG–TR–18–90)

[Hutton90] Graham Hutton. *Functional Programming with Relations.* Proc. Third Glasgow Workshop on Functional Programming (ed. Kehler Holst, Hutton and Peyton Jones), Ullapool 1990, Springer Workshops in Computing.

[Jones90a] Geraint Jones and Mary Sheeran. *Relations and refinement in circuit design.* Proc. BCS FACS workshop on refinement (ed. Carroll Morgan), Hursley, January 1990.

[Jones90b] Mary Sheeran and Geraint Jones. *Circuit design in Ruby.* Formal Methods for VLSI Design (ed. Staunstrup), Elsevier Science Publications (Amsterdam), 1990.

[Rossen90] Lars Rossen. *Formal Ruby.* Formal Methods for VLSI Design (ed. Staunstrup), Elsevier Science Publications (Amsterdam), 1990.

[Sheeran81] Mary Sheeran. *Functional geometry and integrated circuit layout.* M.Sc. dissertation, University of Oxford, 1981.

[Sheeran83] Mary Sheeran. *μFP – an algebraic VLSI design language.* D.Phil. thesis, Oxford University, 1983. (PRG–39)

[Sheeran89] Mary Sheeran. *Categories for the working hardware designer.* Hardware specification, verification and synthesis: Mathematical aspects (ed. Leeser et al), Springer LNCS, 1989.

[Wadler89] Phil Wadler. *Theorems for Free!* Proc. Conference on Functional Programming and Computer Architecture, London, Springer 1989.