Preliminary Notes
Wires and reversible languages

Luca Roversi, Luca Paolini, and Armando B. Matos
October 2017

Contents

Notes.
– File: `wire.tex`
– Reference in `bib.bib`: `matos-wires`

Abstract

In this preliminary note we analyse a natural correspondence between the wires of a reversible classical digital circuit and the registers of a SRL-like program. For instance, during a computation, neither the wires nor the names of the registers change. Only their contents can be modified. We review some of the Category Theory background, which is used to define the category $\mathbf{FCC}^{\simeq}$, see for instance [GA08, YY09, Sel11].
The categorical formulation of SRL-like programs is not included in this note; is only mentioned an the end, as an open task. In the appendices (starting in page 19) we include several transcriptions.
The questions studied in this note are part of a family of more general correspondences between Logic, Proof Theory, Type Theory, Computation and Physics, see for instance [Sel11, BS09, Bae06, Sob17].

1 Introduction

In a programming language the interplay between the name of a register and the value it contains is interesting... and a source of many confusions. The situation is similar for digital (or other forms of) circuits, where a "wire" corresponds to a register[1]. During the execution of a program — or during "a computation" of a digital circuit — the content of a register (or of a wire) may, of course, change.

Wires are an important part of both circuits (digital or not) and of diagrams corresponding to certain programs. Those diagrams (and circuits) are essentially made of wires and computation blocks.

About wires

1. Individuality.
   Each wire is unique. It never merges with another wire nor divides (forks) in two or more wires.
2. Contents.
   Each wire transports some information, which may be a bit, an integer, or other something else.
3. Change of contents.
   The information carried by a wire may change: (i) in a circuit: as the "signals" propagate from left to right (say), (ii) in a program: as the program is executed.
4. Time in a program = space in a circuit
   More precisely: the evolution of a computation (of a program) may be specified by certain functions of *time* (contents of the registers, program counter...), while the computation of a circuit is characterised by the propagation of information *along* the wires.
5. What is a wire?
   It depends on the case. For digital circuits it may be a metallic conductor. For the language SRL it corresponds to a programming register.
6. Computation blocks.
   Each block has a certain arity, say n. The input is an ordered sequence of n wires, and the output is the same ordered sequence of wires, usually containing different values.
7. No circularity in the overall diagram.
   The overall diagram is like a block in the sense that it has the same wires at the input and at the output, by the same order.

In SRL programs the use of wire diagrams is particularly interesting: no cloning, no new registers are ever created, the input and output registers are the same:

[1] Given this correspondence, we will sometimes call "wires" to program registers.

"wires" seem to have some reality.

When looking at wire diagrams, or simply "diagrams", we should distinguish between (i) the bents of a wire exist only for the convenience of drawing the circuit and (ii) the bents of a wire correspond to a permutation of the values stored in the wires. The later case *can only occur inside a block* because the information carried by the wires is modified.

Figure 1, page 6, illustrates the difference between cases (i) and (ii). Two other examples of the case (i) – drawing convenience – are shown in Figures 2 (page 7) and 3 (page 8).

Observation. Figure 1 suggests a possible duality between

(i) Change of names: a rename, denoted by ρ, and

(ii) Change of contents: done inside a *block* (or program) π that permutes the contents of the registers. The simplest non-trivial case is a swap.

Under a "black box" view, we would have, for instance $\pi_{\mathsf{swap}(a,b)} \equiv \rho_{a \leftrightarrow b}$. □

2 Composing blocks

This section is essentially transcribed from [Mat17b].
We describe some forms of combining two or more programs in a single program. Due to the reversibility, some common forms of program composition are not allowed; these include, for instance, the compositions that involve the "connection" of one output register of some program (or part of a circuit) to two or more inputs registers of other programs ("cloning" not allowed); this kind of connection would in general result in a program (or circuit) that does not implement a bijective transformation.

