Relations and Refinement in Circuit Design

Geraint Jones* Oxford University

Mary Sheeran† Glasgow University

Abstract

A language of relations and combining forms is presented in which to describe both the behaviour of circuits and the specifications which they must meet. We illustrate a design method that starts by selecting representations for the values on which a circuit operates, and derive the circuit from these representations by a process of refinement entirely within the language.

Formal methods have always been used in circuit design. It would be unthinkable to attempt to design combinational circuits without using Boolean algebra. This means that circuit designers, unlike programmers, already use mathematical tools as a matter of course. It also means that we have a good basis on which to build higher level formal design methods. Encouraged by these observations, we have been investigating the application of formal program development techniques to circuit design.

We view circuit design as the transformation of a program describing the required behaviour into an equivalent program that is suitable for direct implementation in hardware, so there is little difference between hardware and software design. A program is implementable in hardware if it can be made from wires and circuit primitives. A program that is elegantly regular will result in a good, efficient circuit because it is communication rather than computation that is expensive in hardware. Ugly, 'spaghetti' programs are simply unacceptable as hardware designs. Because of this we have been developing notations and transformations that encourage the design of regular planar structures.

In our earlier work on the design language μFP, we modelled circuits as stream functions, and the ways that circuits are plugged together we modelled by higher-order functions [Sheer84]. These higher-order functions were chosen to have simple mathematical properties, giving a useful set of theorems for use in design by transformation. Unsurprisingly, the higher-order functions for building regular structures have attractive mathematical properties as well as being suitable for hardware implementation. This functional approach has been shown to be useful in the design of regular array architectures [Bhan89, Luk89, Jones86]. Bird and Meertens have pioneered a similar calculational approach to program design [Bird87, Bird88, Meert86, Meert89]. Their methods – which are still developing very rapidly – have been used to perform many impressive program derivations. We have presented a derivation in the Bird-Meertens style of the fast Fourier transform from its specification [Jones89].

*Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, England

†Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland

Our more recent work has been on a generalisation of the earlier functional approach. We now model circuits as binary relations between streams. The symmetry of relations gives them interesting mathematical properties [Sheer87, Luk88]. In particular, having relational inverse (converse) as a higher-order function has proved to be invaluable.

A key benefit of this approach is that it allows us to describe and reason about both combinational and sequential circuits in a uniform framework. We transform sequential circuits in much the same way as a designer uses the laws of Boolean algebra to transform combinational circuits. Reference [Sheer88] discusses the use of relations to reason about time in sequential circuits. For simplicity, we often consider only combinational circuits: circuits can then be modelled as relations between data values rather than as relations between streams of data values. Subsequent lifting to streams presents no problems and preserves the algebraic properties of the structural higher-order functions [Jones88], so the sequential design follows immediately from the combinational design.

The aim of this paper is to give a fairly gentle introduction to our calculus of relations and to show how it is used to describe and reason about data abstraction in arithmetic circuits.

Using relations to describe hardware

We view a circuit as a binary relation between data values. In this paper we discuss only combinational circuits, so the possible data values are integers, tuples of data values and homogeneous lists of data values.

For example, the primitive relation $+$ relates a pair of integers to their integer sum. To assert that x is related by the relation R to y, we write $x \; R \; y$. Examples of true statements about $+$ are $(1,3) + 4$, $(-1,1) + 0$ and $(-3,2) + -1$. The definition of $+$ is $(x,y) + z \equiv z = x + y$. The plus sign on the right hand side is ordinary (infix) addition on the integers. We read this definition as saying that the pair (x,y) is related to $x + y$ and to no other value, so $+$ is a function from its domain to its range.

The identity relation we write as ι, and we name some common identities: *int* is the identity on the integers, *nat* on the naturals, and *bit* on the set $\{0,1\}$. The primitive relations are

$$\begin{aligned}
(x,y) + z &\equiv z = x + y \\
(x,y) \times z &\equiv z = xy \\
y \times 2 \; z &\equiv z = 2y \\
(x,y) \; \pi_1 \; z &\equiv z = x \\
(x,y) \; \pi_2 \; z &\equiv z = y
\end{aligned}$$

The only one of these relations for which there is a direct hardware implementation is the identity on bits, which corresponds to a single wire. However there are implementations for restricted versions of some relations: for example, an 'and' gate is an implementation of $\times$ restricted to bits. We assume a standard library of primitive components: in this paper, we use only logic gates, half-adders and full-adders. Primitive components usually operate on bits, so we will have to find suitable ways of representing data in terms of bits.

Circuit descriptions are built by combining relations using higher-order functions. The first two, composition and inverse, are standard operations on relations.

$$x \, (R \, ; S) \, z \equiv \exists y. \, x \; R \; y \; \& \; y \; S \; z$$

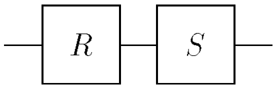

Figure 1: the composition $R\,;S$

$$x\,(R^{-1})\,y \quad\equiv\quad y\,R\,x$$

They have the expected mathematical properties.

$$\begin{aligned}(R\,;S)^{-1} &= S^{-1}\,;R^{-1}\\ (R^{-1})^{-1} &= R\\ R\,;(S\,;T) &= (R\,;S)\,;T\end{aligned}$$

We draw pictures of circuits using the convention that the 'left-hand' edge is the domain and the 'right-hand' edge is the range. This means that the composition $R\,;S$ is drawn as shown in figure 1, and that taking the inverse of a circuit means flipping its picture about a vertical axis. Imagine flipping figure 1 to get a picture of $S^{-1}\,;R^{-1}$. Now you see why the equation $(R\,;S)^{-1} = S^{-1}\,;R^{-1}$ holds. Repeated composition is defined by $R^1 = R$ and $R^{n+1} = R\,;R^n$. It can be shown by induction that $(R^n)^{-1} = (R^{-1})^n$, so we feel safe in writing R^{-n} for $(R^n)^{-1}$.

It is convenient to model not only circuits but types as relations. A *type* is an equivalence relation: that is a relation R for which $R = R^{-1} = R^2$. If D and R are types for which $C = D\,;C\,;R$, we say that a relation C has type D to R, and write $C : D \sim R$. The type D is a type of the domain of C and the type R is a type of the range of C. A relation may have many types.

In the composition $R\,;S$ the types of the range of R and the domain of S must match. If $R : A \sim B$ and $S : B \sim C$ then $R\,;S : A \sim C$. Similarly, if $R : T \sim T$ then $R^n : T \sim T$. Defining $R^0 = T$ for $R : T \sim T$ is reasonable since it means that $R^0\,;R^n = R^n\,;R^0 = R^n$.

The relation $+\,;\times 2$ is well typed and relates the pair (x, y) to the value $2(x + y)$ for all integers x and y. The relation $+\,;+$ is ill-typed. Note that

$$\begin{aligned}\times 2\,;\times 2^{-1} &= \mathit{int}\\ +^{-1}\,;+ &= \mathit{int}\end{aligned}$$

but that

$$\times 2^{-1}\,;\times 2 \quad<\quad \mathit{int}$$

(The ordering on relations is the subset ordering on the graphs of the relations.) Every even number is related by $\times 2^{-1}\,;\times 2$ to itself, but the odd numbers do not appear in the domain or range.

The following form, called the conjugate of R by S, appears so often that we introduce it as a new higher-order function.

$$R \setminus S \quad=\quad S^{-1}\,;R\,;S$$

Taking the inverse of each side we see that

$$\begin{aligned}(R \setminus S)^{-1} &= (S^{-1}\,;R\,;S)^{-1}\\ &= S^{-1}\,;R^{-1}\,;S\\ &= (R^{-1}) \setminus S\end{aligned}$$

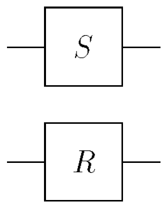

Figure 2: the parallel composition $[R, S]$

If $R : A \sim A$ and $S : A \sim C$ then $R \setminus S : C \sim A$. If in addition $Q : A \sim A$ and $S ; S^{-1} = A$ then $(Q \setminus S) ; (R \setminus S) = (Q ; R) \setminus S$. We call such an S a *representation* of A in C, and S^{-1} is the corresponding *abstraction*. The conjugate $R \setminus S$ *implements* R on the concrete values in C, and the equality $(Q \setminus S) ; (R \setminus S) = (Q ; R) \setminus S$ shows that the process of implementation preserves composition.

