A Relational Derivation of a Functional Program

Graham Hutton
Chalmers University of Technology,
Göteborg, Sweden
graham@cs.chalmers.se

September 15, 1992

Abstract

This article is an introduction to the use of relational calculi in deriving programs. We present a derivation in a relational language of a functional program that adds one bit to a binary number. The resulting program is unsurprising, being the standard 'column of half–adders', but the derivation illustrates a number of points about working with relations rather than functions.

1 Ruby

Our derivation is made within the relational calculi developed by Jones and Sheeran [14, 15]. Their language, called *Ruby*, is designed specifically for the derivation of 'hardware–like' programs that denote finite networks of simple primitives. Ruby has been used to derive a number of different kinds of hardware–like programs [13, 22, 23, 16].

Programs in Ruby are built piecewise from smaller programs using a simple set of combining forms. Ruby is not meant as a programming language in its own right, but as a tool for developing and explaining algorithms. Fundamental to Ruby is the use of terse notation; most formulae fit onto a single line. Having compact formulae makes it easier to pattern–match parts of a program with laws for transforming programs, and lessens the drudgery of copying unchanged parts of a formulae from one step to the next. A transformation step in Ruby is the replacement of some part of the program with another part that denotes the same relation, but is defined in a different way. Some transformation steps directly make the program more efficient in some way, many just shift parts of programs around with a view to improving efficiency later on. The calculational approach is not about being pedantic; rather it is about finding succinct and elegant arguments. Much of the art in presenting calculations is finding just the right level of detail. In the remainder of this section we give a brief introduction to Ruby.

Recall that a (binary) relation is a set of pairs. Throughout this article, letters $R, S, T, \ldots$ denote relations. We often write $x\ R\ y$ rather than $(x, y) \in R$.

The primitive relations from which Ruby programs are built are commonly just simple arithmetic and logical functions interpreted as relations. For a function $f : \mathcal{A} \to \mathcal{B}$, the corresponding 'functional relation' is defined by $\{(a, f\,a) \mid a \in \mathcal{A}\}$. Lifting of functions to relations is often left implicit in Ruby; the same symbol is used for both a function and its interpretation as a relation. Partially–applied infix functions (sections) can also be implicitly lifted; for example, $(*2) = \{(a, a*2) \mid a \in \mathcal{Z}\}$. When defining a relation R in Ruby, $a\ R\ b\ \ \widehat{=}\ \ P$ (where P is a predicate that may refer to a and b) abbreviates $R\ \ \widehat{=}\ \ \{(a,b) \mid P)\}$. If $P =$ *true*, then the "$\widehat{=}$ *true*" part is omitted.

The basic combining forms are *compose*, *par* and *converse*:

Definition 1:

$$a\ (R\ ;\ S)\ c \quad \widehat{=} \quad \exists b.\ a\ R\ b \wedge b\ S\ c,$$

$$(a,b)\ [R,S]\ (c,d) \quad \widehat{=} \quad a\ R\ c\ \ \wedge\ \ b\ S\ d,$$

$$a\ R^{-1}\ b \quad \widehat{=} \quad b\ R\ a.$$

For example,

$$(3,4)\ (+\ ;\ *2)\ 24,$$

$$(1,6)\ [+1, *2^{-1}]\ (2,3).$$

Par can be generalised to an arbitrary number of arguments; for example, we write $[R,S,T]$ for the parallel composition of three relations R, S and T.

The basic combining forms have a number of useful properties:

Lemma 2:

$$(R\ ;\ S)\ ;\ T \ = \ R\ ;\ (S\ ;\ T),$$

$$(R^{-1})^{-1} \ = \ R,$$

$$(R\ ;\ S)^{-1} \ = \ S^{-1}\ ;\ R^{-1},$$

$$[R,S]^{-1} \ = \ [R^{-1}, S^{-1}],$$

$$[R,S]\ ;\ [T,U] \ = \ [R\ ;\ T, S\ ;\ U].$$

A number of restructuring relations are used:

Definition 3:

$$a\ \textit{id}\ a,$$

$$(a,b)\ \textit{swap}\ (b,a),$$

$$(a,b)\ \pi_1\ a,$$

$$(a,b)\ \pi_2\ b,$$

$$a\ \textit{fork}\ (a,a),$$

$$((a,b),c)\ \textit{lsh}\ (a,(b,c)),$$

$$(a,(b,c))\ \textit{rsh}\ ((a,b),c).$$

Two common uses of *par* merit special abbreviations:

Definition 4:

$$\text{fst } R \;\widehat{=}\; [R, \textit{id}],$$

$$\text{snd } R \;\widehat{=}\; [\textit{id}, R].$$

The restructuring relations satisfy useful *shunting* laws, as shown below. We use the term shunting for any transformation of the form $R \;;\; S = S \;;\; T$.

Lemma 5:

$$\textit{id} \;;\; R \;=\; R \;=\; R \;;\; \textit{id},$$

$$[R, S] \;;\; \textit{swap} \;=\; \textit{swap} \;;\; [S, R],$$

$$\text{fst } R \;;\; \pi_1 \;=\; \pi_1 \;;\; R,$$

$$\text{snd } R \;;\; \pi_2 \;=\; \pi_2 \;;\; R,$$

$$R \text{ is functional} \;\Rightarrow\; R \;;\; \textit{fork} \;=\; \textit{fork} \;;\; [R, R],$$

$$[[R, S], T] \;;\; \textit{lsh} \;=\; \textit{lsh} \;;\; [R, [S, T]],$$

$$[R, [S, T]] \;;\; \textit{rsh} \;=\; \textit{rsh} \;;\; [[R, S], T].$$

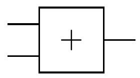
The n–fold composition of a relation R is written R^n. For example, $R^3 = R \;;\; R \;;\; R$. Note that $(R^n)^{-1} = (R^{-1})^n$. We abbreviate $(R^n)^{-1}$ by R^{-n}.

