Getting your wires crossed

Geraint Jones*

Things should be as simple as possible 'and no simpler', so they say. This paper is about simplifying a construct in Ruby, to reduce the complexity of the algebra without sacrificing expressive power. Ruby, for which Mary Sheeran is largely responsible, is a language of relations and a collection of rules – equalities – for algebraic transformation of expressions in that language.

The presentation here is by way of being a natural history of the combining forms that build state machines. It shows that over time the operators have evolved as we have been able to simplify calculation by chosing to name combining forms which have simple properties, rather than naming those forms which correspond to the original intuition about the application area, and this apparently without making it significantly harder to relate the calculations to reality. Whether there are any general lessons which can be drawn from this history, I leave to the reader.

μ, Mealy, Moore and Mary

μFP is a language of functions [6], from which Ruby has been developed. The μ in its name is the operator which constructs a state machine from its transition function, something traditionally denoted by μ. This operator constructs an automaton $\mu[f,g]$ from two functions, f which delivers the output as a function of the present state and input, and g which delivers the next state also as a function of the present state and input.

This is the characterisation of a state machine known as a Mealy machine, as opposed to Moore machines, which describe the same machine by a different pair of functions. An implementation is likely, for detailed operational reasons like propagation delay, to be more like a Mealy machine than the apparently simpler corresponding Moore machine which calculates its output as a function of its state. So it was the Mealy machine that was chosen as a named combining form to be added to Backus' FP [1]. It is an odd coincidence that Mealy, Moore, and Mary might all be being spelled with a capital μ.

The μ operator of μFP satisfies the equation

$$\mu[f,g] \cdot \mu[h,j] \quad = \quad \mu[f \cdot [h \cdot [1, 2 \cdot 2], 1 \cdot 2], [g \cdot [h \cdot [1, 2 \cdot 2], 1 \cdot 2], j \cdot [1, 2 \cdot 2]]]$$

which is called Cμ in reference [6]. Here the dot is (forward) composition of functions, the ones and twos are the FP notation for the projection functions which deliver the first and second components of a pair, and $[f,g]$ is the function that returns the pair of results of applying each of f and g to its argument.

*Address: Programming Research Group, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, England; Email: *geraint.jones@comlab.oxford.ac.uk*.

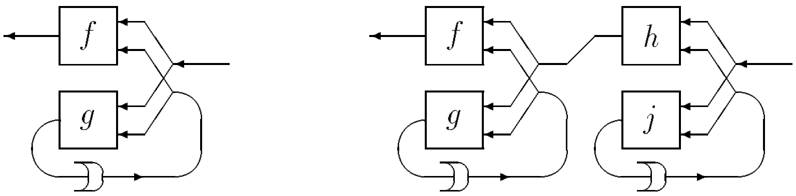

Figure 1: the μFP form $\mu[f,g]$, and the composition $\mu[f,g]\cdot\mu[h,j]$, each drawn with the conventions of [6], inputs on the right, outputs on the left, indexed from top to bottom

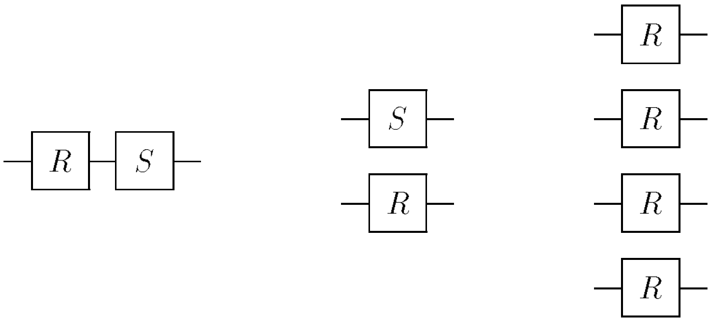

Figure 2: $R\,;S$ and $[R,S]$, and an instance of $\mathsf{map}\,R$, drawn with the Ruby conventions: domain on the left, range on the right, indexed from bottom to top

The law Cμ expresses the composition of two state machines – that is a circuit in which the output of one machine is supplied as input to another, as illustrated in figure 1 – as a single state machine, albeit with a more complicated state and a more complicated output function and a much more complicated next-state function. The composite machine just has as its state the states of the two machines as a pair, although this is difficult to see from the form of Cμ.

This law has two possible applications: having composed two machines in this way, it might be possible to simplify the expressions $f\cdot[h\cdot[1,2\cdot 2],1\cdot 2]$ and $[g\cdot[h\cdot[1,2\cdot 2],1\cdot 2],j\cdot[1,2\cdot 2]]$ which describe its output function and its next-state function. More likely, the law would be used to factorise a large state-machine into two smaller ones: for distributed implementation, or for VLSI implementation, it pays to have the state as near as possible to the places where it is going to be used.

Unfortunately, in order to use the law to factorise some machine $\mu[F,G]$ it would be necessary first to be able to find f, g, h and j for which $F = f\cdot[h\cdot[1,2\cdot 2],1\cdot 2]$ and $G = [g\cdot[h\cdot[1,2\cdot 2],1\cdot 2],j\cdot[1,2\cdot 2]]$. On the face of it this is a daunting prospect, since there is no obvious approach.

Ruby

This section treats rapidly most of the Ruby notation used in this paper. If you are unfamiliar with the notation an introduction can be found in, for example, reference [3].

A circuit is represented by the relation between the signals at its terminals, rather than a function. This abstracts from the distinction between inputs and outputs. We think of binary relations with domain on the left and range on the right so that the relational composition, $R;S$, defined by $x\,(R\,;S)\,z \iff \exists y.\, x\,R\,y\,\&\,y\,S\,z$, reads naturally from left to right, as illustrated in figure 2. We tend to give to

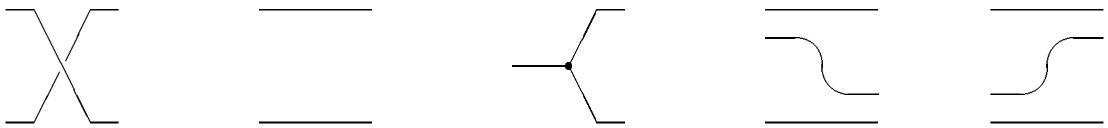

Figure 3: *swp*, 2, *fork*, *rsh*, and *lsh*, with grouping of signals represented by spacing

circuits names that suggest input in the domain and output in the range, but this is just a matter of convention and not part of the formalism.

One of the advantages of using relations is that we can easily treat the inverse (or more properly 'converse'), R^{-1}, of any relation, for which $x\, R^{-1}\, y \iff y\, R\, x$. Frequently this appears as a part of a conjugation, $R \setminus S = S^{-1} ; R ; S$, where the S is effectively a translation applied to all the signals from R.