Our next higher-order function, parallel composition (or par), makes relations on tuples. Only pairs are used in this paper but par can be extended to longer tuples.

$$\begin{array}{rcl}
(v, w)\, [R, S]\, (x, y) & \equiv & v\, R\, x \,\&\, w\, S\, y \\
[R, S]^{-1} & = & [R^{-1}, S^{-1}] \\
[R, S] ; [T, U] & = & [R ; T, S ; U] \\
[R, S] \setminus [T, U] & = & [R \setminus T, S \setminus U]
\end{array}$$

Figure 2 shows how $[R, S]$ is drawn. Notice that $[int, int]$ is the identity on pairs of integers, and so is a type in its own right. For example, the relation $+$ has type $[int, int] \sim int$.

All of the examples given so far have been functions either from domain to range or from range to domain. However the parallel composition of a function and the inverse of a function is in general neither a function nor the inverse of one. For example

$$[+, +^{-1}] : [[int, int], int] \sim [int, [int, int]]$$

is neither a function from domain to range nor a function from range to domain. Because $+ ; +^{-1} > [int, int]$ and $\times 2^{-1} ; \times 2 < int$ the relation $[+, \times 2^{-1}] ; [+^{-1}, \times 2]$ is incomparable with $[[int, int], int]$. Beware of expecting the composition of a relation and its inverse to be an identity.

The relation $s+$ (read 'step plus') will turn out to be useful later when we consider binary arithmetic.

$$s+ \;\; = \;\; [\iota, \times 2] ; +$$

The type of $s+$ is $[int, int] \sim int$ and each of the following is true

$$\begin{array}{ccc}
(1, 0) & s+ & 1 \\
(0, 1) & s+ & 2 \\
(1, 1) & s+ & 3
\end{array}$$

The remaining higher-order functions that we will need make relations that operate on lists. We need some notation for describing lists of values, and some primitive relations for manipulating lists. We write $\langle\,\rangle$ for the empty list and $\langle x \rangle$ for the singleton list containing x. Appending of lists is written as $\frown$, so $\langle x \rangle \frown xs$ is the list whose first element is x and

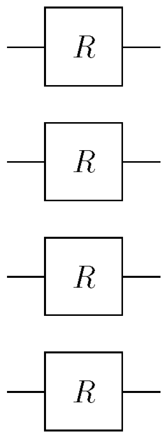

Figure 3: an instance of $\mathsf{map}\,R$

whose tail is the list xs. All the elements of a list must be of the same type. We write just n for the identity on lists of length n, for $n \geq 0$.

The higher-order function map builds a relation on lists from a relation on elements. If A is a type then $\mathsf{map}\,A$ is the type of lists of elements of type A and $\mathsf{map}\,\iota$ is the type of lists.

$$\begin{aligned} R : A \sim B &\implies \mathsf{map}\,R : \mathsf{map}\,A \sim \mathsf{map}\,B \\ \langle\,\rangle\ \mathsf{map}\,R\ \langle\,\rangle & \\ \langle x\rangle \frown xs\,(\mathsf{map}\,R)\,\langle y\rangle \frown ys &\equiv x\,R\,y\ \&\ xs\,(\mathsf{map}\,R)\,ys \end{aligned}$$

This is the relational version of the map functor that is standard in functional programming. An instance of $\mathsf{map}\,R$ is shown in figure 3, and for example

$$\begin{aligned} &\langle 4,4,4,4\rangle\ \mathsf{map}\,+^{-1}\ \langle (0,4),(0,4),(0,4),(0,4)\rangle \\ &\langle 4,4,4,4\rangle\ \mathsf{map}\,+^{-1}\ \langle (1,3),(0,4),(0,4),(0,4)\rangle \\ &\langle 4,4,4,4\rangle\ \mathsf{map}\,+^{-1}\ \langle (2,2),(0,4),(0,4),(0,4)\rangle \\ &\langle 4,4,4,4\rangle\ \mathsf{map}\,+^{-1}\ \langle (2,2),(1,3),(0,4),(3,1)\rangle \end{aligned}$$

and so on. The key point to note is that the i^{th} elements of the lists in the domain and range are related completely independently of all the other elements of the lists. Many of the properties of map derive from its pointwise nature: it interacts nicely with composition and inverse, and hence with conjugate.

$$\begin{aligned} (\mathsf{map}\,R)^{-1} &= \mathsf{map}(R^{-1}) \\ \mathsf{map}\,R\,;\mathsf{map}\,S &= \mathsf{map}(R\,;S) \\ \mathsf{map}\,R \setminus \mathsf{map}\,S &= \mathsf{map}(R \setminus S) \end{aligned}$$

The list constructors append, *app*, append-left, *apl*, and append-right, *apr*, are used to build lists

$$\begin{aligned} (xs, ys)\ app\ zs &\equiv xs \frown ys = zs \\ (x, xs)\ apl\ ys &\equiv \langle x\rangle \frown xs = ys \\ (xs, x)\ apr\ ys &\equiv xs \frown \langle x\rangle = ys \end{aligned}$$

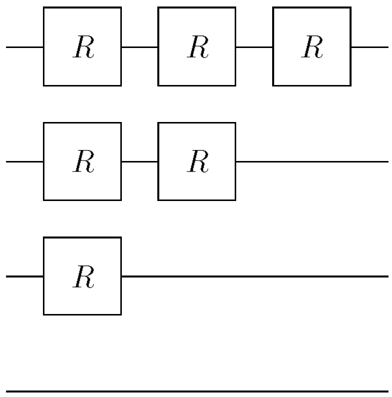

Figure 4: an instance of $\mathsf{tri}\, R$

and their inverses to take lists apart. This is an example of the convenience of having available the inverse of any 'circuit'. The inverse of *app* relates a list to each pair of lists that can be concatenated to form it. The inverse of *apl* relates a non-empty list to the pair containing the head and tail of the list.

The relation *zip* interleaves two lists

$$\begin{aligned} \langle\,\rangle \; zip \; \langle\,\rangle & \\ (\langle x \rangle \frown xs, \langle y \rangle \frown ys) \; zip \; \langle z \rangle \frown zs \quad &\equiv \quad z = (x, y) \;\;\&\;\; (xs, ys) \; zip \; zs \end{aligned}$$

for example

$$(\langle 1,2,3,4 \rangle, \langle 5,6,7,8 \rangle) \; zip \; \langle (1,5),(2,6),(3,7),(4,8) \rangle$$

Two further wiring relations are useful. They each distribute a value across a list to give a list of pairs.

$$\begin{aligned} (x, \langle\,\rangle) \; dstl \; \langle\,\rangle & \\ (x, \langle y \rangle \frown ys) \; dstl \; \langle z \rangle \frown zs \quad &\equiv \quad z = (x, y) \;\;\&\;\; (x, ys) \; dstl \; zs \end{aligned}$$

and

$$\begin{aligned} (\langle\,\rangle, y) \; dstr \; \langle\,\rangle & \\ (\langle x \rangle \frown xs, y) \; dstr \; \langle z \rangle \frown zs \quad &\equiv \quad z = (x, y) \;\;\&\;\; (xs, y) \; dstr \; zs \end{aligned}$$

for example

$$(1, \langle 2,3,4,5 \rangle) \; dstl \; \langle (1,2),(1,3),(1,4),(1,5) \rangle$$

Triangle is a pointwise higher-order function that is slightly more complicated than map. The relation $\mathsf{tri}\, R$ relates the i^{th} elements of the lists in its domain and range not by R but by R^i. Elements of lists are numbered from zero so the first elements are related by R^0.

$$\begin{aligned} R : T \sim T \quad &\Longrightarrow \quad \mathsf{tri}\, R : \mathsf{map}\, T \sim \mathsf{map}\, T \\ \langle\,\rangle \; \mathsf{tri}\, R \; \langle\,\rangle & \\ \langle x \rangle \frown xs \; (\mathsf{tri}\, R) \; \langle y \rangle \frown ys \quad &\equiv \quad x \; R^0 \; y \;\&\; xs \; (\mathsf{map}\, R \, ; \mathsf{tri}\, R) \; ys \end{aligned}$$

Relations expressed by triangle are sometimes called 'skewing' operations. The properties of tri and map depend crucially on their pointwise nature. An instance of $\mathsf{tri}\, R$ is shown in figure 4.