Definition 6:

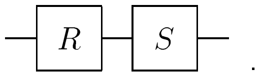
$$\begin{array}{lcl} R^0 & = & \textit{id}, \\ R^{n+1} & = & R^n \;;\; R. \end{array}$$

1.1 Pictures

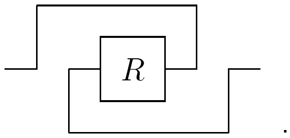
As well as denoting binary relations, Ruby terms also have a geometric interpretation as networks of primitive relations connected by wires. For example, the primitive relation $+$ (addition) can be pictured as a single node:



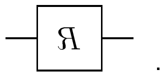
By convention, domain values are carried on the left-hand wires of a primitive, and range values on the right-hand wires. Just as tuples of values are read from left to right, busses of wires are read from bottom to top. Terms built using operators of Ruby are pictured in terms of pictures of their arguments. A term $R \;;\; S$ is pictured by placing a picture of R to the left of a picture of S, and joining the intermediate wires:



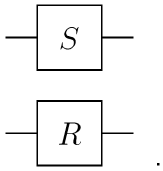
A term R^{-1} can be pictured by bending wires in a picture of R:



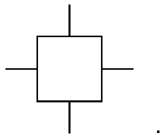
In practice, such a picture is simplified by 'pulling the domain and range wires tight', such that the picture of R flips about the vertical axis:



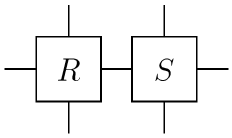
A term $[R, S]$ is pictured by placing a picture of R below a picture of S:

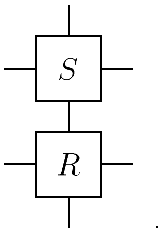


Relations from pairs to pairs can be pictured as 4–sided components:



In Ruby the left and top connections in such a picture are interpreted as the domain of the corresponding relation, and the bottom and right connections as the range. As shown below, there are two natural ways to combine relations from pairs to pairs; either place one relation beside the other, or place one below the other.





The combining forms *beside* and *below* are defined as follows:

Definition 7:

$$(a,(b,c))\ R \leftrightarrow S\ ((d,e),f) \ \ \widehat{=}\ \ \exists x.\ (a,b)\ R\ (d,x) \wedge (x,c)\ S\ (e,f),$$

$$((a,b),c)\ R \updownarrow S\ (d,(e,f)) \ \ \widehat{=}\ \ \exists x.\ (a,x)\ R\ (d,e) \wedge (b,c)\ S\ (x,f).$$

Notice that *below* is dual to *beside*:

Lemma 8: $R \updownarrow S \ = \ (R^{-1} \leftrightarrow S^{-1})^{-1}.$

1.2 Generic combining forms

A *generic* combining form is one whose first argument is a natural number that specifies which instance of a general pattern is required. The most commonly used generic combining form is *map*; two n–tuples are related by $\mathsf{map}_n\ R$ when their corresponding elements are related by R.

Definition 9: $(a_0, \ldots, a_{n-1})\ (\mathsf{map}_n\ R)\ (b_0, \ldots, b_{n-1}) \ \ \widehat{=}\ \ a_i\ R\ b_i.$

Map expresses repeated parallel composition; for example, $\mathsf{map}_4\ R = [R, R, R, R]$. It is left implicit that the variable i in the map definition ranges over naturals $< n$. Map distributes through composition, and commutes with converse:

Lemma 10: $\mathsf{map}_n\ (R\ ;\ S) \ = \ (\mathsf{map}_n\ R)\ ;\ (\mathsf{map}_n\ S).$

Lemma 11: $(\mathsf{map}_n\ R)^{-1} \ = \ \mathsf{map}_n\ R^{-1}.$

A few generic restructuring relations are used:

Definition 12:

$$(a, (b_0, \ldots, b_{n-1}))\ \ \mathit{apl}_n\ \ (a, b_0, \ldots, b_{n-1}),$$

$$((a_0, \ldots, a_{n-1}), b)\ \ \mathit{apr}_n\ \ (a_0, \ldots, a_{n-1}, b),$$

$$(a_0, a_1, \ldots, a_{n-1})\ \ \mathit{rev}_n\ \ (a_{n-1}, \ldots, a_1, a_0).$$

For example,

$$(1, (2, 3, 4))\ \mathit{apl}_3\ (1, 2, 3, 4),$$

$$((1, 2, 3), 4)\ \mathit{apr}_3\ (1, 2, 3, 4),$$

$$(1, 2, 3, 4)\ \mathit{rev}_4\ (4, 3, 2, 1).$$

They satisfy shunting laws:

Lemma 13:

$$[R, \mathsf{map}_n\ R]\ ;\ \mathit{apl}_n \ = \ \mathit{apl}_n\ ;\ \mathsf{map}_{n+1}\ R,$$

$$[\mathsf{map}_n\ R, R]\ ;\ \mathit{apr}_n \ = \ \mathit{apr}_n\ ;\ \mathsf{map}_{n+1}\ R,$$

$$\mathsf{map}_n\ R\ ;\ \mathit{rev}_n \ = \ \mathit{rev}_n\ ;\ \mathsf{map}_n\ R.$$

Similar to *map* is *triangle*; two n–tuples are related by $\mathsf{tri}_n\ R$ when their ith components are related by R^i, rather than just R as is the case for *map*:

Definition 14: $(a_0, \ldots, a_{n-1})\ (\mathsf{tri}_n\ R)\ (b_0, \ldots, b_{n-1}) \ \ \widehat{=}\ \ a_i\ R^i\ b_i.$

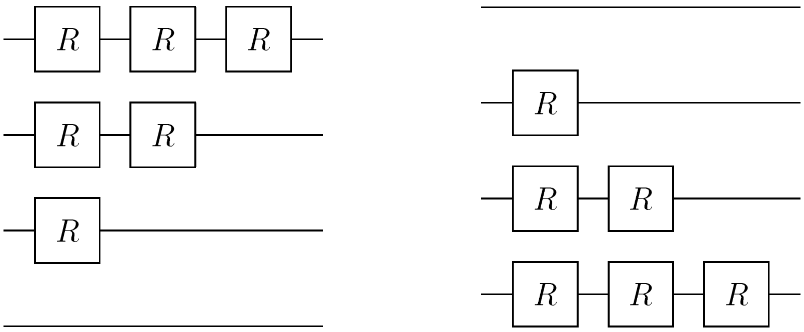


Figure 1: Pictures of $\mathsf{tri}_4\ R$ and $\mathsf{irt}_4\ R$

Triangles which grow in the opposite direction are also useful:

Definition 15: $\mathsf{irt}_n\ R \ \widehat{=}\ \mathit{rev}_n \ ; \ \mathsf{tri}_n\ R \ ; \ \mathit{rev}_n.$

See Figure 1. For example,

$$(a, b, c, d)\ \mathsf{tri}_4\ (*2)\ (a, 2b, 4c, 8d),$$

$$(a, b, c, d)\ \mathsf{irt}_4\ (*2)\ (8a, 4b, 2c, d).$$

The generic combining form *row* yields a relation that takes a value and an n–tuple to an n–tuple and a value, by stacking n copies of a relation from pairs to pairs beside one another:

Definition 16:

$$(a, (b))\ \mathsf{row}_1\ R\ ((c), d) \ \widehat{=}\ (a, b)\ R\ (c, d),$$

$$\mathsf{row}_{n+2}\ R \ \widehat{=}\ \mathsf{snd}\ \mathit{apl}_{n+1}{}^{-1} \ ; \ (R \leftrightarrow \mathsf{row}_{n+1}\ R) \ ; \ \mathsf{fst}\ \mathit{apl}_{n+1}.$$

(Choosing $n = 1$ as the base–case for $\mathsf{row}_n\ R$ avoids some technical problems with types [17].) Just as *below* is dual to *beside*, so *column* is dual to *row*. See Figure 2.

Definition 17: $\mathsf{col}_n\ R \ \widehat{=}\ (\mathsf{row}_n\ R^{-1})^{-1}.$

Combining forms *reduce left* and *reduce right* are relational versions of the familiar operators of the same name from functional programming:

Definition 18:

$$\mathsf{rdl}_n\ R \ \widehat{=}\ \mathsf{row}_n\ (R \ ; \ \pi_2{}^{-1}) \ ; \ \pi_2,$$

$$\mathsf{rdr}_n\ R \ \widehat{=}\ \mathsf{col}_n\ (R \ ; \ \pi_1{}^{-1}) \ ; \ \pi_1.$$

See Figure 3. For example, if $\oplus$ is a functional relation from pairs to values, then

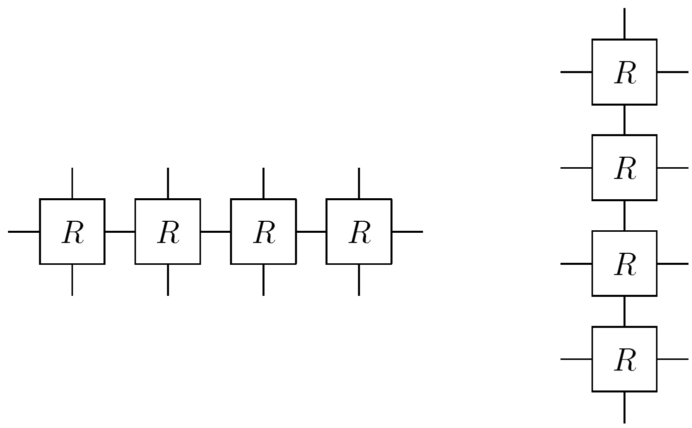


Figure 2: Pictures of $\mathsf{row}_4\ R$ and $\mathsf{col}_4\ R$

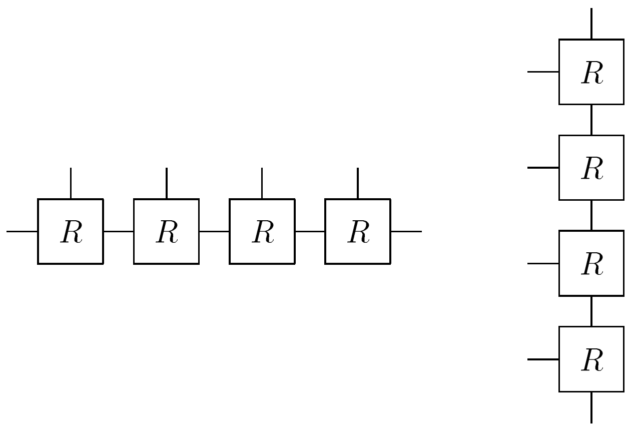


Figure 3: Pictures of $\mathsf{rdl}_4\ R$ and $\mathsf{rdr}_4\ R$

$$(a,(b,c,d)) \;\; (\mathsf{rdl}_3 \oplus) \;\; ((a \oplus b) \oplus c) \oplus d,$$

$$((a,b,c),d) \;\; (\mathsf{rdr}_3 \oplus) \;\; a \oplus (b \oplus (c \oplus d)).$$