The parallel composition of two circuits, $[R, S]$, relates a pair of signals in the domain to a pair of signals in the range if and only if the components are related by the corresponding component circuits, $\langle a, b\rangle\, [R, S]\, \langle x, y\rangle \iff a\, R\, x \,\&\, b\, S\, y$. In drawings, such as figure 2, the convention is that lists read from left to right if you stand in the range of the relation, facing into the domain. In the frequent case that one of the circuits is just an abstract wire – a wire or perhaps a bundle of wires, represented by the identity (or 'equality') relation, id – we abbreviate this to $\mathsf{fst}\, R = [R, id]$ or $\mathsf{snd}\, S = [id, S]$.

Parallel composition and sequential composition distribute over each other, meaning that $[A, B] ; [C, D] = [A ; C,\ B ; D]$. Be careful that parallel composition in Ruby differs from the $[f, g]$ of μFP, which supplies one input to both f and g to produce a pair of outputs. That would correspond more closely to something like $fork ; [R, S]$ where $x\, fork\, y \iff y = \langle x, x\rangle$.

Repeated composition we write as exponentiation, so that $R^1 = R$ and $R^{n+1} = R^n ; R$. Similarly for negative powers, $R^{-n} = (R^{-1})^n = (R^n)^{-1}$. The repeated parallel composition of R is denoted by $\mathsf{map}\, R$, and relates two lists of signals if they are the same length and corresponding elements are related by R, that is $x\, (\mathsf{map}\, R)\, y \iff \#x = \#y \;\&\; \forall i.\, x_i\, R\, y_i$.

In this paper we need also a small number of primitive circuits, essentially wiring like *fork*, which we define here. The identity on pairs we write $2 = [id, id]$, and *swp* is the relation between pairs whose components are reversed, $\langle x, y\rangle\, swp\, \langle y, x\rangle$. Each of 2 and *swp* is its own inverse. The right- and left-shift relations on pairs are defined by $\langle x, \langle y, z\rangle\rangle\, rsh\, \langle\langle x, y\rangle, z\rangle$ and $\langle\langle x, y\rangle, z\rangle\, lsh\, \langle x, \langle y, z\rangle\rangle$. Clearly each is the other's inverse.

The wiring *zip* represents the interleaving of two buses of wires: it relates a pair of lists of signals to a list of pairs, provided the lists are of equal length and corresponding values are equal, that is $\langle x, y\rangle\, zip\, z \iff \#x = \#y = \#z \;\&\; \forall i.\, \langle x_i, y_i\rangle = z_i$.

Delay, statelessness, and state

As outlined so far, Ruby is a language in which only combinational circuits can be described. To deal with sequential circuits there is a slightly more sophisticated reading of the same formulae. A sequential circuit is a relation between two signals, and a signal is a doubly infinite sequence of samples. The samples can be thought of as the values of signals sampled on consecutive cycles of a logical clock. There need not be a physical clock, nor need the circuit be synchronous.

Simple combinational circuits like the wiring relations are promoted to time-sequences by acting pointwise: in the new reading, $x\ R\ y$ if and only if in the old reading, $x_t\ R\ y_t$ for all times t. Any circuit which is the promotion of some moment-by-moment relation like this is *stateless*.

The meaning of sequential composition, inverse, conjugate, and so on can be read immediately from the definition. However in the case of parallel composition, since all circuits are relations between a single signal in the domain and a single signal in the range, the $\langle a, b\rangle$ in the definition must be read as interleaving – *zip*-ping together – a and b into a sequence for which $\langle a, b\rangle_t$ is the pair of a_t and b_t for each time t. Similarly $\mathsf{map}\, R$ relates a single time-sequence consisting of the interleaved domain signals, a list of values at each time, to a similar sequence in the range.

The properties of the operators are preserved by this reading, for example that sequential and parallel composition distribute over each other, so the algebra of the language is unchanged by the change of semantics. Moreover any expression consisting only of combinational primitives and the operators introduced so far can be read by promoting all the primitives and applying the new readings of the operators. It will be stateless, and its meaning in this reading will be the stateless promotion of its combinational reading.

In order to deal with state, one more primitive is needed. A delay, $\mathcal{D}$, relates two signals which are shifted by one time unit, $x\, \mathcal{D}\, y \iff \forall t.\, x_t = y_{t+1}$. It might be a latch or register with its input in the domain and output in the range, supplying its output one time unit later in the range. Its converse, the anti-delay $\mathcal{D}^{-1}$, appears to be a circuit which predicts in the range what its domain signal is going to be one unit later. This need not be as unrealistic as it seems at first sight, because it may be a circuit taking its input in the range and giving output in the domain.

Delay and anti-delay are cancellable, that is $\mathcal{D} \,;\, \mathcal{D}^{-1} = id = \mathcal{D}^{-1} \,;\, \mathcal{D}$, because there is no 'first time' and so there is no initial undefined state of the latch. Note that $\mathcal{D}$ does not have any combinational reading: any expression with $\mathcal{D}$ in it must necessarily mean a relation between time-sequences. However equations in the Ruby operators remain true even when circuits with state, such as expressions containing $\mathcal{D}$, are substituted for the variables.

Although $\mathcal{D}$ does not appear as such in μFP, it is possible to define an isolated delay circuit by the μFP expression $\mu[2, 1]$, but notice that it is not possible to construct a circuit which behaves like an anti-delay. Moreover (at least in the semantics given in reference [6], where there is a first instant and the output of the latch is initially undefined) there is no function which can be cancelled on *whichever* side of this delay it is composed.

The μFP operator in Ruby

The first thing to do, to make later comparison fairer, is to cast the original μ operator into Ruby. I claim that the operation on relations which corresponds to the μ operator in μFP is μ_{M}, defined by

$$i\ (\mu_{\mathrm{M}}\, R)\ o \iff \exists s, s'.\ \langle i, s\rangle\ R\ \langle o, s'\rangle\ \&\ s'\ \mathcal{D}\ s$$

and illustrated in figure 4. It takes a transition relation R and returns a machine

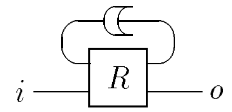

Figure 4: $\mu_{\mathrm{M}}\, R$ drawn with the conventions of Ruby

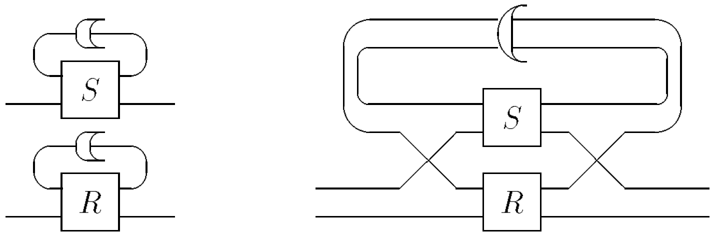

Figure 5: $[\mu_{\mathrm{M}}\, R, \mu_{\mathrm{M}}\, S]$ and $\mu_{\mathrm{M}}([R,S] \setminus zip^{-1})$

$\mu_{\mathrm{M}}\, R$. The transition relation relates a pair consisting of an input and a present state, to a pair consisting of an output and a next state. The definition says that the next state, s', is related to the present state, s, by a single delay $\mathcal{D}$; that is to say that the present state is the previous value of the next-state terminal of the relation.