It is possible, and eventually it may be advantageous, to describe various forms of wire composition (or of program composition) in terms of Category Theory, see for instance [GA08, YY09] and the citations therein.

2.1 Series and parallel composition

Let A and B be two blocks that use respectively the wire sets $R_a \cup R_c$ and $R_b \cup R_c$, where $R_a \cap R_b = \emptyset$. Figure 4, page 9, illustrates a form of composing A with B that includes both *parallel* and *series* composition.

In fact, the situation is a bit more complex, because when composing blocks, we

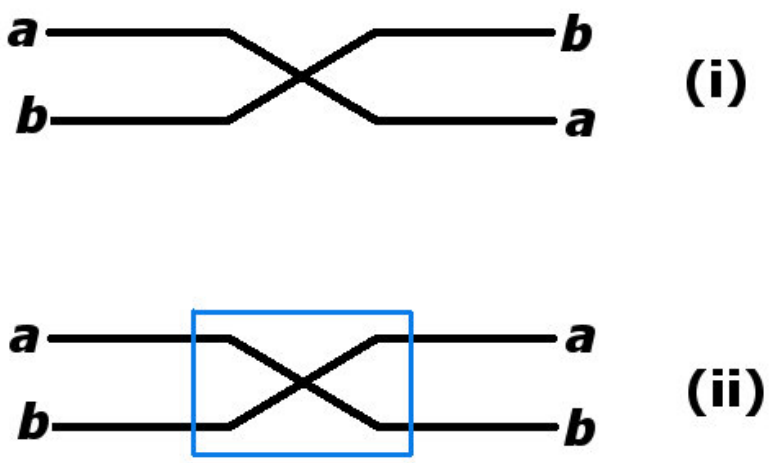


Figure 1: There is a difference between cases (i) and (ii). In case (i) we have a mere way of drawing the wires: wire a and b exchange vertical coordinates. There is no modification whatsoever — in other words, the transformation is the identity, $a' = a_0$, $b' = b_0$ (equivalent to two parallel lines).
In case (ii) there is a block (marked as a blue rectangle) that implements a transformation: $a' = b_0$, $b' = a_0$. In summary, (i) is the identity and (ii) is a swap.

are free to select and permute wires (of course without cloning). See the text of Figure 4.

$$\begin{cases} R_c = \emptyset & \text{parallel composition} \\ R_a = R_b = \emptyset & \text{series composition} \end{cases}$$

2.2 Extension and projection

The set of wires (registers) used by a program may be *extended*, as exemplified in Figure 5, page 10. The extension operation includes the *projection* which consists in removing some registers from the program; clearly, it is only possible to "remove a register from a program" when the program does not mention it.