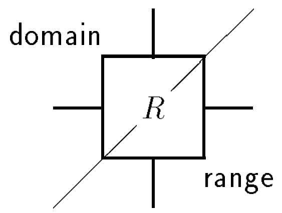

Figure 5: a picture of $R : [\iota, \iota] \sim [\iota, \iota]$

Next we introduce some higher-order functions that allow elements of lists to be combined. A convenient way to do this is to consider pair-to-pair circuits, that is circuits of type $[\iota, \iota] \sim [\iota, \iota]$, laid out according to the convention illustrated in figure 5. Two such circuits can be plugged together horizontally using the higher-order function 'beside', written $\leftrightarrow$.

$$\begin{aligned} R : [T, U] \sim [V, W] \;\&\; S : [W, X] \sim [Y, Z] \quad &\Longrightarrow \quad R \leftrightarrow S : [T, [U, X]] \sim [[V, Y], Z] \\ (t, (u, x))\, R \leftrightarrow S\, ((v, y), z) \quad &\equiv \quad \exists w.\, (t, u)\, R\, (v, w) \;\&\; (w, x)\, S\, (y, z) \end{aligned}$$

Because $R \leftrightarrow S$ is itself pair-to-pair (or more precisely $[\iota, [\iota, \iota]] \sim [[\iota, \iota], \iota]$) circuits like $(R \leftrightarrow S) \leftrightarrow T$ make sense. Note that beside is not associative, and that $(R \leftrightarrow S) \leftrightarrow T$ is different from $R \leftrightarrow (S \leftrightarrow T)$. Perhaps the simplest example of a relation built with beside is $[\iota, \iota] \leftrightarrow [\iota, \iota]$, a relation that re-brackets three values.

$$\begin{array}{ccc} (1, (2, 3)) & [\iota, \iota] \leftrightarrow [\iota, \iota] & ((1, 2), 3) \\ (1, ((2, 3), 4)) & ([\iota, \iota] \leftrightarrow [\iota, \iota]) \leftrightarrow [\iota, \iota] & (((1, 2), 3), 4) \\ (1, (2, (3, 4))) & [\iota, \iota] \leftrightarrow ([\iota, \iota] \leftrightarrow [\iota, \iota]) & ((1, (2, 3)), 4) \end{array}$$

The higher-order function row uses beside repeatedly to build a linear array, an example of which is shown in figure 6. Notice that like beside, row makes a pair-to-pair relation from a smaller pair-to-pair relation.

$$\begin{aligned} R : [T, U] \sim [V, T] \quad &\Longrightarrow \quad \mathsf{row}\, R : [T, \mathsf{map}\, U] \sim [\mathsf{map}\, V, T] \\ (t, \langle u \rangle)\, (\mathsf{row}\, R)\, (\langle v \rangle, w) \quad &\equiv \quad (t, u)\, R\, (v, w) \\ (t, \langle u \rangle \frown us)\, (\mathsf{row}\, R)\, (\langle v \rangle \frown vs, w) \quad &\equiv \quad (t, (u, us))\, (R \leftrightarrow \mathsf{row}\, R)\, ((v, vs), w) \end{aligned}$$

For example the relation $\mathsf{row}(+\, ;\, +^{-1})$ relates an (integer, list of n integers) pair to every (list of n integers, integer) pair that has the same sum, so that

$$\begin{gathered} (1, \langle 2, 3, 4 \rangle)\, \mathsf{row}(+\, ;\, +^{-1})\, (\langle 1, 2, 3 \rangle, 4) \\ (1, \langle 2, 3, 4 \rangle)\, \mathsf{row}(+\, ;\, +^{-1})\, (\langle 1, 2, 7 \rangle, 0) \\ (1, \langle 2, 3, 4 \rangle)\, \mathsf{row}(+\, ;\, +^{-1})\, (\langle 4, 3, 2 \rangle, 1) \end{gathered}$$

and so on.

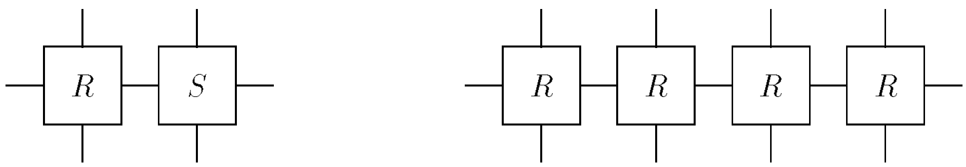

Figure 6: R beside S making $R \leftrightarrow S$, and an instance of $\mathsf{row}\, R$

If $R : [Y, \iota] \sim [\iota, Y]$ and $D \,;\, A = Y$ then

$$[A, \mathsf{map}\, B] \,;\, \mathsf{row}\, R \,;\, [\mathsf{map}\, C, D] \quad = \quad \mathsf{row}([A, B] \,;\, R \,;\, [C, D])$$

In particular, letting $D = A = Y = \iota$ gives

$$[\iota, \mathsf{map}\, B] \,;\, \mathsf{row}\, R \,;\, [\mathsf{map}\, C, \iota] \quad = \quad \mathsf{row}([\iota, B] \,;\, R \,;\, [C, \iota]) \tag{1}$$

A row of Rs can be transformed into a row of Ss using

$$[T, U] \,;\, R = S \,;\, [V, T] \quad \Longrightarrow \quad [T, \mathsf{map}\, U] \,;\, \mathsf{row}\, R = \mathsf{row}\, S \,;\, [\mathsf{map}\, V, T] \tag{2}$$

These theorems are proved by induction. To specify the length of a row, define

$$\mathsf{row}_n\, R = [\iota, n] \,;\, \mathsf{row}\, R \,;\, [n, \iota]$$

remembering that by n we mean the identity on lists of length n.

Below (written $\updownarrow$) and column (written col) are useful duals of beside and row.

$$\begin{aligned} R \updownarrow S &= (R^{-1} \leftrightarrow S^{-1})^{-1} \\ \mathsf{col}\, R &= (\mathsf{row}\, R^{-1})^{-1} \end{aligned}$$

This pattern appears often. To get the dual of a higher-order function, invert all the arguments, apply the function and invert the result. For example the dual of map is just map, but the dual of composition is reverse composition since $(F^{-1} \,;\, G^{-1})^{-1} = G \,;\, F$.

Beside, below, row and column have attractive properties.

$$\begin{aligned} (P \leftrightarrow Q) \updownarrow (R \leftrightarrow S) &= (P \updownarrow R) \leftrightarrow (Q \updownarrow S) \\ (\mathsf{row}\, R) \updownarrow (\mathsf{row}\, S) &= \mathsf{row}(R \updownarrow S) \end{aligned}$$

and by duality

$$\begin{aligned} (\mathsf{col}\, R) \leftrightarrow (\mathsf{col}\, S) &= \mathsf{col}(R \leftrightarrow S) \\ \mathsf{row}(\mathsf{col}\, R) &= \mathsf{col}(\mathsf{row}\, R) \end{aligned}$$

Of course not all interesting relations are pair-to-pair. It is necessary to be able to make continued versions of pair-to-value and value-to-pair relations also. The appropriate higher-order function is called reduce and is standard in functional programming. A pair-to-value relation can be viewed as a pair-to-pair relation composed with a selector. Similarly a pair-to-pair relation can be seen as a pair-to-value relation composed with an inverse selector. This allows reduce to be defined in terms of row. We concentrate on linear versions of reduce because they have attractive layouts on silicon. There are left- and right-handed versions of reduce, because there are two distinct linear ways of combining pair-to-value relations.

$$\begin{aligned} P : [X, Y] \sim X \quad &\Longrightarrow \quad \mathsf{rdl}\, P : [X, \mathsf{map}\, Y] \sim X \\ \mathsf{rdl}\, P &= \mathsf{row}(P \,;\, {\pi_2}^{-1}) \,;\, \pi_2 \\ \mathsf{rdr}\, P &= \mathsf{col}(P \,;\, {\pi_1}^{-1}) \,;\, \pi_1 \end{aligned}$$

The two versions of reduce correspond to different orders of association, as shown by the following examples

$$\begin{aligned} &(w, \langle x, y, z \rangle)\ (\mathsf{rdl}\, +)\ (((w + x) + y) + z) \\ &(\langle w, x, y \rangle, z)\ (\mathsf{rdr}\, +)\ (w + (x + (y + z))) \end{aligned}$$

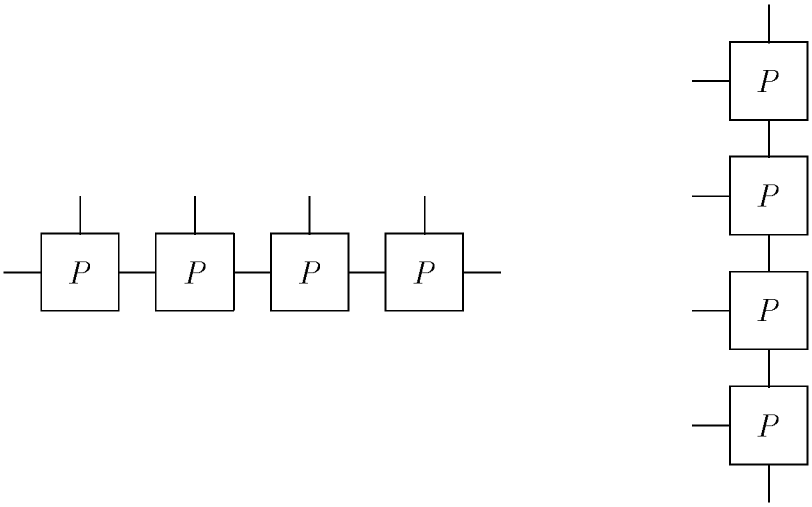

Figure 7: an instance of $\mathsf{rdl}\,P$, and one of $\mathsf{rdr}\,P$

which may help to explain their names. Instances of $\mathsf{rdl}\,P$ and $\mathsf{rdr}\,P$ are shown in figure 7. Explicitly sized versions of rdl and rdr are defined by $\mathsf{rdl}_n\,P = [\iota, n]\,;\,\mathsf{rdl}\,P$ and $\mathsf{rdr}_n\,P = [n, \iota]\,;\,\mathsf{rdr}\,P$.