This is in some ways a more natural description of a state machine, because the description in μFP tends to lead you to have as part of the transition-function the fork (on the right of the pictures in figure 1) which copies the input to each of the components of $[f, g]$. Parts of this wiring might not be needed in any particular machine.

A pair of these machines operating in parallel can always be rewritten as a single machine, using the fact that $\mathcal{D}\,;2 = [\mathcal{D}, \mathcal{D}]$

$$[\mu_{\mathrm{M}}\, R, \mu_{\mathrm{M}}\, S] \quad = \quad \mu_{\mathrm{M}}([R,S] \setminus zip^{-1})$$

as illustrated in figure 5. This says that the parallel composition of two state machines is a state machine whose transition relation is the parallel composition of those of the constituent machines, but with the state and input/output suitably interleaved.

There seems to be nothing in reference [6] that corresponds to this law, because there is nothing in μFP which corresponds simply to Ruby's parallel composition – the $[R, S]$ on the right-hand side would be difficult to express. Parallel composition in Ruby generalises to map, and the rule about the parallel composition of μ_{M} machines generalises to

$$\mathsf{map}\,\mu_{\mathrm{M}}\, R \quad = \quad \mu_{\mathrm{M}}(\mathsf{map}\, R \setminus zip^{-1})$$

which might have been expressed in μFP, but seems never to have been, at least not in anything I have read so far, although a circuit like $\mathsf{map}\,\mu_{\mathrm{M}}\, R$ is mentioned in passing in reference [7].

In fact the form used in reference [7], which was a first published attempt at choosing a reasonable suite of combining forms for Ruby, is

$$i\ (\mathsf{loop}\, R)\ o \quad \Longleftrightarrow \quad \exists s.\ \langle i, s\rangle\ R\ \langle o, s\rangle$$

which omits the delay from the feedback path. This makes no difference to the algebraic laws quoted so far, and Mary quotes as justification the symmetry of

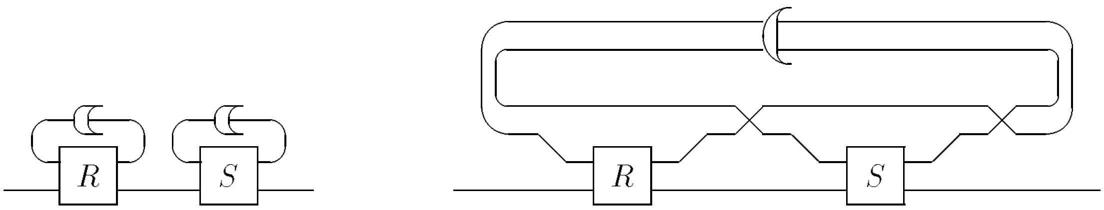

Figure 6: $\mu_{\mathrm{M}}\, R\,;\mu_{\mathrm{M}}\, S$ and $\mu_{\mathrm{M}}(rsh\,;\mathsf{fst}\,R\,;lsh\,;\mathsf{snd}\,swp\,;rsh\,;\mathsf{fst}\,S\,;lsh\,;\mathsf{snd}\,swp)$

$(\mathsf{loop}\,R)^{-1} = \mathsf{loop}(R^{-1})$, in contrast to $(\mu_{\mathrm{M}}\,R)^{-1} = \mu_{\mathrm{M}}(R^{-1}\,;\mathsf{snd}\,\mathcal{D}^{-2})$. I will return to the advisability of omitting the delay once I have sorted out the wiring.

Sequential (de)composition of state machines

As I said earlier, we are really interested in sequential decomposition: in taking large state machines and expressing them as compositions of machines with simpler transition relations. In search of techniques for this, consider first how to represent the composition of two state machines as a single machine. The equality about expressions in μ_{M} that corresponds to Cμ is

$$\mu_{\mathrm{M}}\,R\,;\mu_{\mathrm{M}}\,S \quad = \quad \mu_{\mathrm{M}}(rsh\,;\mathsf{fst}\,R\,;lsh\,;\mathsf{snd}\,swp\,;rsh\,;\mathsf{fst}\,S\,;lsh\,;\mathsf{snd}\,swp)$$

each side of which is illustrated in figure 6.

Already some progress has been made by arranging the plumbing as a collection of applications of fst and snd and instances of *lsh* and *swp*. Indeed this equation is so much simpler than the original that you might think that it does not capture the same thing. In a sense that is true: this equation is only about the merging of the two state machines into one. The original statement of Cμ in μFP has also interleaved the transition relations, that is it has gone on to simplify $rsh\,;\mathsf{fst}\,R\,;lsh\,;\mathsf{snd}\,swp\,;rsh\,;\mathsf{fst}\,S\,;lsh\,;\mathsf{snd}\,swp$, given the knowledge that in the case that corresponds most closely to the μFP the relations R and S are of the form $fork\,;[f,g]$ and $fork\,;[h,j]$.

Pursuing those assumptions is probably not worth doing in general, since in the cases which happen to arise in applications it may be much easier to simplify the particular $rsh\,;\mathsf{fst}\,R\,;lsh\,;\mathsf{snd}\,swp\,;rsh\,;\mathsf{fst}\,S\,;lsh\,;\mathsf{snd}\,swp$ than to deal with a more complex expression in separate relations determining the outputs and next states. There is clearly a pattern in this composite transition relation and the next thing to do is to study that pattern.

Beside and below

Ruby has been particularly successfully applied to describing and designing regular arrays connected in Cartesian grids. These are particularly attractive for the implementer, because they have simple connections, each of which is local. Yet they are sufficiently general to describe anything that might reasonably be built on a silicon surface.