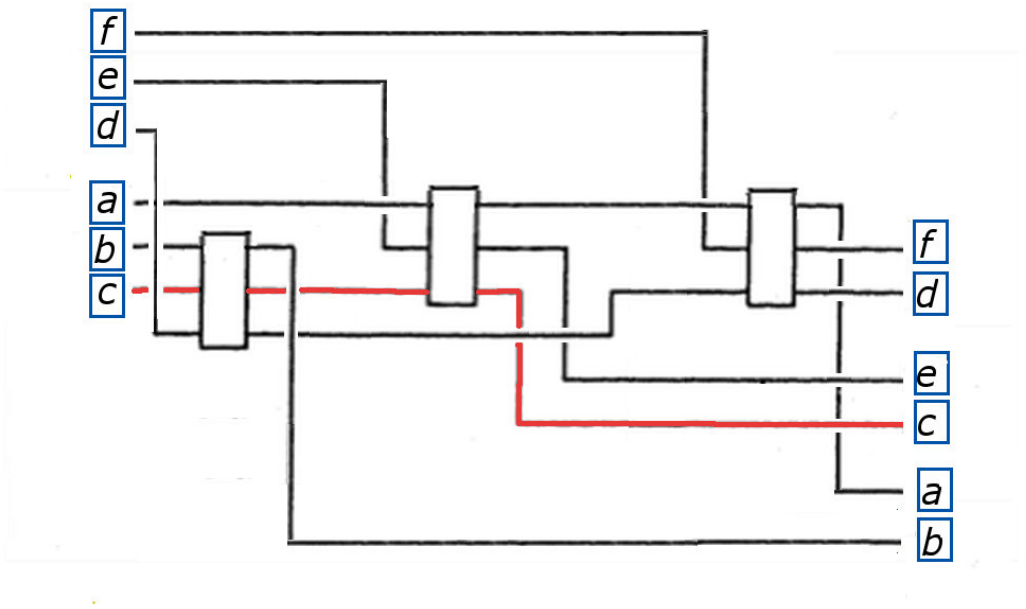


Figure 2: From [FT82]: in this example the names of the six language registers (or "wires") are placed inside blue boxes. They are $a,\ldots,f$. An example of a path with bents is shown in red; it corresponds to register c. This diagram represents a "1-line-to 4-line demultiplexer". See [Mat17a] for more details.

2.3 Replacement of a part of a program by another program

We mention another form of composing the blocks A and B which consists in *replacing* some sequence A' of instructions of A by B. This replacement is only possible when the following sets are disjoint

- The set of loop registers whose scope includes the replaced sequence of instructions A'.
- The set of registers modified (at some step of a computation) by B.

For instance, in "$A = \mathsf{for}\, a(\mathsf{inc}\, b;\ \mathsf{dec}\, c)$" we can not replace the instruction "$A' = \mathsf{dec}\, c$" by "$B = \mathsf{for}\, b(\mathsf{inc}\, a)$".

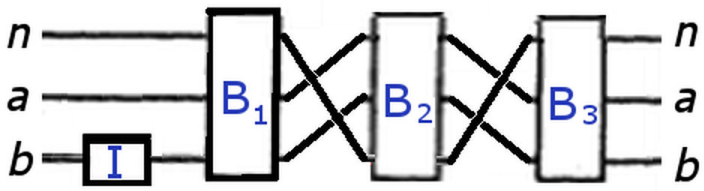


Figure 3: From [MRP17]: the blocks B_1, B_2, and B_3 are identical, each implements the transformation $Q(x, y, z)$. The wires are bent so as to "implement" the composition "$Q(n, a, b)$; $Q(a, b, n)$; $Q(n, a, b)$", where $Q(n, a, b)$ denotes "$\mathsf{for}\, n(\mathsf{for}\, b(\mathsf{inc}\, a);\ \mathsf{for}\, a(\mathsf{inc}\, b))$" and the block I increments b by 1. (For $a_0 = b_0 = 0$ and $n \geq 0$ the final values of n, a and b are greater than $2^{2^{2^n}}$.) All the bents were introduced for the convenience of the drawing.

2.4 Register renaming

Given a set of blocks composed as described above, we can *rename* the wires. This renaming must be done globally, that is, it should affect all the composed programs.

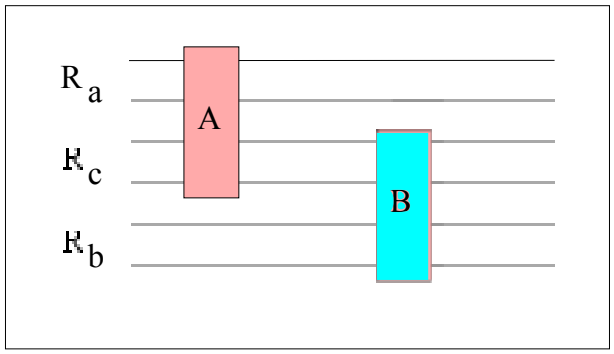


Figure 4: A method of composing two programs or circuits; the horizontal lines represent registers or "wires", respectively. It is assumed that $R_a \cap R_b = \emptyset$. This scheme includes both the parallel (when $R_c = \emptyset$) and the series (when $R_a = R_b = \emptyset$) forms of composition.
In fact the admissible connections from block A to block B are more complex: a *selection* of the wires of the block A is connected to a selection of the wires of the block B. Thus, a (fixed) selection of wires and a (fixed) permutation of wires — or equivalently, two fixed selections — are involved.