Sometimes to sum the elements of a list of numbers, we will use the constant *zero* relation (defined by $x\ zero\ y \equiv x = y = 0$) to give a starting value.

$$\begin{aligned} sum &= \mathsf{map}\,int\,;\,{\pi_2}^{-1}\,;\,[zero, \iota]\,;\,\mathsf{rdl}\,+ \\ &= \mathsf{map}\,int\,;\,{\pi_1}^{-1}\,;\,[\iota, zero]\,;\,\mathsf{rdr}\,+ \end{aligned}$$

The type of *sum* is $\mathsf{map}\,int \sim int$, and by the associativity of addition

$$\begin{aligned} [int, sum]\,;\,+ &= \mathsf{rdl}\,+ \\ [sum, int]\,;\,+ &= \mathsf{rdr}\,+ \end{aligned}$$

A version of *sum* that operates only on lists of a specific length is going to prove useful: $sum_n = n\,;\,sum$.

A triangle of $\times 2$ components can be used to give weight 2^i to the i^{th} element of the list being summed. Think of a sequence of increasing weight as a 'ladder'.

$$\begin{aligned} lad &= \mathsf{tri}\times 2\,;\,sum \\ lad_n &= n\,;\,\mathsf{tri}\times 2\,;\,sum \end{aligned}$$

For example $\langle 1, 0, 1, 1\rangle\ lad\ 13$ and $\langle 2, 3, 4, 5\rangle\ lad\ 64$.

The properties of reduce can be calculated from the properties of row. Equation 1 degenerates to

$$P : [Y, \iota] \sim Y\ \&\ D\,;\,A = Y \implies [A, \mathsf{map}\,B]\,;\,(\mathsf{rdl}\,Q)\,;\,D = \mathsf{rdl}([A, B]\,;\,Q\,;\,D) \qquad (3)$$

The relationship between reduce and triangle is rather more interesting.

$$[R, R]\,;\,S = S\,;\,R \implies ((\mathsf{tri}\,R) \setminus apr^{-1})\,;\,\mathsf{rdr}\,S = \mathsf{rdr}([R^0, R]\,;\,S)$$

This could also be written using explicit sizes.

$$[R, R]\,;\,S = S\,;\,R \implies [\mathsf{tri}\,R, R^n]\,;\,\mathsf{rdr}_n\,S = \mathsf{rdr}_n([R^0, R]\,;\,S)$$

For a list of length n in the domain, the circuit on the left hand side of the equality needs $n(n-1)/2$ Rs while the circuit on the right – although it has the same behaviour – needs only n Rs.

A familiar example of this reduction from $O(n^2)$ to $O(n)$ is the application of Horner's rule to the evaluation of a polynomial, for example the evaluation of a binary number. Because $[\times 2, \times 2] \,;\, + = + \,;\, \times 2$, it follows from Horner's rule that

$$((\mathsf{tri}\times 2) \setminus apr^{-1}) \,;\, \mathsf{rdr}+ \quad = \quad \mathsf{rdr}([\times 2^0, \times 2] \,;\, +)$$

and

$$[\mathsf{tri}\times 2, \times 2^n] \,;\, \mathsf{rdr}_n + \quad = \quad \mathsf{rdr}_n([\times 2^0, \times 2] \,;\, +)$$

This last is essentially an expression of the equality

$$x_0 + 2x_1 + 2^2x_2 + \cdots + 2^n x_n \quad = \quad x_0 + 2(x_1 + 2(x_2 + \cdots + 2(x_n)\cdots))$$

telling us that a binary number can be evaluated using a right reduction. In terms of the relations already introduced

$$apr \,;\, lad \quad = \quad \mathsf{rdr}\, s+$$

and

$$[lad_n, \times 2^n] \,;\, + \quad = \quad \mathsf{rdr}_n\, s+$$

Bird uses a slightly different formulation of Horner's rule to great effect in program development [Bird88].

Horner's rule and many similar skewing theorems are special cases of a very general theorem relating row and triangle. The theorem tells us how $(\mathsf{row}\, Q)\leftrightarrow(\mathsf{row}\, R)$ and $\mathsf{row}(Q\leftrightarrow R)$ are related when Q and R satisfy certain conditions. It is discussed in reference [Jones90] where it is called the *general skewing theorem*, and will not take it any further here. An example of a consequence of the general skewing theorem is that

$$[\iota, P] \,;\, Q \,;\, [S, S] = [S, \iota] \,;\, Q \quad\Longrightarrow\quad [\iota, \mathsf{tri}\, P] \,;\, \mathsf{row}_n\, Q \,;\, [\mathsf{tri}\, S, S^n] = \mathsf{row}_n(Q \,;\, [\iota, S])$$

This can be used to prove that

$$\begin{aligned} [\iota, \mathsf{tri}\times 2] \,;\, \mathsf{row}_n(+ \,;\, +^{-1}) \,;\, [\mathsf{tri}\times 2^{-1}, (\times 2^n)^{-1}] \quad &= \quad \mathsf{row}_n(+ \,;\, +^{-1} \,;\, [\iota, \times 2^{-1}]) \\ &= \quad \mathsf{row}_n(+ \,;\, s+^{-1}) \end{aligned} \qquad (4)$$

This will prove useful in the design of arithmetic circuits. Another consequence of the general skewing theorem – this time without any triangles at all – tells us that under certain conditions a reduce-left and the inverse of a reduce-right can be combined to make a row. For example, it can be shown that $\mathsf{rdl}_n + \,;\, (\mathsf{rdr}_n +)^{-1} = \mathsf{row}_n(+ \,;\, +^{-1})$. Since $\mathsf{rdl}_n + = [\iota, sum_n] \,;\, +$ and $\mathsf{rdr}_n + = [sum_n, \iota] \,;\, +$, this can be rewritten as

$$([\iota, sum_n] \,;\, +) \,;\, ([sum_n, \iota] \,;\, +)^{-1} \quad = \quad \mathsf{row}_n(+ \,;\, +^{-1}) \qquad (5)$$

Abstraction and representation

An abstraction relates a concrete value to the abstract value that it represents. Recall that we say that $abs : C \sim A$ is an abstraction from concrete type C onto the abstract type A provided that

$$abs^{-1} \,;\, abs \quad = \quad A$$

The converse $abs^{-1} : A \sim C$ is the corresponding representation relation. All of our higher-order functions except inverse and conjugate preserve this property. So for example if a is an abstraction of type $[S, T] \sim T$, then $\mathsf{rdl}\, a$ is an abstraction of type $[S, \mathsf{map}\, T] \sim T$. In dealing with arithmetic circuits, we will be considering abstractions onto the integers and the naturals.

The relations $+$ and $s+$ are abstractions of type $[int, int] \sim int$. An example of a relation that might look like an abstraction onto the integers but is not is $[bit, \times 2\ ;\ \times 2]\ ;\ +$. Not all integers are in its range; for instance, the integer 3 is not represented; however the very similar $[int, \times 2\ ;\ \times 2]\ ;\ +$ is an abstraction onto the integers.

The list constructors, apl and apr and their inverses are all abstractions, for $apl^{-1}\ ;\ apl$ and $apr^{-1}\ ;\ apr$ are both the type of non-empty lists, $apl\ ;\ apl^{-1}$ is the type of (value, list) pairs and $apr\ ;\ apr^{-1}$ is the type of (list, value) pairs. The relation sum is an abstraction from $\mathsf{map}\, int$ to int. The relations $\mathsf{rdl} + : [int, \mathsf{map}\, int] \sim int$ and $\mathsf{rdr} + : [\mathsf{map}\, int, int] \sim int$ are both abstractions onto the integers. Because $\times 2$ is an abstraction onto the *even* integers, not the integers, $\mathsf{tri} \times 2$ is an abstraction onto lists of even integers. However $lad = \mathsf{tri} \times 2\ ;\ sum$ is an abstraction onto the integers, as are $[int, lad]\ ;\ +$, and $[int, lad]\ ;\ s+$ and $[lad_n, int\ ;\ \times 2^n]\ ;\ +$.