To describe these grids, we use two new combining forms, $R \leftrightarrow S$, read 'R beside

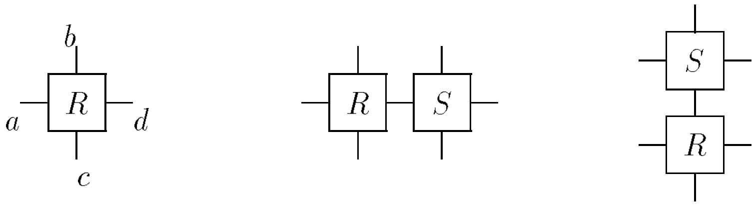

Figure 7: $\langle a, b\rangle\, R\, \langle c, d\rangle$ and layouts suggested by $R \leftrightarrow S$ and $R \updownarrow S$

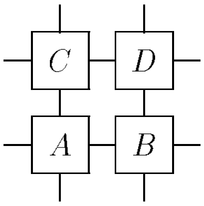

Figure 8: $(A \leftrightarrow B) \updownarrow (C \leftrightarrow D) = (A \updownarrow C) \leftrightarrow (B \updownarrow D)$

S', and $R \updownarrow S$, read 'R below S', defined by

$$\begin{aligned} \langle a, \langle b, c\rangle\rangle\, (R \leftrightarrow S)\, \langle\langle x, y\rangle, z\rangle \quad &\iff \quad \exists n.\ \langle a, b\rangle\, R\, \langle x, n\rangle\ \ \&\ \ \langle n, c\rangle\, S\, \langle y, z\rangle \\ \langle\langle a, b\rangle, c\rangle\, (R \updownarrow S)\, \langle x, \langle y, z\rangle\rangle \quad &\iff \quad \exists n.\ \langle a, n\rangle\, R\, \langle x, y\rangle\ \ \&\ \ \langle b, c\rangle\, S\, \langle n, z\rangle \end{aligned}$$

and illustrated in figure 7, which uses the convention that wires on the left and top of the picture are in the domain of the relation and those on the bottom and the right are in the range. The ordering is the same as for all the other pictures: stand in the range, facing the domain, and read from left to right; and the division between left and top is that between elements of a pair, as is that between bottom and right. Consequently, inversion of a relation corresponds to reflecting it about the line from bottom-left to top-right which divides domain from range. (This is the Tees-Exe convention, after the geographical dividing line from the Tees estuary in the North East of England to the Exe estuary in the South West, which divides the moors and the mountains from the mud and the money.)

Beside and below need not be thought of as new primitives, for

$$\begin{aligned} R \leftrightarrow S \quad &= \quad rsh\, ;\, \mathsf{fst}\, R\, ;\, lsh\, ;\, \mathsf{snd}\, S\, ;\, rsh \\ R \updownarrow S \quad &= \quad lsh\, ;\, \mathsf{snd}\, S\, ;\, rsh\, ;\, \mathsf{fst}\, R\, ;\, lsh \end{aligned}$$

However it is very useful to have names for these forms both because they occur often in practical circuits, and because they have a simple algebra: for example they distribute over each other in the sense that both of

$$(A \leftrightarrow B) \updownarrow (C \leftrightarrow D) \quad = \quad (A \updownarrow C) \leftrightarrow (B \updownarrow D)$$

describe the same relation, illustrated in figure 8. Beside and below are duals of each other in the sense that $(R \leftrightarrow S)^{-1} = (R^{-1}) \updownarrow (S^{-1})$ and $(R \updownarrow S)^{-1} = (R^{-1}) \leftrightarrow (S^{-1})$. Moreover, $rsh = 2 \leftrightarrow 2$ and $lsh = 2 \updownarrow 2 = rsh^{-1}$, where 2 is the identity on pairs, as illustrated in figure 9.

In the same way that parallel composition can be generalised to map, the beside and below operators generalise to row and column operators, defined by

$$\begin{aligned} &\langle a, bs\rangle\ \mathsf{row}\, R\, \langle ys, z\rangle \\ &\qquad \iff \quad \#bs = \#ys\ \ \& \\ &\qquad\qquad\quad (\exists ps.\, a = ps_0\ \ \&\ \ \forall i.\, \langle ps_i, bs_i\rangle\, R\, \langle ys_i, ps_{i+1}\rangle\ \ \&\ \ ps_{\#bs} = z) \end{aligned}$$

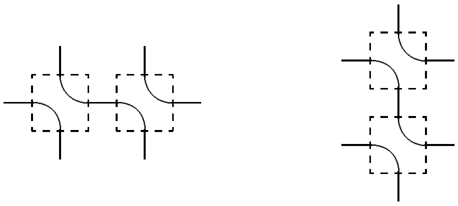

Figure 9: $rsh = 2 \leftrightarrow 2$ and $lsh = 2 \updownarrow 2$

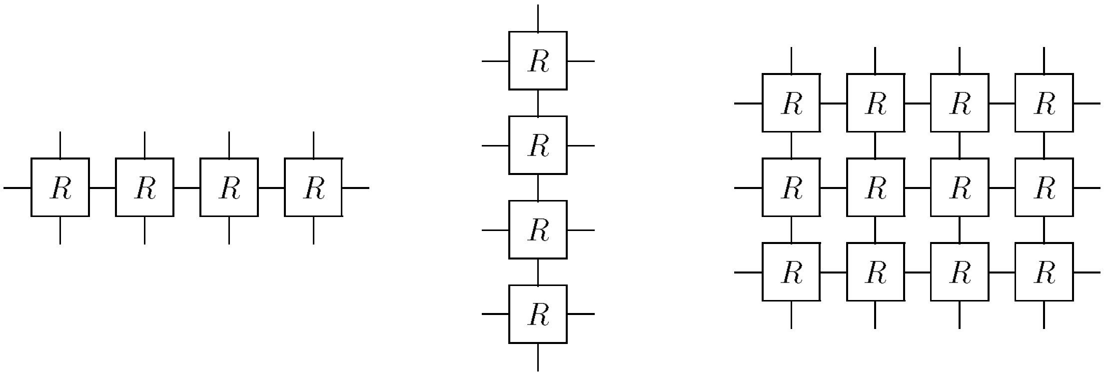

Figure 10: layouts for instances of $\mathsf{row}\, R$, $\mathsf{col}\, R$, and $\mathsf{row}\,\mathsf{col}\, R = \mathsf{col}\,\mathsf{row}\, R$

and

$$\begin{aligned} &\langle as, b\rangle\ \mathsf{col}\, R\ \langle y, zs\rangle \\ &\qquad \iff \quad \#as = \#zs\ \ \& \\ &\qquad\qquad\quad (\exists ps.\ b = ps_{\#as}\ \ \&\ \ \forall i.\ \langle as_i, ps_{i+1}\rangle\ R\ \langle ps_i, zs_i\rangle\ \ \&\ \ ps_0 = y) \end{aligned}$$

and row and col are also duals, meaning that $(\mathsf{row}\, R)^{-1} = \mathsf{col}(R^{-1})$ and $\mathsf{row}(R^{-1}) = (\mathsf{col}\, R)^{-1}$, so we might write col as $\overline{\mathsf{row}}$ and and row as $\overline{\mathsf{col}}$. Figure 10 shows the expected layouts for row and column. It is a consequence of the distribution of beside and below over each other that row and col commute, meaning that $\mathsf{row}\,\mathsf{col}\, R = \mathsf{col}\,\mathsf{row}\, R$. Unsurprisingly if we define $\mathsf{grid}\, R = \mathsf{row}\,\mathsf{col}\, R$, an instance of which also appears in figure 10, then grid is its own dual.