3 Drawing, bents, and breaks

During the execution of a program, a register has some form of "identity". We can, for instance, observe the contents of a particular register as a function of time. Moreover a register has a *name*. Similarly, each "wire" of a circuit (digital or not) has an "identity".

When drawing diagrams that represent SRL programs[2], the following conditions are desirable.

- Each register is represented in an horizontal line.
- Let a "block" be a recognisable part of a program or circuit. Do not break the blocks. More of this later...

If these conditions are satisfied, it may be easier to find opportunities for the detection of parallelism and for the application of "series/parallel" transformations.

But, of course, that may be not possible.

[2]But is this possible in general? How to represent for instructions?

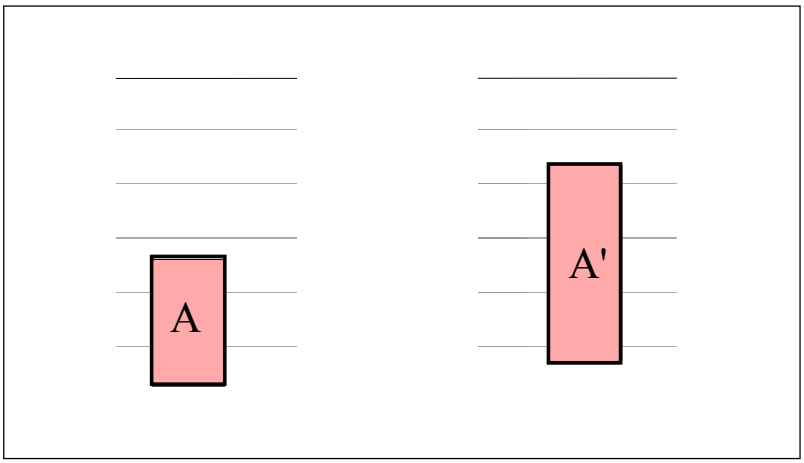


Figure 5: The program (or circuit) on the right was obtained from the program (or circuit) on the left by the "extend" transformation. The extension can also be a projection. This case happens when the program (or "block") does not mention some registers and they are "removed"; for instance, A is a projection of A'.

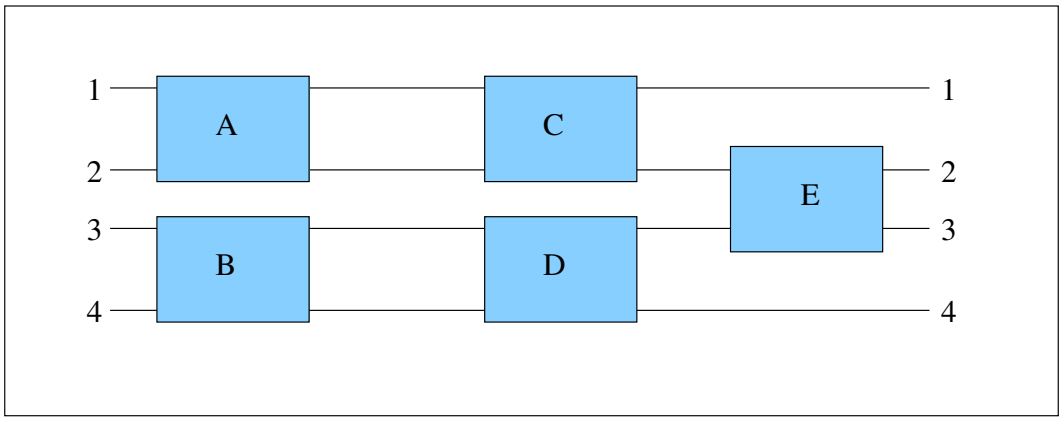


Figure 6: In this case a neat representation (of a SRL program) is possible: each wire (or program register) is associated with an horizontal line (1, 2, 3, or 4), and the blocks (A, B, C, D, and E) were not broken. Compare with Figures 7 (page 12) and 8 (page 13).