Summing an n–tuple $(n > 0)$ of numbers merits its own abbreviation:

Definition 19: $\mathit{sum}_{n+1} \;\; \widehat{=} \;\; \mathit{apl}_n^{-1} \; ; \; \mathsf{rdl}_n \; +.$

Here are some simple properties:

Lemma 20: $\mathsf{fst}\ \mathit{sum}_n \; ; \; + \; = \; \mathsf{rdr}_n \; +.$

Lemma 21: $\mathsf{snd}\ \mathit{sum}_n \; ; \; + \; = \; \mathsf{rdl}_n \; +.$

1.3 Useful laws

The previous two sections include some laws about Ruby primitives and combining forms. In this section we give the other Ruby laws (all standard) that are used in our derivation of the binary addition program. Proofs of laws 25 to 28 can be found in some of the earlier Ruby articles; all are 'clearly true' if one thinks in terms of pictures.

The first three laws are about the primitives $+$ and $*$. The first is a dummy–free version of the standard distributivity rule. The remaining two are relational properties that are used again and again when deriving programs using Ruby.

Lemma 22: $+ \; ; \; *n = [*n, *n] \; ; \; +.$

Lemma 23: $\mathsf{fst}\ *n \; ; \; + \; ; \; *n^{-1} \; = \; \mathsf{snd}\ *n^{-1} \; ; \; +.$

Lemma 24: $+ \; ; \; +^{-1} = \mathsf{snd}\ +^{-1} \; ; \; \mathit{rsh} \; ; \; \mathsf{fst}\ +.$

The next law expresses that the argument to map can be pushed inside a column. (Of course, similar laws hold for rows, and reductions, but we don't use them here.) This law is used in the left–to–right direction, to bring together the argument of a map and the argument of a column so that they can be manipulated together.

Lemma 25: (map through column)

$$\mathsf{fst}\ (\mathsf{map}_n\ R) \, ; \mathsf{col}_n\ S \, ; \mathsf{snd}\ (\mathsf{map}_n\ T) \;\; = \;\; \mathsf{col}_n\ (\mathsf{fst}\ R \; ; \; S \; ; \; \mathsf{snd}\ T).$$

The next law expresses that a transformation of a certain form (essentially a 'shunting transformation') can be rippled through a column of components. (Again, similar laws hold for rows and reductions.) When deriving programs, this law is often used to push a type constraint upon the domain or range of a column through to the argument program.

Lemma 26: (column induction)

$$\begin{array}{ll} & \text{snd } R \ ; \ S \ = \ T \ ; \ \text{fst } R \\ \Rightarrow & \\ & \text{snd } R \ ; \ \text{col}_n \ S \ = \ \text{col}_n \ T \ ; \ \text{fst } R. \end{array}$$

The term *Horner's rule* usually refers to

$$a_n x^n + \ldots a_2 x^2 + a_1 x^1 + a_0 x^0 \ = \ (((a_n x + \ldots)x + a_2)x + a_1)x + a_0,$$

which shows how to evaluate polynomials more efficiently. Bird and Meertens have used a variant of Horner's rule to great effect in deriving functional programs [4]. The law below is a variant of Horner's rule that is used in Ruby.

Lemma 27: (Horner's rule for column)

$$\begin{array}{ll} & \text{fst } R \ ; \ S \ ; \ [T, T] \ = \ \text{snd } T \ ; \ S \\ \Rightarrow & \\ & \text{fst } (\text{irt}_n \ R) \ ; \ \text{col}_n \ S \ ; \ [T^n, \text{irt}_n \ T] \ = \ \text{col}_n \ (S \ ; \ \text{fst } T). \end{array}$$

Our final law allows a reduction and the converse of a reduction to be combined to give a column. The application of this rule is a key step in our derivation.

Lemma 28: (combining reductions)

$$\begin{array}{ll} & R \ ; \ S^{-1} \ = \ \text{snd } S^{-1} \ ; \ \textit{rsh} \ ; \ \text{fst } R \\ \Rightarrow & \\ & \text{rdr}_n \ R \ ; \ (\text{rdl}_n \ S)^{-1} \ = \ \text{col}_n \ (R \ ; \ S^{-1}). \end{array}$$

2 Binary addition

In this section we derive a Ruby program that takes an n–bit binary number $(n > 0)$ and a single bit, and gives a carry bit and an n–bit sum; the example is taken from [12]. The resulting program is unsurprising, being the standard 'column of half–adders', but the derivation illustrates a number of important points about working with relations rather than functions, which are discussed in section 3.

An n–bit binary number will be represented as an n–tuple of 0's and 1's. The leftmost value in the tuple is assumed to be the most significant. For example, the tuple $(1, 1, 0, 1, 0)$ is the representation of the decimal number 26 as a 5–bit binary number; that is, $1.2^4 + 1.2^3 + 0.2^2 + 1.2^1 + 0.2^0 = 26$.

We begin by defining a program $\textit{bin}_n$ that converts an n–bit binary number to the corresponding natural number; b abbreviates the identity relation $\{(0, 0), (1, 1)\}$ on bits. Figure 4 depicts $\textit{bin}_4$; the $\bullet$'s in this picture represent b.