A good idea pinched from Wayne

Wayne Luk and I spent a considerable time, as part of an Alvey project, applying Ruby to the designs of regular arrays. As a result we fell into a way of thinking where rows and columns, besides and belows were almost simpler to deal with than sequential and parallel compositions. I do not know how he came to think of the idea, but it was at his suggestion that I adopted a slightly different state-machine operator in the lecture notes for my Ruby course [2].

This is one for which the composition of two state machines is a machine whose transition relation is made by putting the component translations together with the beside operator. Specifically,

$$i\ (\mu_{\mathrm{W}}\, R)\ o \quad \iff \quad \exists s, s'.\ \langle i, s\rangle\ R\ \langle s', o\rangle\ \ \&\ \ s'\ \mathcal{D}\ s$$

so $\mu_{\mathrm{W}}\, R = \mu_{\mathrm{M}}(R\,;swp)$ and $\mu_{\mathrm{M}}\, R = \mu_{\mathrm{W}}(R\,;swp)$. This is shown laid out following

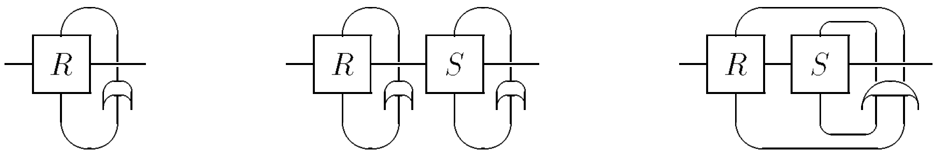

Figure 11: $\mu_{\mathrm{W}}\, R$, the composition $\mu_{\mathrm{W}}\, R\,;\mu_{\mathrm{W}}\, S$ and $\mu_{\mathrm{W}}(R \leftrightarrow S)$

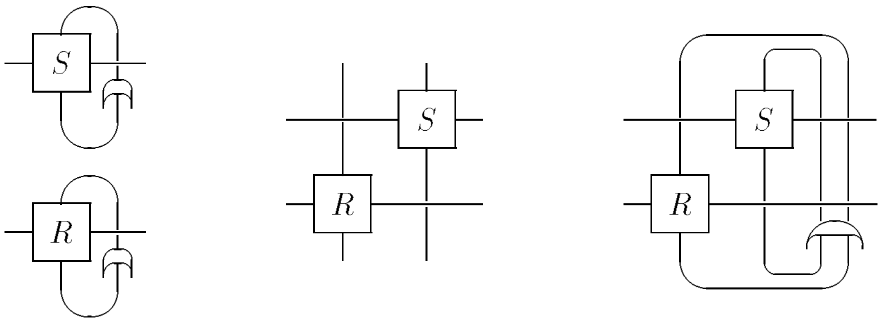

Figure 12: $[\mu_{\mathrm{W}}\, R, \mu_{\mathrm{W}}\, S]$, $[R, S] \setminus zip^{-1}$, and $\mu([R, S] \setminus zip^{-1})$

the Tees-Exe conventions in figure 11, along with a composition of machines which satisfies

$$\mu_{\mathrm{W}}\, R\,;\mu_{\mathrm{W}}\, S \quad = \quad \mu_{\mathrm{W}}(R \leftrightarrow S)$$

That, which is now the statement of Cμ, is so much simpler than the original form that I find it difficult to see why we did not think of it sooner! You can of course calculate one form of Cμ from the other, given the definitions and a bit of patience with the properties of the plumbing relations.

The law describing the parallel composition of two state machines, when cast in term of μ_{W} turns out to be formally identical to that for μ_{M}, that is

$$[\mu_{\mathrm{W}}\, R, \mu_{\mathrm{W}}\, S] \quad = \quad \mu_{\mathrm{W}}([R, S] \setminus zip^{-1})$$

as is immediate from the definition, because $[swp, swp]\,; zip^{-1} = zip^{-1}\,; swp$. This law is illustrated in figure 12, and generalises to

$$\mathsf{map}\, \mu_{\mathrm{W}}\, R \quad = \quad \mu_{\mathrm{W}}((\mathsf{map}\, R) \setminus zip^{-1})$$

Retiming and slowing

One of the uses to which Ruby is put is called *retiming*. The highest clock frequency at which a synchronous clocked circuit can be operated, giving the highest throughput, is limited by the longest propagation delay from the output of a latch to the input of the next. In the course of optimising performance, latches are introduced and moved about to reduce the longest paths, but without changing the specified behaviour of the circuit.

A Ruby relation R for which $R\,; \mathcal{D} = \mathcal{D}\,; R$ is called *timeless*. Compositions of timeless relations are easy to retime, and this makes timeless relations particularly interesting. Suppose E is some expression in Ruby, that is an expression made up of Ruby operations and constants, and perhaps some variable names. Then on the assumption that all the variables in E are timeless, so also is E itself. That

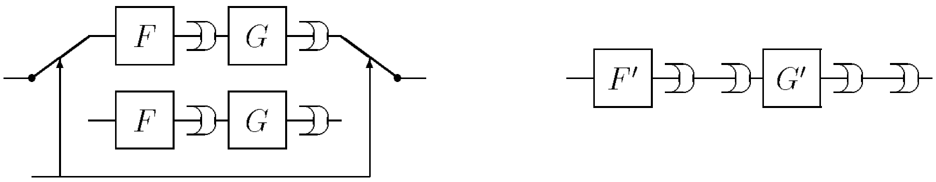

Figure 13: $\mathsf{slow}(F\,;\mathcal{D}\,;G\,;\mathcal{D})$ and $(\mathsf{slow}\,F)\,;\mathcal{D}^2\,;(\mathsf{slow}\,G)\,;\mathcal{D}^2$

is because each of the Ruby operators preserves timelessness, and all the constant circuits we usually deal with are timeless (even $\mathcal{D}$ itself).

At first it may seem surprising, but both μ_{M} and μ_{W} also preserve timelessness. Even though there is state held in the feedback latches, it is still the case that if $\mathcal{D}\,;R = R\,;\mathcal{D}$ then necessarily $\mathcal{D}\,;\mu_{\mathrm{M}}\,R = \mu_{\mathrm{M}}\,R\,;\mathcal{D}$ and $\mathcal{D}\,;\mu_{\mathrm{W}}\,R = \mu_{\mathrm{W}}\,R\,;\mathcal{D}$. What this means is that the fed-back state is not anchored to a particular time: Ruby delays are not reset at any particular time, and any initial setting has to be by explicit reset signals. This means that we could add either of these operators to a suite of basic operators in Ruby without loosing the property that all operators preserve timelessness.