Restricted versions of integer abstractions are used to make abstractions onto the naturals. For example, bin relates a binary number (a list of bits, least significant bit first) to the natural number that it represents.

$$\begin{aligned} bin &= \mathsf{map}\, bit\ ;\ lad \\ bin_n &= \mathsf{map}\, bit\ ;\ lad_n \end{aligned}$$

The abstraction for a (bit-carry, n-bit binary) pair is just $[bit, bin_n]\ ;\ +$ and that for an (n-bit binary, carry-out bit) pair is $[bin_n, bit\ ;\ \times 2^n]\ ;\ +$. A carry-save number is like a binary number in which each bit is replaced by a pair of equal-weight bits, described by the abstractions

$$\begin{aligned} csv &= \mathsf{map}([bit, bit]\ ;\ +)\ ;\ lad \\ csv_n &= \mathsf{map}([bit, bit]\ ;\ +)\ ;\ lad_n \end{aligned}$$

One can also represent a number as a ladder of bit-steps.

$$\begin{aligned} scsv &= \mathsf{map}([bit, bit]\ ;\ s+)\ ;\ lad \\ scsv_n &= \mathsf{map}([bit, bit]\ ;\ s+)\ ;\ lad_n \end{aligned}$$

Because $+\ ;\ \times 2 = [\times 2, \times 2]\ ;\ +$ and $zip\ ;\ \mathsf{map} +\ ;\ sum_n = [sum_n, sum_n]\ ;\ +$, two ladders can be zipped together

$$[lad_n, lad_n]\ ;\ + \quad = \quad zip\ ;\ \mathsf{map} +\ ;\ lad_n$$

This means that one way to add two binary numbers is to interleave them and regard the result as a carry-save number

$$\begin{aligned} [bin_n, bin_n]\ ;\ + &= [\mathsf{map}\, bit\ ;\ lad_n, \mathsf{map}\, bit\ ;\ lad_n]\ ;\ + \\ &= [\mathsf{map}\, bit, \mathsf{map}\, bit]\ ;\ zip\ ;\ \mathsf{map} +\ ;\ lad_n \\ &= zip\ ;\ \mathsf{map}[bit, bit]\ ;\ \mathsf{map} +\ ;\ lad_n \\ &= zip\ ;\ csv_n \end{aligned} \tag{6}$$

Similarly

$$[bin_n, bin_n]\ ;\ s+ \quad = \quad zip\ ;\ scsv_n \tag{7}$$

and

$$\begin{aligned}[bin_n, csv_n]\ ;\ + \quad &= \quad [\mathsf{map}\ bit\ ;\ lad_n, \mathsf{map}([bit, bit]\ ;\ +)\ ;\ lad_n]\ ;\ + \\ &= \quad [\mathsf{map}\ bit, \mathsf{map}([bit, bit]\ ;\ +)]\ ;\ zip\ ;\ \mathsf{map}\ +\ ;\ lad_n \\ &= \quad zip\ ;\ \mathsf{map}([bit, [bit, bit]\ ;\ +]\ ;\ +)\ ;\ lad_n \end{aligned} \tag{8}$$

Notice that $\times$ is just as good an abstraction as $+$. The product of two binary numbers can be expressed in terms of some wiring and binary-bit products.

$$[bin_n, bin_n]\ ;\ \times \quad = \quad dstl\ ;\ \mathsf{map}([bin_n, bit]\ ;\ \times)\ ;\ lad_n \tag{9}$$

The product of a binary number and a bit can be expressed in terms of wiring and bit-bit products.

$$\begin{aligned}[bin_n, bit]\ ;\ \times \quad &= \quad dstr\ ;\ \mathsf{map}([bit, bit]\ ;\ \times)\ ;\ lad_n \\ &= \quad dstr\ ;\ \mathsf{map}\ andgate\ ;\ bin_n \\ & \quad\quad \text{where } andgate = [bit, bit]\ ;\ \times\ ;\ bit \end{aligned} \tag{10}$$

Note that the product of two bits is implemented by an 'and' gate, written *andgate*.

Designing circuits to change representations

Many signal processing circuits can be viewed as changers of representation. If $p : P \sim A$ and $q : Q \sim A$ are abstractions onto the same type A, then $p^{-1}\ ;\ (p\ ;\ q^{-1})\ ;\ q = A$ and $p\ ;\ q^{-1} : P \sim Q$ relates equivalent (concrete) values from P and Q, that is representatives of abstract values (of the type A) which are equivalent under the relation A.

This makes it easy to write down the original specification of a circuit as a changer of representation – the composition of one abstraction with the inverse of another. Theorems about the higher-order functions are then used to massage the description into something that is more easily implementable. The final circuit must use components that can be built in hardware. Such components generally perform small scale changes of representation, so the aim is to work out how to implement a large scale change of representation in terms of smaller ones.

For example, the addition of a carry-in to an weighted n-list to give a weighted n-list and carry-out is a change from the representation described by $[\iota, lad_n] ; +$ to that described by $[lad_n, \times 2^n]\ ;\ +$, so it is

$$\begin{aligned} & ([\iota, lad_n]\ ;\ +)\ ;\ ([lad_n, \times 2^n]\ ;\ +)^{-1} \\ & \quad = \{ \text{def. } lad_n \} \\ & \quad\quad [\iota, \mathsf{tri} \times 2]\ ;\ [\iota, sum_n]\ ;\ +\ ;\ +^{-1}\ ;\ [sum_n^{-1}, \iota]\ ;\ [\mathsf{tri} \times 2^{-1}, (\times 2^n)^{-1}] \\ & \quad = \{ \text{equation } 5 \} \\ & \quad\quad [\iota, \mathsf{tri} \times 2]\ ;\ \mathsf{row}_n(+, +^{-1})\ ;\ [\mathsf{tri} \times 2^{-1}, (\times 2^n)^{-1}] \\ & \quad = \{ \text{equation } 4 \} \\ & \quad\quad \mathsf{row}_n(+\ ;\ s+^{-1}) \end{aligned} \tag{11}$$

This equation expresses a large scale change of representation as a row of smaller ones. Composing $[\iota, \mathsf{map}\times 2]$ on the left of each side of the equality gives

$$\begin{aligned}
([\iota, \mathsf{map}\times 2 \,;\, lad_n] \,;\, +) \,;\, ([lad_n, \times 2^n] \,;\, +)^{-1} &= \{\text{ equation 11 }\} \\
&\quad [\iota, \mathsf{map}\times 2] \,;\, \mathsf{row}_n(+ \,;\, s{+}^{-1}) \\
&= \{\text{ equation 2 }\} \\
&\quad \mathsf{row}_n([\iota, \times 2] \,;\, + \,;\, s{+}^{-1}) \\
&= \{\text{ def. } s+ \} \\
&\quad \mathsf{row}_n(s+ \,;\, s{+}^{-1})
\end{aligned}$$

and since $\mathsf{map}\times 2 \,;\, lad_n = lad_n \,;\, \times 2$ and $[\iota, \times 2] \,;\, + = s+$

$$([\iota, lad_n] \,;\, s+) \,;\, ([lad_n, \times 2^n] \,;\, +)^{-1} \quad = \quad \mathsf{row}_n(s+ \,;\, s{+}^{-1}) \tag{12}$$

Taking inverses on both sides of equations 11 and 12 gives

$$([lad_n, \times 2^n] \,;\, +) \,;\, ([\iota, lad_n] \,;\, +)^{-1} \quad = \quad \mathsf{col}_n(s+ \,;\, +^{-1}) \tag{13}$$

and

$$([lad_n, \times 2^n] \,;\, +) \,;\, ([\iota, lad_n] \,;\, s+)^{-1} \quad = \quad \mathsf{col}_n(s+ \,;\, s{+}^{-1}) \tag{14}$$

Equations 11 to 14 represent patterns of computation that appear again and again in arithmetic circuits. To design a circuit that manipulates natural numbers or bits, we specialise by restricting the domain and range of one of these general representation changers.