3.1 Drawing: bending the wires so as not to break the blocks

In Figure 7 the wires have been bent and the blocks are (vertically) continuous, that is, not "broken".

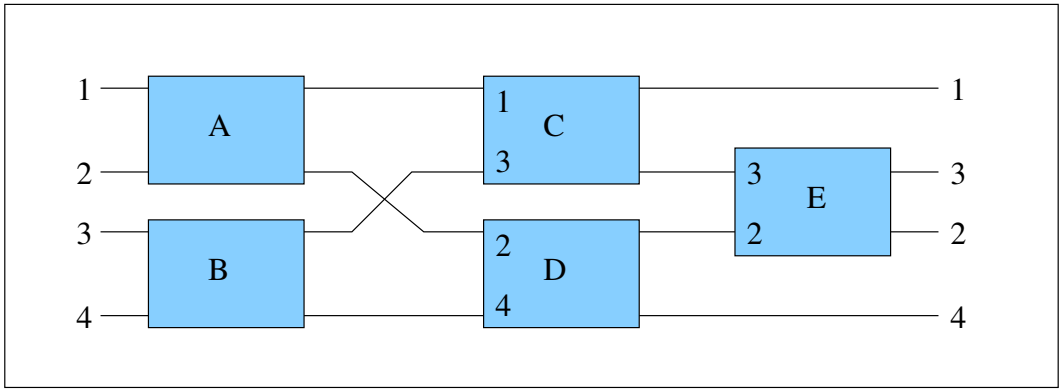


Figure 7: To avoid breaking the blocks (A, B, C, D, or E) there is a crossing between the wires (or program registers 2 and 3), so that, for instance, the second line (from the top) is used by two registers, 2 and 3.

3.2 Drawing: breaking the blocks to keep the wires on the same horizontal line

In Figure 8, which represents the same SRL program as Figure 7, the blocks were broken so that each wire remains on the same horizontal line.

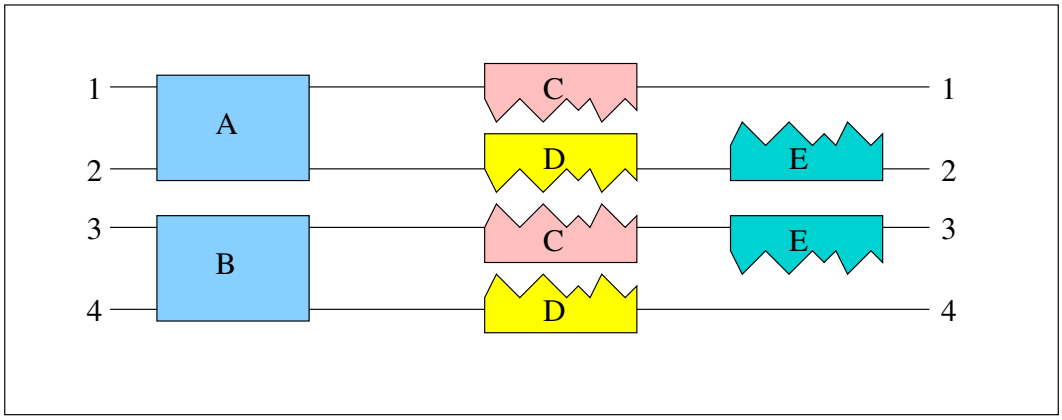


Figure 8: In order to keep each wire (or program register) in the same horizontal line (1, 2, 3, and 4), the blocks C, D, and E are broken in two parts.