Another manipulation which is used in optimising performance is *slowing*. A slow version of a circuit can be obtained by taking some expression which describes the circuit, and substituting $\mathcal{D}^2$ (or some other multiple delay, with consequent changes) for every instance of $\mathcal{D}$ in that expression. The slow circuit is one that has to be clocked twice as often, or rather at twice the frequency, to get the same outputs for the same inputs. This might not seem in itself a good thing, but it allows the designer to introduce extra latches which can be moved around in the course of retiming.

Since all the delays appear in pairs, it takes two clock cycles for the signal at the input to each delay to appear at the output. On even-numbered clock cycles, the circuit will be performing the same calculation as an unslowed circuit. The state trapped between the two latches in a pair of delays is never involved in the even-cycle calculation, but after one clock cycle only this state is connected to the rest of the circuit, so the circuit is also performing the same calculation on odd-numbered clock cycles.

That is to say, a slow R behaves like two copies of R, alternately connected to their environment so that they operate on interleaved inputs to produce interleaved outputs. Figure 13 is meant to suggest this: the circuit on the left contains two copies of $F\,;\mathcal{D}\,;G\,;\mathcal{D}$, clocked on alternate cycles of the clock which switches the multiplexers between the two copies. Its behaviour is the same as that of $(\mathsf{slow}\,F)\,;\mathcal{D}^2\,;(\mathsf{slow}\,G)\,;\mathcal{D}^2$, and if F and G are stateless this is $F\,;\mathcal{D}^2\,;G\,;\mathcal{D}^2$.

This correspondence is interesting because it shows that we can define slow as an operator on relations, rather than on Ruby expressions. That is, we define slow as the interleaving operation, and if R and S are different expressions which denote the same relation, $R = S$, it is necessarily the case that $\mathsf{slow}\,R = \mathsf{slow}\,S$. Substituting $\mathcal{D}^2$ for $\mathcal{D}$ in *any* expression which describes a circuit will give the same slow circuit.

That is not obvious: a substitution which is not of equals for equals need not be a well-defined operation on the meaning of an expression! It works for slow in Ruby because apart from $\mathcal{D}$ the primitives are all stateless, which means that $R = \mathsf{slow}\,R$, and slow distributes over all the combining forms, for example $\mathsf{slow}(R\,;S) = \mathsf{slow}\,R\,;\mathsf{slow}\,S$, and $\mathsf{slow}[R,S] = [\mathsf{slow}\,R, \mathsf{slow}\,S]$.

Figure 14: two layouts for $\mu\, R$ and two for $\bar{\mu}\, R$

However this property would be lost by adding either of the the state-machine operators to Ruby, because $\mathsf{slow}\; \mu_{\mathrm{M}}\, R \neq \mu_{\mathrm{M}}\, \mathsf{slow}\; R$, and $\mathsf{slow}\; \mu_{\mathrm{W}}\, R \neq \mu_{\mathrm{W}}\, \mathsf{slow}\; R$. If we define slow by the interleaving of relations, $\mathsf{slow}\; \mu_{\mathrm{M}}\, R = \mu_{\mathrm{M}}((\mathsf{slow}\; R)\,;\mathsf{snd}\,\mathcal{D})$ and likewise $\mathsf{slow}\; \mu_{\mathrm{W}}\, R = \mu_{\mathrm{W}}((\mathsf{slow}\; R);\mathsf{fst}\,\mathcal{D})$ so there is a substitution which corresponds to slowing, but it is more complicated because it has to deal specially with the μ operators as well as with $\mathcal{D}$.

A new μ and its dual

Clearly the problem with slowing state machines is that I have stubbornly retained the delay – hidden in the feedback path – which is the difference between slowing a machine and making a machine from a slowed transition relation. The difficulty that this causes can be overcome simply by omitting the delay, so I finally propose another μ operator defined by

$$i\,(\mu\, R)\, o \quad \Longleftrightarrow \quad \exists s.\ \langle i, s\rangle\ R\ \langle s, o\rangle$$

illustrated in figure 14. It can be used to implement either of the earlier operations because $\mu_{\mathrm{W}}\, R = \mu(R\,;\mathsf{fst}\,\mathcal{D})$ and so $\mu_{\mathrm{M}}\, R = \mu(R\,; swp\,; \mathsf{fst}\,\mathcal{D})$. Moreover, it should be entirely unsurprising that $\mathsf{slow}\;\mu\, R = \mu\,\mathsf{slow}\; R$.

The dual of this operation is given by $\bar{\mu}\, R = (\mu(R^{-1}))^{-1}$, and satisfies

$$i\,(\bar{\mu}\, R)\, o \quad \Longleftrightarrow \quad \exists s.\ \langle s, i\rangle\ R\ \langle o, s\rangle$$

so $\mu\, R = \bar{\mu}(R \setminus swp)$ and $\bar{\mu}\, R = \mu(R \setminus swp)$.

This is again a simplification over the earlier operators, whose duals introduced odd anti-delays. It would hardly have been worth noting that $\bar{\mu}_{\mathrm{M}}\, R = \mu_{\mathrm{M}}(R\,;\mathsf{snd}\,\mathcal{D}^{-2})$, nor that $\bar{\mu}_{\mathrm{W}}\, R = \mu_{\mathrm{W}}((R \setminus swp)\,;\mathsf{fst}\,\mathcal{D}^{-2})$, and while the excess anti-delays that appear on the right-hand sides can be moved about in the expression they cannot be eliminated. The loop operation is its own dual, $\overline{\mathsf{loop}}\; R = \mathsf{loop}\; R$, so there would be no way of re-expressing $\mathsf{loop}(R \setminus swp)$.

The delay-less μ operator has just as simple a form of Cμ as before,

$$\mu\, R\,;\mu\, S \quad = \quad \mu(R \leftrightarrow S)$$

as illustrated in figure 15, and the same simple law for parallel composition,

$$[\mu\, R, \mu\, S] \quad = \quad \mu([R, S] \setminus zip^{-1})$$

as illustrated in figure 16. Moreover, the operation dual to this new constructor