Definition 29: $\textit{bin}_n \ \widehat{=} \ \text{map}_n \ b \ ; \ \text{irt}_n \ (*2) \ ; \ \textit{sum}_n.$

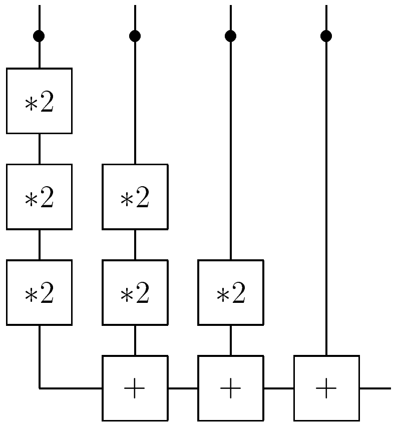


Figure 4: $\mathsf{map}_4\ b\ ;\ \mathsf{irt}_4\ (*2)\ ;\ \mathit{sum}_4$

Given below is our specification for the addition program: R converts the binary number to an integer and adds the bit, S^{-1} separates off the most significant bit of the result, and converts the remaining integer back to binary.

Definition 30: $\mathit{add1}_n\ \widehat{=}\ R\ ;\ S^{-1}$

$$R\ =\ [\mathit{bin}_n, b]\ ;\ +,$$

$$S\ =\ [b\ ;\ (*2)^n, \mathit{bin}_n]\ ;\ +.$$

There are natural reasons to regard this as a specification rather than an implementation. Firstly, even though $\mathit{add1}_n$ as a whole denotes a functional relation, there are parts of $\mathit{add1}_n$ that do not, for example $+\ ;\ +^{-1}$. Secondly, one might expect that internally the binary addition program should manipulate only 0's and 1's, whereas natural numbers are used within $\mathit{add1}_n$ as defined above. We shall calculate a program that denotes the same relation as $\mathit{add1}_n$, but whose sub–parts are all functional, and whose primitives are essentially bit–level operations.

We begin with some simple rearranging steps. (Note that we sometimes underline parts of a term within a calculation. This simple trick guides the eye to the parts being changed, and helps in pattern matching against the laws, particularly when the term properly matches the law only after some simple re–arranging. As a byproduct, many of the hints between steps become simpler too.)

$$\begin{array}{cl}
 & \mathit{add1}_n \\
= & \quad \{\ \text{def } 30\ \} \\
 & [\underline{\mathit{bin}_n}, b]\ ;\ +\ ;\ ([b\ ;\ (*2)^n, \underline{\mathit{bin}_n}]\ ;\ +)^{-1} \\
= & \quad \{\ \text{def } 29\ \} \\
 & [\mathsf{map}_n\ b\ ;\ \mathsf{irt}_n\ *2\ ;\ \underline{\mathit{sum}_n}, b]\ ;\ \underline{+}\ ;\ ([b\ ;\ (*2)^n, \mathsf{map}_n\ b\ ;\ \mathsf{irt}_n\ *2\ ;\ \underline{\mathit{sum}_n}]\ ;\ \underline{+})^{-1} \\
= & \quad \{\ \text{sums } (20{,}21)\ \}
\end{array}$$

$$[\mathsf{map}_n\ b\ ;\ \mathsf{irt}_n\ {*2}, b]\ ;\ \mathsf{rdr}_n\ +\ ;\ \underline{([b\ ;\ (*2)^n, \mathsf{map}_n\ b\ ;\ \mathsf{irt}_n\ {*2}]\ ;\ \mathsf{rdl}_n\ +)^{-1}}$$

$$=\qquad \{\ \text{converse}\ \}$$

$$[\mathsf{map}_n\ b\ ;\ \mathsf{irt}_n\ {*2}, b]\ ;\ \underline{\mathsf{rdr}_n\ +\ ;\ (\mathsf{rdl}_n\ +)^{-1}}\ ;\ [(*2)^{-n}\ ;\ b, \mathsf{irt}_n\ {*2^{-1}}\ ;\ \mathsf{map}_n\ b]$$

The right–reduction and converse left–reduction can now be combined using lemma 28. This law is used here with $R = S = +$. Under these assignments, the precondition is given by lemma 24, a property of addition much used in Ruby. We continue:

$$[\mathsf{map}_n\ b\ ;\ \mathsf{irt}_n\ {*2}, b]\ ;\ \underline{\mathsf{rdr}_n\ +\ ;\ (\mathsf{rdl}_n\ +)^{-1}}\ ;\ [(*2)^{-n}\ ;\ b, \mathsf{irt}_n\ {*2^{-1}}\ ;\ \mathsf{map}_n\ b]$$

$$=\qquad \{\ \text{combining reductions (28)}\ \}$$

$$[\mathsf{map}_n\ b\ ;\ \underline{\mathsf{irt}_n}\ {*2}, b]\ ;\ \underline{\mathsf{col}_n\ (+\ ;\ +^{-1})}\ ;\ [(*2)^{-n}\ ;\ b, \underline{\mathsf{irt}_n\ {*2^{-1}}}\ ;\ \mathsf{map}_n\ b]$$

Now the triangles can be pushed inside the column using a variant of Horner's rule, lemma 27. This law is used here with $R = {*2}$, $S = (+\ ;\ +^{-1})$, and $T = {*2^{-1}}$. We verify the precondition of the law under these assignments as follows:

$$\mathsf{fst}\ R\ ;\ S\ ;\ [T, T]\ =\ \mathsf{snd}\ T\ ;\ S$$

$$\equiv\qquad \{\ \text{assignments}\ \}$$

$$\mathsf{fst}\ {*2}\ ;\ +\ ;\ \underline{+^{-1}\ ;\ [*2^{-1}, *2^{-1}]}\ =\ \mathsf{snd}\ {*2^{-1}}\ ;\ +\ ;\ +^{-1}$$

$$\equiv\qquad \{\ \text{distribution (22)}\ \}$$

$$\mathsf{fst}\ {*2}\ ;\ +\ ;\ {*2^{-1}}\ ;\ \underline{+^{-1}}\ =\ \mathsf{snd}\ {*2^{-1}}\ ;\ +\ ;\ \underline{+^{-1}}$$