Examples: adders

The addition of a carry-in bit to a binary number, giving a binary number and carry-out, matches the pattern of equation 11.

$$\begin{aligned}
&([bit, bin_n] \,;\, +) \,;\, ([bin_n, bit \,;\, \times 2^n] \,;\, +)^{-1} \\
&\quad = \{\text{ def. } bin\text{, and that } bit \text{ is a type }\} \\
&\qquad [bit, \mathsf{map}\, bit] \,;\, [\iota, lad_n] \,;\, + \,;\, ([lad_n, \times 2^n] \,;\, +)^{-1} \,;\, [\mathsf{map}\, bit, bit] \\
&\quad = \{\text{ equation 11 }\} \\
&\qquad [bit, \mathsf{map}\, bit] \,;\, \mathsf{row}_n(+ \,;\, s{+}^{-1}) \,;\, [\mathsf{map}\, bit, bit]
\end{aligned} \tag{15}$$

The next step is to push the constraints on the values in the domain and range of the circuit inside the row. If we know that the circuit is $\mathsf{row}(C)$ for some C, we can proceed to find an implementation of C. The approach is hierarchical in that we can always choose C to be itself a change of representation. The decomposition continues until we reach components that can be implemented directly in hardware.

$$\begin{aligned}
&[bit, \mathsf{map}\, bit] \,;\, \mathsf{row}_n(+ \,;\, s{+}^{-1}) \,;\, [\mathsf{map}\, bit, bit] \\
&\quad = \{\text{ equation 1 }\} \\
&\qquad [bit, \mathsf{map}\, \iota] \,;\, \mathsf{row}_n([\iota, bit] \,;\, + \,;\, s{+}^{-1} \,;\, [bit, \iota]) \,;\, [\mathsf{map}\, \iota, bit] \\
&\quad = \{\text{ equation 2 and } [bit, bit] \,;\, + \,;\, s{+}^{-1} \,;\, [bit, \iota] = [bit, bit] \,;\, + \,;\, s{+}^{-1} \,;\, [bit, bit] \} \\
&\qquad \mathsf{row}_n([bit, bit] \,;\, + \,;\, s{+}^{-1} \,;\, [bit, bit])
\end{aligned} \tag{16}$$

The justification for the last step just says that if both values in the domain and the first value of the range of $(+\,;s+^{-1})$ are bits, then the second element of the range must also be a bit. We can stop here because the component of the row, $[bit, bit]\,;+\,;s+^{-1}\,;[bit, bit]$, is just a half-adder. It relates a pair of equal weight bits to a bit-step (the sum and carry) representing the same value drawn from $\{0, 1, 2\}$.

We conclude from equations 15 and 16 that

$$\begin{aligned}&([bit, bin_n]\,;+)\,;([bin_n, bit\,;\times 2^n]\,;+)^{-1}\\ &\quad= \mathsf{row}_n\, halfadd\\ &\qquad \text{where } halfadd = [bit, bit]\,;+\,;s+^{-1}\,;[bit, bit]\end{aligned}$$

To add a bit to a carry-save number yielding a binary number and carry-out bit, we again constrain the pattern described in equation 11. The same sequence of laws is used to push the constraints inside the row.

$$\begin{aligned}&([bit, csv_n]\,;+)\,;([bin_n, bit\,;\times 2^n]\,;+)^{-1}\\ &\quad=\{\text{ def. } bin,\ csv\ \}\\ &\qquad [bit, \mathsf{map}([bit, bit]\,;+)\,;lad_n]\,;+\,;+^{-1}\,;[lad_n^{-1}\,;\mathsf{map}\, bit, (\times 2^n)^{-1}\,;bit]\\ &\quad=\{\text{ equation 11 }\}\\ &\qquad [bit, \mathsf{map}([bit, bit]\,;+)]\,;\mathsf{row}_n(+\,;s+^{-1})\,;[\mathsf{map}\, bit, bit]\\ &\quad=\{\text{ equation 1 }\}\\ &\qquad [bit, \mathsf{map}\,\iota]\,;\mathsf{row}_n([\iota, [bit, bit]\,;+]\,;+\,;s+^{-1}\,;[bit, \iota])\,;[\mathsf{map}\,\iota, bit]\\ &\quad=\{\text{ equation 2, types }\}\\ &\qquad \mathsf{row}_n([bit, [bit, bit]\,;+]\,;+\,;s+^{-1}\,;[bit, bit])\end{aligned}$$

We can stop here because the component of the row, $[bit, [bit, bit]\,;+]\,;+\,;s+^{-1}\,;[bit, bit]$, is a full-adder. It relates an equal weight pair of a bit and a pair of equal weight bits to a bit-step (sum and carry) representing the same element of $\{0, 1, 2, 3\}$. The conclusion is that

$$\begin{aligned}&([bit, csv_n]\,;+)\,;([bin_n, bit\,;\times 2^n]\,;+)^{-1}\\ &\quad= \mathsf{row}_n\, fulladd\\ &\qquad \text{where } fulladd = [bit, [bit, bit]\,;+]\,;+\,;s+^{-1}\,;[bit, bit]\end{aligned} \tag{17}$$

The circuit that adds a bit and two binary numbers to give a binary number and a carry-out bit also uses a row of full-adders.

$$\begin{aligned}&([bit, [bin_n, bin_n]\,;+]\,;+)\,;([bin_n, bit\,;\times 2^n]\,;+)^{-1}\\ &\quad=\{\text{ equation 6 }\}\\ &\qquad [bit, zip\,;csv_n]\,;+\,;([bin_n, bit\,;\times 2^n]\,;+)^{-1}\\ &\quad=\{\text{ equation 17 }\}\\ &\qquad [\iota, zip]\,;\mathsf{row}_n\, fulladd\end{aligned}$$

Both *halfadd* and *fulladd* are functions from domain to range. If $+\,;s+^{-1}$ is to be restricted to be a function from domain to range then the first component in the range must be restricted to be a bit, as indeed it is in both the full-adder and the half-adder.

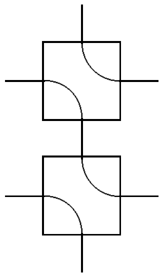

Figure 8: $[bit, bit] \updownarrow [bit, bit]$

This means that if a relation of the form $\mathsf{row}_n(+\,;s{+}^{-1})$ is to be a function from domain to range, then the first component in its range must be a binary number.

Full-adders are also useful when adding binary and carry-save numbers.

$$
\begin{aligned}
&[bin_n, csv_n]\,;+\\
&\quad = \{\,\text{equation } 8\,\}\\
&\qquad zip\,;\mathsf{map}([bit,[bit,bit]\,;+]\,;+)\,;lad_n\\
&\quad = \{\,([bit,[bit,bit]\,;+]\,;+) : [bit,[bit,bit]] \sim (s{+}^{-1}\,;[bit,bit]\,;s{+})\,\}\\
&\qquad zip\,;\mathsf{map}([bit,[bit,bit]\,;+]\,;+\,;s{+}^{-1}\,;[bit,bit]\,;s{+})\,;lad_n\\
&\quad = \{\,\text{def. } fulladd \text{ and } scsv_n\,\}\\
&\qquad zip\,;\mathsf{map}\,fulladd\,;scsv_n
\end{aligned}
\tag{18}
$$

So one can add a binary number to a carry-save using just wiring and full-adders provided one regards the output – in this case the range – as a ladder of bit-steps.

Sometimes wiring alone can implement a change of representation, for example

$$[bit,\iota] \quad = \quad [bit,\iota]\,;s{+}\,;s{+}^{-1}\,;[bit,\iota] \tag{19}$$

Similarly using $bsh\,;[\iota,+]\,;s{+} = [s{+},\iota]\,;s{+}$, where $bsh = [bit,bit] \updownarrow [bit,bit]$ is the wiring relation illustrated in figure 8, it can be shown that

$$[scsv_n, bit\,;\times 2^n]\,;+ \quad = \quad \mathsf{col}_n\,bsh\,;[bit, csv_n]\,;s{+}$$

Now composing $s{+}^{-1}\,;[bit, csv_n^{-1}]$ on the right of each side and using equation 19 yields

$$[scsv_n, bit\,;\times 2^n]\,;+\,;s{+}^{-1}\,;[bit, csv_n^{-1}] \quad = \quad \mathsf{col}_n\,bsh\,;[bit, csv_n\,;csv_n^{-1}] \tag{20}$$

Example: a multiplier

Finally to demonstrate that circuits other than adders fit into this framework, we outline the design a binary multiplier. It turns out to be easier to design a multiplier with a carry-in and carry-out in some suitable representation. For simplicity we design an n-bit by n-bit multiplier, although similar techniques would be used in an n-bit by m-bit multiplier. The initial specification is

$$([abs,[bin_n,bin_n]\,;\times]\,;+)\,;([bin_n, abs\,;\times 2^n]\,;+)^{-1}$$

where the abstraction abs has yet to be chosen. It is assumed that abs relates a fixed-length value to the naturals. The maximum carry-in or carry-out that needs to be represented is calculated by solving for c in

$$c + (2^n - 1)(2^n - 1) = (2^n - 1) + 2^n c$$

The solution is $c = 2^n - 2$, so, possible choices for abs are bin_n and csv_{n-1}, which can represent 0 to $2^n - 2$. Choosing bin_n gives the multiplier described in reference [Sheer85]. Here we will choose csv_{n-1} and design a multiplier similar to that presented in reference [McCan82]. Assuming $n > 0$, the specification is now

$$[csv_{n-1}, [bin_n, bin_n] \,;\, \times] \,;\, + \,;\, +^{-1} \,;\, [bin_n^{-1}, (\times 2^n)^{-1} \,;\, csv_{n-1}^{-1}]$$