Figure 15: $\mu\, R \,;\, \mu\, S$ and $\mu(R \leftrightarrow S)$

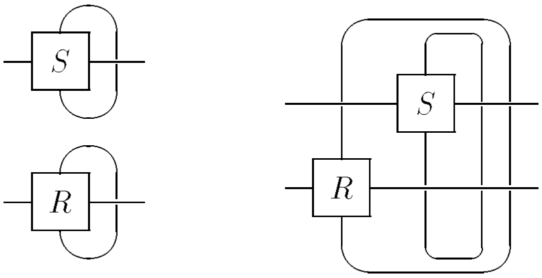

Figure 16: $[\mu\, R, \mu\, S]$ and $\mu([R,S] \setminus zip^{-1})$

has similarly simple laws, for

$$\begin{aligned}
\bar{\mu}\, R \,;\, \bar{\mu}\, S &= (\mu\, R^{-1})^{-1} \,;\, (\mu\, S^{-1})^{-1} \\
&= (\mu\, S^{-1} \,;\, \mu\, R^{-1})^{-1} \\
&= (\mu(S^{-1} \leftrightarrow R^{-1}))^{-1} \\
&= \bar{\mu}(S^{-1} \leftrightarrow R^{-1})^{-1} \\
&= \bar{\mu}(S \updownarrow R)
\end{aligned}$$

and

$$\begin{aligned}
[\bar{\mu}\, R, \bar{\mu}\, S] &= [(\mu\, R^{-1})^{-1}, (\mu\, S^{-1})^{-1}] \\
&= [\mu\, R^{-1}, \mu\, S^{-1}]^{-1} \\
&= (\mu([R^{-1}, S^{-1}] \setminus zip^{-1}))^{-1} \\
&= \bar{\mu}(([R^{-1}, S^{-1}] \setminus zip^{-1})^{-1}) \\
&= \bar{\mu}([R,S] \setminus zip^{-1})
\end{aligned}$$

which gives μ and $\bar{\mu}$ the same standing in calculations.

Other laws which these new operators satisfy are also simplified. For example, the interface laws

$$\begin{aligned}
A \,;\, \mu\, R \,;\, B &= \mu(\mathsf{fst}\, A \,;\, R \,;\, \mathsf{snd}\, B) \\
A \,;\, \bar{\mu}\, R \,;\, B &= \bar{\mu}(\mathsf{snd}\, A \,;\, R \,;\, \mathsf{fst}\, B)
\end{aligned}$$

are broadly the same as for the earlier operators, but the laws capturing the idea of change of state

$$\begin{aligned}
\mu(R \,;\, \mathsf{fst}\, S) &= \mu(\mathsf{snd}\, S \,;\, R) \\
\bar{\mu}(R \,;\, \mathsf{snd}\, S) &= \bar{\mu}(\mathsf{fst}\, S \,;\, R)
\end{aligned}$$

are much simpler, because there are no side-conditions about the timelessness of the change of state, S. Indeed they would be true for any relation S, even relations which are not themselves expressible in Ruby.

Many people, when they first see a delay-free μ operator, are concerned about whether $\mu\, R$ means anything in case the signal in the first component of the range depends immediately (that is, with no delay) on that in the second component of

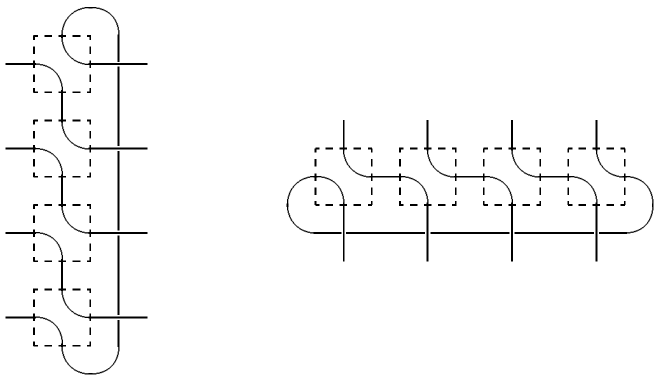

Figure 17: $\mu\,\mathsf{col}\,2$ and its inverse, $\bar{\mu}\,\mathsf{row}\,2$

the domain. The delay in the loop in the μFP primitive was there to ensure that any loop in the implementation of a μFP expression would contain a positive number of delays. There could therefore never be any loops in the causal dependency, and all signals in a circuit described in μFP are therefore necessarily defined.

This is not the case in Ruby, because it is not possible to guarantee that there are no loops in the causality graph: the delay in the feedback loop can be cancelled in, for example, $\mu_{\mathrm{M}}(R;\mathsf{snd}\,\mathcal{D}^{-1})$. Even if $\mathcal{D}$ and $\mathcal{D}^{-1}$, and even the taking of inverses, are omitted from the language, they can be constructed from μ_{M} and some plumbing, for $\mathcal{D} = \mu_{\mathrm{M}}\, swp$ and $\mathcal{D}^{-1} = \mu_{\mathrm{M}}(any \setminus fork)$, where $\langle a, b\rangle\ (any \setminus fork)\ \langle c, d\rangle \iff a = b\ \&\ c = d$.

However, this is to confuse the definedness of the signals with the definedness of the relations between them. Whatever the relation R, it yields a well-defined relation $\mu\, R$, and anything you can prove about $\mu\, R$ will be true whether or not the signal on the feedback path is defined by R. The causal relationship between inputs and outputs is one of the things from which Ruby chooses to abstract, and in consequence so is the definedness of signals whose definedness is not material to the observable behaviour of a circuit.

$\mathcal{D}$-light μ-using

It seems that I was not paying too much attention to Wayne's suggestion. While I adopted Wayne's wiring for μ_{W} in my course notes it was actually the delay-free μ that he used in reference [5] (where *it* also is called loop). In that paper he proves and uses properties of the delay-free operator, such as that $(\mu\, R)^n = \mu\,\mathsf{row}_n\, R$, where $\mathsf{row}_n\, R = \mathsf{snd}\, n\ ; \mathsf{row}\, R\ ; \mathsf{fst}\, n$ is a row with a size of exactly n. (This follows immediately from Cμ by an induction on n.)

However it turns out that all the uses of loop in the paper are instances of $\mu(R\,;\mathsf{fst}\,\mathcal{D})$ and so might as well have been uses of μ_{W}. Moreover, all the properties used are also properties of μ_{W} as well!