$$\Leftarrow\qquad \{\ \text{Liebniz}\ \}$$

$$\mathsf{fst}\ {*2}\ ;\ +\ ;\ {*2^{-1}}\ =\ \mathsf{snd}\ {*2^{-1}}\ ;\ +$$

$$\equiv\qquad \{\ \text{lemma 23}\ \}$$

$$\mathit{true}$$

Continuing with the *add1*$_n$ calculation:

$$[\mathsf{map}_n\ b\ ;\ \underline{\mathsf{irt}_n}\ {*2}, b]\ ;\ \underline{\mathsf{col}_n\ (+\ ;\ +^{-1})}\ ;\ [(*2)^{-n}\ ;\ b, \underline{\mathsf{irt}_n\ {*2^{-1}}}\ ;\ \mathsf{map}_n\ b]$$

$$=\qquad \{\ \text{Horner's rule (27)}\ \}$$

$$[\underline{\mathsf{map}_n\ b}, b]\ ;\ \underline{\mathsf{col}_n\ (+\ ;\ +^{-1}\ ;\ \mathsf{fst}\ {*2^{-1}})}\ ;\ [b, \underline{\mathsf{map}_n\ b}]$$

$$=\qquad \{\ \text{map through column (25)}\ \}$$

$$\underline{\mathsf{snd}\ b\ ;\ \mathsf{col}_n\ (\mathsf{fst}\ b\ ;\ +\ ;\ +^{-1}\ ;\ [*2^{-1}, b])}\ ;\ \mathsf{fst}\ b$$

Our next step is to ripple the type constraint (snd b) through the column, using column induction (lemma 26). This law is used here with $R = b$, $S = \mathsf{fst}\ b\ ;\ +\ ;\ +^{-1}\ ;\ [*2^{-1}, b]$, and $T = [b, b]\ ;\ +\ ;\ +^{-1}\ ;\ [*2^{-1}\ ;\ b, b]$. Let us verify the precondition:

$$\mathsf{snd}\ R\ ;\ S\ =\ T\ ;\ \mathsf{fst}\ R$$

$$\equiv\qquad \{\ \text{assignments}\ \}$$

$$[b, b]\ ;\ +\ ;\ +^{-1}\ ;\ [*2^{-1}, b]\ =\ [b, b]\ ;\ +\ ;\ +^{-1}\ ;\ [*2^{-1}\ ;\ b, b]$$

We verify this identity as follows. (Within the calculation below, a set is lifted to an identity relation by enclosing its elements between parenthesis $[\![$ and $]\!]$.)

$$
\begin{array}{ll}
 & \underline{[b,b] \ ; \ +} \ ; \ +^{-1} \ ; \ [*2^{-1}, b] \\
= & \qquad \{ \text{ addition } \} \\
 & [b,b] \ ; \ + \ ; \ \underline{[\![0,1,2]\!] \ ; \ +^{-1} \ ; \ [*2^{-1}, b]} \\
= & \qquad \{ \text{ converse } \} \\
 & [b,b] \ ; \ + \ ; \ ([*2, \underline{b}] \ ; \ \underline{+ \ ; \ [\![0,1,2]\!]})^{-1} \\
= & \qquad \{ \text{ shunting } \} \\
 & [b,b] \ ; \ + \ ; \ ([\underline{*2 \ ; \ [\![-1,0,1,2]\!]}, b] \ ; \ +)^{-1} \\
= & \qquad \{ \text{ shunting } \} \\
 & [b,b] \ ; \ + \ ; \ \underline{([b \ ; \ *2, b] \ ; \ +)^{-1}} \\
= & \qquad \{ \text{ converse } \} \\
 & [b,b] \ ; \ + \ ; \ +^{-1} \ ; \ [*2^{-1} \ ; \ b, b]
\end{array}
$$

Continuing with the *add1*$_n$ calculation:

$$
\begin{array}{ll}
 & \underline{\mathsf{snd}\ b \ ; \ \mathsf{col}_n\ (\mathsf{fst}\ b \ ; \ + \ ; \ +^{-1} \ ; \ [*2^{-1}, b])} \ ; \ \mathsf{fst}\ b \\
= & \qquad \{ \text{ column induction (26) } \} \\
 & \mathsf{col}_n\ \underline{([b,b] \ ; \ + \ ; \ +^{-1} \ ; \ [*2^{-1} \ ; \ b, b])} \ ; \ \mathsf{fst}\ b \\
= & \qquad \{ \text{ def 31 } \} \\
 & \mathsf{col}_n\ \mathit{HA} \ ; \ \mathsf{fst}\ b \\
= & \qquad \{ \ n > 0 \ \} \\
 & \mathsf{col}_n\ \mathit{HA}
\end{array}
$$

A *half adder* (*HA*) is a relation that gives the binary carry and sum of a pair of bits:

Definition 31: *HA* $\ \widehat{=}\ \ [b,b] \ ; \ + \ ; \ ([b \ ; \ *2, b] \ ; \ +)^{-1}.$

Lemma 32: *HA* $=$

$$
\begin{array}{l}
\{((0,0),(0,0)), \\
((0,1),(0,1)), \\
((1,0),(0,1)), \\
((1,1),(1,0))\}.
\end{array}
$$

It can be implemented in terms of functional primitives:

Lemma 33: *HA* $\ = \ [b,b] \ ; \ + \ ; \ \mathit{fork} \ ; \ [\mathit{div}\,2 \ ; \ b, \mathit{mod}\,2 \ ; \ b].$

This result follows quickly from the following:

Lemma 34: $x > 0 \quad \Rightarrow \quad ([*x, [\![0 \ldots x-1]\!]] \ ; \ +)^{-1} \ = \ \mathit{fork} ; [\mathit{div}\ x, \mathit{mod}\ x]$.