The first two steps are the same as in all previous examples. The specification is massaged into a form that matches one of the standard patterns represented by equations 11 to 14 and then the constraints on the domain and range are pushed inside the row or column that results.

$$\begin{aligned}
&[csv_{n-1}, [bin_n, bin_n] \,;\, \times] \,;\, + \,;\, +^{-1} \,;\, [bin_n^{-1}, (\times 2^n)^{-1} \,;\, csv_{n-1}^{-1}] \\
&\quad = \{ \text{equation 9 and def. } bin \} \\
&\qquad [csv_{n-1}, dstl \,;\, \mathsf{map}([bin_n, bit] \,;\, \times) \,;\, lad_n] \,;\, + \,;\, \\
&\qquad\quad +^{-1} \,;\, [lad_n^{-1} \,;\, \mathsf{map}\, bit, (\times 2^n)^{-1} \,;\, csv_{n-1}^{-1}] \\
&\quad = \{ \text{equation 11} \} \\
&\qquad [csv_{n-1}, dstl \,;\, \mathsf{map}([bin_n, bit] \,;\, \times)] \,;\, \mathsf{row}_n(+ \,;\, s+^{-1}) \,;\, [\mathsf{map}\, bit, csv_{n-1}^{-1}] \\
&\quad = \{ \text{equation 10} \} \\
&\qquad [\iota, dstl \,;\, \mathsf{map}(dstr \,;\, \mathsf{map}\, andgate)] \,;\, \\
&\qquad\quad [csv_{n-1}, \mathsf{map}\, bin_n] \,;\, \mathsf{row}_n(+ \,;\, s+^{-1}) \,;\, [\mathsf{map}\, bit, csv_{n-1}^{-1}] \\
&\quad = \{ \text{equation 1} \} \\
&\qquad [\iota, dstl \,;\, \mathsf{map}(dstr \,;\, \mathsf{map}\, andgate)] \,;\, \\
&\qquad\quad [csv_{n-1}, \iota] \,;\, \mathsf{row}_n([\iota, bin_n] \,;\, + \,;\, s+^{-1} \,;\, [bit, \iota]) \,;\, [\iota, csv_{n-1}^{-1}]
\end{aligned}$$

Considering the sizes of the numbers that are related by $[csv_{n-1}, bin_n] \,;\, + \,;\, s+^{-1} \,;\, [bit, \iota]$ shows that any number in the second component of the range of this relation must be no bigger than $(2(2^{n-1} - 1) + (2^n - 1))/2 = 2(2^{n-1} - 1)$ and so has an $(n-1)$-bit carry-save representation. This means that

$$\begin{aligned}
&[csv_{n-1}, bin_n] \,;\, + \,;\, s+^{-1} \,;\, [bit, \iota] \\
&\quad = \quad [csv_{n-1}, bin_n] \,;\, + \,;\, s+^{-1} \,;\, [bit, \iota] \,;\, [\iota, csv_{n-1}^{-1} \,;\, csv_{n-1}] \\
&\quad = \quad [csv_{n-1}, bin_n] \,;\, + \,;\, s+^{-1} \,;\, [bit, csv_{n-1}^{-1}] \,;\, [\iota, csv_{n-1}]
\end{aligned} \tag{21}$$

so we conclude that

$$\begin{aligned}
&([csv_{n-1}, [bin_n, bin_n] \,;\, \times] \,;\, +) \,;\, ([bin_n, csv_{n-1} \,;\, \times 2^n] \,;\, +)^{-1} \\
&\quad = \{ \text{equations 2 and 21} \} \\
&\qquad [\iota, dstl \,;\, \mathsf{map}(dstr \,;\, \mathsf{map}\, andgate)] \,;\, \\
&\qquad\quad \mathsf{row}_n([csv_{n-1}, bin_n] \,;\, + \,;\, s+^{-1} \,;\, [bit, csv_{n-1}^{-1}]) \,;\, \\
&\qquad\qquad [\iota, csv_{n-1} \,;\, csv_{n-1}^{-1}] \\
&\quad = \quad [\iota, dstl \,;\, \mathsf{map}(dstr \,;\, \mathsf{map}\, andgate)] \,;\, \mathsf{row}_n\, cell \,;\, [\iota, csv_{n-1} \,;\, csv_{n-1}^{-1}] \\
&\qquad \text{where } cell = [csv_{n-1}, bin_n] \,;\, + \,;\, s+^{-1} \,;\, [bit, csv_{n-1}^{-1}]
\end{aligned} \tag{22}$$

and the next step is to find an implementation for *cell*. We have already solved a very similar problem. The addition of an $(n-1)$-bit carry-save number to an n-bit binary number can be recast as the addition of an $(n-1)$-bit carry-save number, an $(n-1)$-bit binary number and the most significant bit of the n-bit binary number.

$$\begin{array}{ll} [csv_{n-1}, bin_n] \mathbin{;} + & \\ \quad = \; w \mathbin{;} [[bin_{n-1}, csv_{n-1}] \mathbin{;} +, bit \mathbin{;} \times 2^{n-1}] \mathbin{;} + & \\ \quad\quad \text{where } (xs, ys \frown \langle y \rangle)\; w\; ((as, bs), c) \equiv as = ys \;\&\; bs = xs \;\&\; c = y & \end{array} \tag{23}$$

So *cell* can be rewritten into a form that is more suitable for implementation in hardware.

$$\begin{array}{ll} cell & = \{ \text{equation } 23 \} \\ & \quad w \mathbin{;} [[bin_{n-1}, csv_{n-1}] \mathbin{;} +, bit \mathbin{;} \times 2^{n-1}] \mathbin{;} + \mathbin{;} s{+}^{-1} \mathbin{;} [bit, csv_{n-1}^{-1}] \\ & = \{ \text{equation } 18 \} \\ & \quad w \mathbin{;} [zip \mathbin{;} \mathsf{map}\, fulladd, \iota] \mathbin{;} [scsv_{n-1}, bit \mathbin{;} \times 2^{n-1}] \mathbin{;} + \mathbin{;} s{+}^{-1} \mathbin{;} [bit, csv_{n-1}^{-1}] \\ & = \{ \text{equation } 20 \} \\ & \quad w \mathbin{;} [zip \mathbin{;} \mathsf{map}\, fulladd, \iota] \mathbin{;} \mathsf{col}_{n-1}\, bsh \mathbin{;} [\iota, csv_{n-1} \mathbin{;} csv_{n-1}^{-1}] \end{array}$$

The only part of this design that is not directly implementable is the composition $csv_{n-1} \mathbin{;} csv_{n-1}^{-1}$, which is the (concrete) type of $(n-1)$-bit carry-save representations. It is the relation that identifies any pair of $(n-1)$-bit carry-save representations of the same number – do not be misled into thinking that it is an identity! The best way to deal with it will be to push it out to the edge of the circuit. Because csv_{n-1} is an abstraction, $csv_{n-1} \mathbin{;} csv_{n-1}^{-1} \mathbin{;} csv_{n-1} = csv_{n-1}$ and so $[csv_{n-1} \mathbin{;} csv_{n-1}^{-1}, \iota] \mathbin{;} cell = cell$, just by inspection of the definition of *cell*. It follows that

$$\begin{array}{rcl} [csv_{n-1} \mathbin{;} csv_{n-1}^{-1}, \iota] \mathbin{;} cell & = & cell' \mathbin{;} [\iota, csv_{n-1} \mathbin{;} csv_{n-1}^{-1}] \\ & & \text{where } cell' = w \mathbin{;} [zip \mathbin{;} \mathsf{map}\, fulladd, \iota] \mathbin{;} \mathsf{col}_{n-1}\, bsh \end{array}$$

so by equation 2

$$[csv_{n-1} \mathbin{;} csv_{n-1}^{-1}, \iota] \mathbin{;} \mathsf{row}_{n-1}\, cell \quad = \quad \mathsf{row}_{n-1}\, cell' \mathbin{;} [\iota, csv_{n-1} \mathbin{;} csv_{n-1}^{-1}]$$

and substituting this back into equation 22

$$\begin{array}{ll} ([csv_{n-1}, [bin_n, bin_n] \mathbin{;} \times] \mathbin{;} +) \mathbin{;} ([bin_n, csv_{n-1} \mathbin{;} \times 2^n] \mathbin{;} +)^{-1} & \\ \quad = \; [\iota, dstl \mathbin{;} \mathsf{map}(dstr \mathbin{;} \mathsf{map}\; andgate)] \mathbin{;} \mathsf{row}_n\, cell' \mathbin{;} [\iota, (csv_{n-1} \mathbin{;} csv_{n-1}^{-1})^2] & \\ \quad = \; [\iota, dstl \mathbin{;} \mathsf{map}(dstr \mathbin{;} \mathsf{map}\; andgate)] \mathbin{;} \mathsf{row}_n\, cell' \mathbin{;} [\iota, csv_{n-1} \mathbin{;} csv_{n-1}^{-1}] & \end{array}$$