The thinking in the present paper came about from a need to express a particular $\mu(\mathsf{slow}\, R\ ; \mathsf{fst}\,\mathcal{D})$ in a form in which the slow was at the outside. This arose in the course of the formal development of a bit-serial circuit [4]. It happens that there is no S for which $\mu(\mathsf{slow}\, R\ ; \mathsf{fst}\,\mathcal{D}) = \mathsf{slow}\, S$, because the behaviour of the circuit is different on different phases of the clock. We need to do more work on such multi-phase circuits. Clarity of thought in the reasoning in that paper was only

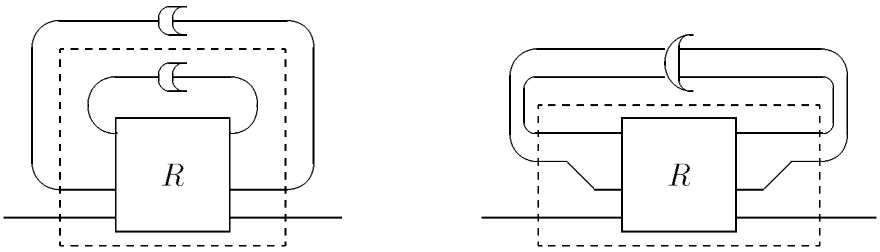

Figure 18: $\mu_{\mathrm{M}}\,\mu_{\mathrm{M}}\,R$ and $\mu_{\mathrm{M}}(R \setminus lsh)$

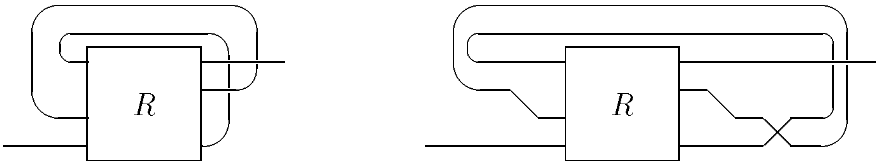

Figure 19: $\mu\,\mu\,R$ and $\mu(rsh\,;R\,;rsh\,;\mathsf{fst}\,swp)$

possible by divorcing the feedback and the delay.

In the course of the reasoning about that bit-serial arithmetic circuit other instances of μ with no delays on the feedback path arose naturally in the calculations. Consider the circuit $\mathsf{col}\,2$, which relates a pair $\langle as, b\rangle$ to a pair $\langle y, zs\rangle$ if and only if $as + \langle b\rangle = \langle y\rangle + zs$. That is to say it is a left-shift operation, generalising $lsh = \mathsf{col}_2\,2$, and similarly $\mathsf{row}\,2$ is a right-shift operation on an arbitrarily wide tuple. In both cases a single signal is moved in at one end of the tuple, and a single one is carried out at the other end. The circuits $\mu\,\mathsf{col}\,2$ and $\bar{\mu}\,\mathsf{row}\,2$, illustrated in figure 17, similarly accomplish cyclic shifts. Of course, in the case of rows and columns of length two, $\mu\,\mathsf{col}_2\,2 = \bar{\mu}\,\mathsf{row}_2\,2 = swp$.

Things that are less simple

One of the μFP laws in reference [6] shows that a state machine whose transition function is itself a state machine can be simplified; in the μFP notation, that

$$\mu\,\mu[[f,g],h] \quad = \quad \mu([f,[g,h]]\cdot[[1,1\cdot 2],2\cdot 2])$$

Here the $[[1,1\cdot 2],2\cdot 2]$ is just the function corresponding to the rsh relation of Ruby, and $[f,[g,h]] = [1\cdot 1,[2\cdot 1,2]]\cdot[[f,g],h]$ and $[1\cdot 1,[2\cdot 1,2]]$ is the function that corresponds to lsh, so the nearest corresponding equation in the notation of this paper would be

$$\mu_{\mathrm{M}}\,\mu_{\mathrm{M}}\,R \quad = \quad \mu_{\mathrm{M}}(R \setminus lsh)$$

which is illustrated in figure 18. This form would be the same, whether or not there were a delay in the feedback loop. The corresponding equation for the new μ operator, illustrated in figure 19, is

$$\begin{aligned}\mu\,\mu\,R \quad &= \quad \mu(rsh\,;R\,;rsh\,;\mathsf{fst}\,swp)\\ &= \quad \mu(\mathsf{snd}\,swp\,;rsh\,;R\,;rsh)\end{aligned}$$

which reveals the twist in the wires, and it would be similar for μ_{W}. I do not think I have ever had need to use this sort of law. However

$$\mu\,\bar{\mu}\,R \quad = \quad \bar{\mu}\,\mu(lsh\,;R\,;lsh)$$

and so, taking inverses throughout,

$$\bar{\mu}\,\mu\,R \quad = \quad \mu\,\bar{\mu}(rsh\,;R\,;rsh)$$

should those ever arise in practice.

Much ado about...

There is nothing extraordinary in anything I am reporting here: the sequential composition of μs was all in Mary's original Cμ in her thesis [6] almost ten years ago. In fact it is no easier to factorise a relation into $R \leftrightarrow S$ than into

$$rsh\,;\mathsf{fst}\;R \qquad ;lsh\,;\mathsf{snd}\,swp\,;rsh\,;\mathsf{fst}\;S \qquad ;lsh\,;\mathsf{snd}\,swp$$

because $R \leftrightarrow S$ is equal to

$$(rsh\,;\mathsf{fst}\,(R\,;swp)\,;lsh\,;\mathsf{snd}\,swp\,;rsh\,;\mathsf{fst}\,(S\,;swp)\,;lsh\,;\mathsf{snd}\,swp)\,;swp$$

and the three extra swps are exactly the difference between μ_{M} and μ_{W}. On the other hand, I think I can justifiably claim that the simpler algebra of the present form of the Ruby μ has encouraged more creative exploration of circuits built from state machines than I could contemplate if faced with expressions like $f\cdot[h\cdot[1,2\cdot 2],1\cdot 2]$ and $[g\cdot[h\cdot[1,2\cdot 2],1\cdot 2],j\cdot[1,2\cdot 2]]$.

In this case it is not so much a matter of getting the names of things right as of getting to name the right things.

References

[1] Backus J. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. Commun ACM 1978;21:613–641

[2] Jones G. Designing circuits by calculation. Programming Research Group technical report PRG–TR–10–90. Oxford University Computing Laboratory, 1990

[3] Jones G, Sheeran M. Circuit design in Ruby. In: Staunstrup J (ed) Formal methods for VLSI design. North-Holland, 1990, pp 13–70

[4] Jones G, Sheeran M. Deriving bit-serial circuits in Ruby. In: Halaas A, Denyer P (eds) VLSI'91. (Proc IFIP TC10/WG10.5 Intl Conf, Edinburgh, Scotland, Aug 1991) North-Holland.

[5] Luk W, Brown G. A systolic LRU processor and its top-down development. Sci Comp Prog, 1990;15:217–233

[6] Sheeran M. μFP – an algebraic VLSI design language.
Programming Research Group technical monograph PRG–39.
Oxford University Computing Laboratory, 1984

[7] Sheeran M. Describing and reasoning about circuits using relations. In: McEvoy K, Tucker JV (eds) Theoretical foundations of VLSI design. (Proc Leeds workshop, 1986) Cambridge University Press, 1990, pp 263–298

Bernard of Chartres used to say that we are like dwarfs on the shoulders of giants, so that we can see more than they, and things at a greater distance, not by virtue of any sharpness of sight on our part, or any physical distinction, but because we are carried high and raised up by their giant size.

John of Salisbury, Metalogicon (1159)

If I have seen further it is by standing on the shoulders of giants.

Isaac Newton, in a letter to Robert Hooke