This completes the derivation of the binary addition program, which is now in the form of an implementation. In summary, we have made the following transformation:

$$
\begin{array}{ll}
 & \mathit{add1}_n \\
= & \qquad \{ \text{ by definition } \} \\
 & [\mathit{bin}_n, b] \ ; \ + \ ; \ ([b \ ; \ (*2)^n, \mathit{bin}_n] \ ; \ +)^{-1} \\
= & \qquad \{ \text{ by calculation } \} \\
 & \mathsf{col}_n \ ([b, b] \ ; \ + \ ; \ \mathit{fork} \ ; \ [\mathit{div}\, 2 \ ; \ b, \mathit{mod}\, 2 \ ; \ b])
\end{array}
$$

3 Discussion

We have shown how the relational language Ruby can be used to derive a simple functional program. In this final section we stand back from this specific example and make some comments about the use of relational calculi in deriving programs.

3.1 Why relations?

The addition program that we have derived is a functional program, in that one could define functional versions of the Ruby combining forms in a language such as ML, and execute the addition program. What has been gained in deriving the program within a relational language? One answer is that during the derivation we were able to make transformations that resulted in some sub–parts of the program being non–functional; for example, $\mathsf{rdr}_n\ +\ ;\ (\mathsf{rdl}_n\ +)^{-1}$ is very much a relation. (Of course, since all our transformation steps are equalities, the program as a whole denotes a functional relation at all stages throughout the derivation.) One might think of relations in this context as being 'imaginary functions' that are useful during the derivation process. Particularly useful in specifying and manipulating programs is the converse operator for relations; only injective functions have an 'inverse', but every relation has a converse. For example, even though in the specification $R\ ;\ S^{-1}$ for the addition program the function S is injective, and hence has an inverse, it is only by treating S as a relation and being able to distribute the converse operator (contravariantly) through S that progress is made.

Another reason to generalise from functions to relations is to allow non–deterministic programs; functional programs produce at most one output for each input, relational programs can produce an arbitrary number. A central topic of [12] is the derivation of relational programs that are non–determinstic in a very structured way, being able to be expressed as the union of disjoint products of sets; such relations are known as 'difunctional' relations. Equivalently, the difunctionals are precisely those relations that can be expressed as the composition of a functional relation and the converse of a functional relation. A great many programs can be specified as such a composition; such programs have been called 'representation changers', converting an 'abstract' value from one 'concrete' representation to another concrete representation [15, 12]. Refinement of a program specified in the form $f \ ; \ g^{-1}$ proceeds by sliding parts of f and g^{-1} through one another, aiming towards a new program with components that are representation changers with smaller abstract types. In this sense, 'thinking about types' guides refinement. The process is repeated until the remaining representation changers can be implemented directly using a few standard primitives. It is encouraging to find that the same patterns of transformations are used again and again when calculating with representation changers. The *add1*$_n$ program is a simple example of a representation changer; in this case refinement stops after one iteration, when we find that the half–adder *HA* can be implemented directly.

Relations $\mathcal{A} \sim \mathcal{B}$ are in one–to–one correspondence with functions $\mathcal{A} \rightarrow P\mathcal{B}$, where P here denotes the power–set operator. Why then not just stay within a functional paradigm, for example the Bird–Meertens formalism (sometimes called *Squiggol*) [4], and admit sets as a type? Our answer is that even though relations and set–valued functions are of equivalent expressive power, the algebra of relations is much cleaner than the algebra of set–valued functions; compare our relational calculations with the functional calculations in [18]. Generalising from functions to relations brings many advantages. Are there yet more general calculi with even more advantages? In [19] de Moor observes that functions $P\mathcal{A} \rightarrow P\mathcal{B}$ (predicate transformers [8]) generalise relations $\mathcal{A} \sim \mathcal{B}$ in the same way that such relations generalise functions $\mathcal{A} \rightarrow \mathcal{B}$. There is a wealth of work in the use of predicate transformers in program calculation; it would be interesting to compare with the use of binary relations. Perhaps there are advantages in making yet another jump and working with relations $P\mathcal{A} \sim P\mathcal{B}$?

3.2 Types

In relational calculus it is common to write $R \subseteq \mathcal{A} \times \mathcal{B}$ (where $\mathcal{A}$ and $\mathcal{B}$ are sets) in the form of a typing judgement $R \in \mathcal{A} \sim \mathcal{B}$. We can make a 'point–free' version of this definition by working with identity relations $A, B \subseteq$ *id* as types:

$$R \in A \sim B \ \equiv \ A \ ; R \ = \ R \ = \ R \ ; B.$$

We say in such a case that A is a 'left domain' of R, and B a 'right domain'. An extensive exploration of this approach is given by Backhouse et al [1]. There are advantages to being more general and adopting equivalence relations as types rather than identity relations [11, 12, 15, 27, 28]. Sometimes the precondition of a Ruby law that is applied during a calculation works out to be an assertion about types; see for example the use of column induction in the *add1*$_n$ derivation, in which the precondition works out to the assertion that the identity relation fst b is a right domain of some program. For simple examples like the *add1*$_n$ derivation, no special machinery is needed to verify assertions about types. For more involved examples however, 'domain operators' (which yield least domains under an appropriate ordering for the kind of types) and their associated calculus have proved fundamental in verifying assertions about types [12]. Indeed, prior to the use of these operators, many assertions about types in Ruby derivations were verified either informally, or outwith the algebraic setting of the Ruby calculus.