Again everything except the $csv_{n-1} \mathbin{;} csv_{n-1}^{-1}$ is directly implementable in hardware. We will choose not to implement this conversion between equivalent carry-save numbers, but simply to accept any carry-save representation produced by our circuit. Although the original specification was a general relation, we are happy to implement a function that gives only one of the possible outputs for any given input. The term $csv_{n-1};csv_{n-1}^{-1}$ describes exactly what this choice entails.

In the design of a real circuit there would be two final steps: first the wiring should be rearranged to make the circuit more regular, and more suitable for implementation in hardware; next the combinational circuit should be pipelined by placing latches on internal arcs.

Pipelining can be done in the same way using behaviour-preserving transformation of the design. The relations that model combinational circuits are lifted to operate pointwise on streams of data values and a single extra primitive – the unit delay element – is introduced. The laws that presented here for 'combinational' relations hold also for relations on streams [Jones88]. In particular, the skewing theorem and its derivatives can be used to calculate the effects of placing delay elements on the internal arcs of linear arrays. A triangle of delay elements corresponds to the 'time-skewing' of a list of signals that is often necessary when arrays are pipelined. We first developed skewing theorems as a way of reasoning about the design of systolic arrays. Reference [Sheer88] uses relations to describe and reason about retiming and slowdown, the two main temporal transformations used in regular array design.

Some recent work on the design and verification of multipliers can be found in references [Borr89, Chin89, Luk88, Simon89].

Conclusion

Describing both abstraction relations and circuits in a uniform relational framework works well. This is also demonstrated in reference [Mel89] which presents various kinds of abstraction for hardware verification using *Higher Order Logic*, which is also relational.

That we can use relations and their inverses allows us to start with a simple specification of a circuit as a representation changer. We then use a small set of algebraic laws to transform the specification into an implementation made of wiring and circuit primitives. The primitives are themselves small-scale representation changers. Some circuits, for example BCD to binary converters, can very obviously be described as representation changers. Others like the multiplier example above, do not immediately appear to be representation changers, but can be expressed as such when we see that multiplication is just as good an abstraction as addition.

Even in the simple examples presented here we have gone beyond the use of relations as a convenient way of describing networks of functions. We make essential use of the generality of relations.

In this paper we have only considered the binary and carry-save abstractions onto the naturals. It seems likely that our techniques will extend to other representations such as the negabinary representation of the integers as ladders of elements of {-1,0,1}. Another avenue of research opens up when we consider that abstractions need not be combinational. Once we have introduced streams and a delay element, we can describe not only how an abstract value is represented but also how the concrete value is presented to the circuit over time. In this way it should be possible to describe and design circuits with complex temporal behaviour – for example bit-serial circuits – using the techniques presented here. We plan to tackle more examples, involving more complicated representations.

ACKNOWLEDGEMENTS This research is supported by SERC grant number GR/F 28939 (Relational Programming) and by SERC/IED grant number IED2/1/1759 (High Performance VLSI DSP Architectures). Sheeran gratefully acknowledges the support of a Royal Society of Edinburgh BP Research Fellowship. Thanks to Wayne Luk, Joe Morris, David Murphy and Lars Rossen for comments on earlier drafts.

References

[deBakk86] J. W. de Bakker et al (eds.), *Mathematics and computer science*, North-Holland, 1986.

[Bhan89] A. S. Bhandal, V. Considine and G. E. Dixon, *An array processor for video picture motion estimation*, in [McCan89].

[Bird87] R. S. Bird, *An introduction to the theory of lists*, in [Broy87]. pp. 3–42. (Programming Research Group technical monograph PRG–56)

[Bird88] R. S. Bird, *Lectures on constructive functional programming*, in [Broy89]. (Programming Research Group technical monograph PRG–69)

[Borr89] D. Borrione and A. Salem, *Proving an on-line multiplier with OBJ and TACHE: a practical experience*, in [Claes89].

[Broy87] M. Broy (ed.), *Logic of programming and calculi of discrete design*, NATO advanced study institutes, Series F: Computer and systems sciences, Springer-Verlag, 1987.

[Broy89] M. Broy (ed.), *Constructive methods in computing science*, NATO advanced study institutes, Series F: Computer and systems sciences, Springer-Verlag, 1989.

[Chin89] S-K. Chin, *Verified synthesis functions for negabinary arithmetic hardware*, in [Claes89].

[Claes89] L. J. M. Claesen, *Applied formal methods for correct VLSI design*, North-Holland, 1989.

[Davis89] K. Davis and J. Hughes (eds.), *Functional programming, Glasgow 1989*, Springer Workshops in Computing, 1990.

[Jones86] G. Jones and W. Luk, *Exploring designs by circuit transformation*, in [Moore86]. pp. 91–98.

[Jones88] G. Jones and M. Sheeran, *Timeless truths about sequential circuits*, in [Tewk88]. pp. 245–259.

[Jones89] G. Jones, *Deriving the fast Fourier algorithm by calculation*, in [Davis89]. (Programming Research Group technical report PRG–TR–4–89)

[Jones90] G. Jones, *Designing circuits by calculation*, PRG technical report PRG–TR–10–90.

[Jouan85] J-P. Jouannaud (ed.), *Functional programming languages and computer architecture*, Springer LNCS 201, 1985.

[Luk88] W. Luk and G. Jones, *From specification to parametrised architectures*, in [Milne88]. pp. 267–288.

[Luk89] W. Luk, G. Jones and M. Sheeran, *Computer-based tools for regular array design*, in [McCan89]. pp. 589–598.

[McCan82] J. V. McCanny and J. G. McWhirter, *Completely iterative, pipelined multiplier array suitable for VLSI*, IEE Proc. Vol. 129, Pt. G, No. 2, April 1982. pp. 40–46.

[McCan89] J. McCanny, J. McWhirter and E. Schwartzlander (eds.), *Systolic array processors*, Prentice Hall, 1989.

[Meert86] L. G. L. T. Meertens, *Algorithmics – towards programming as a mathematical activity*, in [deBakk86].

[Meert89] L. G. L. T. Meertens, *Constructing a calculus of programs*, in [Snep89].

[Mel89] T. F. Melham, *Formalizing abstraction mechanisms for hardware verification in higher order logic*, Ph.D. Dissertation, University of Cambridge, August 1989.

[Milne88] G. J. Milne (ed.), *The fusion of hardware design and verification*, North-Holland, 1988.

[Moore86] W. Moore, A. McCabe and R. Urquhart (eds.), *Systolic arrays*, Adam Hilger, Bristol, 1986.

[Proeb87] W. E. Proebster and H. Reiner (eds.), *Proc IEEE Comp Euro 87: VLSI and computers*, Hamburg, May 1987.

[Sheer84] M. Sheeran, *muFP an algebraic VLSI design language*, in [Steel84]. pp. 104–112.

[Sheer85] M. Sheeran, *Designing regular array architectures using higher-order functions*, in [Jouan85]. pp. 220–237.

[Sheer87] M. Sheeran and G. Jones, *Relations + higher-order functions = hardware descriptions*, in [Proeb87]. pp. 303–306. (University of Glasgow Department of Computing Science technical report 87/R1)

[Sheer88] M. Sheeran, *Retiming and slowdown in Ruby*, in [Milne88]. pp. 289–308.

[Simon89] M. Simonis, *Formal verification of multipliers* in [Claes89].

[Snep89] J. L. A. van de Snepscheut (ed.), *Mathematics of program construction*, Springer-Verlag LNCS 375, 1989.

[Steel84] G. L. Steele Jr. et al (eds.), *Proc ACM Symp on LISP and functional programming*, 1984.

[Tewk88] S. K. Tewksbury, B. W. Dickinson and S. C. Schwartz (eds.), *Concurrent computations: algorithms, architecture and technology*, Plenum Press, New York, 1988.