3.3 Related work

The basic theory of binary relations was developed by Peirce, around 1870. Schröder extended the theory in a very thorough and systematic way around 1895. Tarksi's aim in writing his well–known article [24] in 1941 was to "awaken interest in a certain neglected logical theory", saying that "the calculus of relations deserves much more attention than it receives", having "an intrinsic charm and beauty which makes it a source of intellectual delight to all who become acquainted with it."

Modern introductions to relational algebra that may be of interest to calculational programmers are given by Dijkstra [7] and van Gasteren and Feijen [9]. Schmidt and Ströhlein have recently published a textbook on relational algebra [20]. Backhouse, Voermans and van der Woude [1] extend the work of Tarski in axiomatising the relational calculus, developing their *spec calculus* as a framework for deriving programs; the group have developed a wealth of theory, but have not yet started to experiment with derivations. Haeberer and Veloso [26] derive a number of simple programs using a relational language. Berghammer [3] proposes the use of relational algebra to specify types and programs; the idea of characterising types by a number of relational formulae is also explored in Desharnais' thesis [6]. Haeberer and Veloso [25] have returned to Tarski's question of how the relational calculus might be extended to gain the expressive power of first–order classical logic. An extensive categorical treatment of binary relations is given by Freyd and Scedrov in their recent book [10]; other relevent categorical work includes that of Barr [2], and Carboni, Kasangian and Street [5]. In his D.Phil. thesis [19] de Moor shows how categorical results about relations can be used in solving 'dynamic programming' problems.

Acknowledgements

This research was funded by SERC projects 'relational programming' and 'structured parallel programming'. Thanks to Carolyn Brown, Clare Martin and Mary Sheeran for comments.

References

[1] Roland Backhouse, Ed Voermans and Jaap van der Woude. *A relational theory of datatypes.* Proc. EURICS Workshop on Calculational Theories of Program Structure, Ameland, The Netherlands, September 1991.

[2] Michael Barr. *Relational Algebras*. Reports of the the Midwest Category Seminar IV, Lecture Notes in Mathematics, vol 137, Springer–Verlag, 1970.

[3] Rudolf Berghammer. *Relational specification of data types and programs.* Universitat der Bundeswehr Munchen, Report 9109, September 1991.

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

[5] Aurelio Carboni, Stefano Kasangian, and Ross Street. *Bicategories of spans and relations.* Journal of Pure and Applied Algebra 33 (1984) 259–267.

[6] J. Desharnais. *Abstract relational semantics.* Ph.D. thesis, McGill University, Montreal, 1989.

[7] Edsger W. Dijkstra. *A relational summary.* (EWD1047)

[8] Edsger W. Dijsktra and Caroll Scholten. *Predicate calculus and program semantics.* Springer–Verlag, 1990.

[9] Wim Feijen and Netty van Gasteren. *An introduction into the relational calculus*. Eindhoven University of Technology, 1991. (AvG91/WF140)

[10] Peter Freyd and Andre Scedrov. *Categories, Allegories.* North–Holland, 1990.

[11] Graham Hutton and Ed Voermans. *A calculational theory of pers as types*. Glasgow University Research Report 1992/R1.

[12] Graham Hutton. *Between functions and relations in calculating programs*. Ph.D. thesis, Glasgow University, 1992. (To appear)

[13] Geraint Jones and Mary Sheeran. *Timeless Truths About Sequential Circuits.* Concurrent Computations: Algorithms, Architectures and Technology (ed. Tewksbury et al), Plenum Press, New York 1988.

[14] Geraint Jones. *Designing circuits by calculation.* Oxford University, April 1990. (PRG–TR–10–90)

[15] Geraint Jones and Mary Sheeran. *Relations and refinement in circuit design.* Glasgow University, February 1992.

[16] Geraint Jones and Mary Sheeran. *Designing arithmetic circuits by refinement in Ruby.* Glasgow University, February 1992.

[17] Geraint Jones. *A certain loss of identity*. Draft Proceedings, 1992 Glasgow Workshop on Functional Programming. To appear in Springer Workshops in Computing.

[18] Oege de Moor. *Indeterminacy in optimization problems*. Proc. Summer School on constructive algorithmics, Ameland, The Netherlands, September 1989.

[19] Oege de Moor. *Categories, Relations and Dynamic Programming*. D.Phil. thesis, Oxford University, April 1992. (PRG–98)

[20] G. Schmidt and T. Ströhlein. *Relationen und Grafen.* Springer–Verlag, 1988.

[21] Mary Sheeran. *Describing and Reasoning about Circuits Using Relations* Proc. Workshop in Theoretical Aspects of VLSI (ed. Tucker et al), Leeds 1986.

[22] Mary Sheeran. *Retiming and Slowdown in Ruby.* The Fusion of Hardware Design and Verification (ed. Milne), North–Holland 1988.

[23] Mary Sheeran. *Describing Butterfly Networks in Ruby*. Proc. 1989 Glasgow Workshop on Functional Programming, Springer Workshops in Computing.

[24] Alfred Tarski. *On the calculus of relations.* Journal of Symbolic Logic, 6(3):73–89, September 1941.

[25] Paulo Veloso and Armando Haeberer. *A finitely relational algebra for classical first-order logic.* (Presented at 9th international congress of Logic, Methodology and Phylosophy of Science, Sweden.)

[26] Paulo Veloso and Armando Haeberer. *Partial relations for program derivation.* Proc. IFIP WG–2.1 Working Conference on Constructing Programs from Specifications, USA, May 1991.

[27] Ed Voermans. *A relational theory of datatypes.* (working title) Ph.D. thesis, Eindhoven University of Technology, 1992. (To appear)

[28] Ed Voermans and Jaap van der Woude. *A relational theory of datatypes: the per version.* Eindhoven University of Technology, January 1992.