3.3 SRL programs are trees: how to represent the Fibonacci program?

Due to the loop (or for) instructions, a SRL program has a tree-like structure, so that it may be not possible to represent faithfully the program as a wire diagram.

Figure 9 represents the program

$$\mathsf{for}\, n(\mathsf{for}\, b(\mathsf{inc}\, a);\ \mathsf{for}\, a(\mathsf{inc}\, b))$$

in a single block F (top of Figure), while the body of the outer for loop, namely "$X; X \stackrel{\text{def}}{=} \mathsf{for}\, b(\mathsf{inc}\, a);\ \mathsf{for}\, a(\mathsf{inc}\, b)$", is represented in the lower part of the Figure.

While "$X; X$" can be divided in 2 blocks in sequence, the block F can not be divided in series or parallel blocks.

An alternative, possibly not of general use, is to represent the whole program in a single diagram, using some ad hoc graphical notation, see Figures 10 and 11.

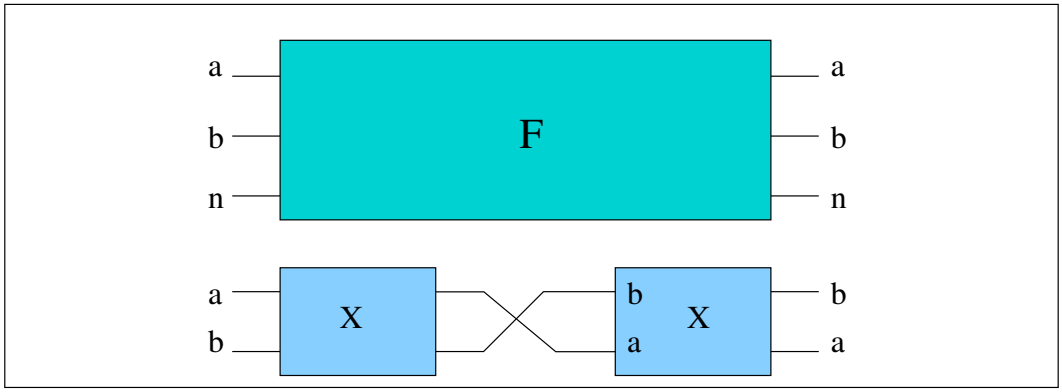


Figure 9: Top: the SRL program "for n(for b(inc a); for a(inc b))". is represented by a single block. Bottom: a representation of the body of the n loop "$X; X \stackrel{\text{def}}{=}$ for b(inc a); for a(inc b)". The two X's represent two identical SRL programs, but the inputs of the right X have been swapped – no operation, only a drawing convenience.

4 Recovering inputs and initial values

We describe a technique based on applying the inverse of a SRL program in order to recover most of the inputs at the output. A similar technique has been used in [FT82]. It can also be applied in other reversible register languages.

Notation

Program registers are distinguished by lower right index, x_i. The tuples "$x_1, \ldots, x_n$" and "$0, \ldots, 0$" (m zeros, the initial contents of $x_{n+1}, \ldots, x_{n+m}$) are denoted by $\overline{x}$ and $\overline{0}$, respectively. The final contents of register y in a computation $P(\ldots)$ is denoted by $P(\ldots)|_y$.

Computing a function with the help of 0's

Suppose that the function $f(x_1, \ldots, x_n)$ can be obtained as the final contents of some register y of a SRL program P. The function f is necessarily primitive recursive. Assume that P uses also $m \geq 1$ other registers, say $y_{n+1}, \ldots y_{n+m}$, which are initialised with zero. Moreover, and without loss of generality, assume that y is the register x_{n+m}. That is,

$$f(x_1, \ldots, x_n) = P(x_1, \ldots, x_n, \underbrace{x_{n+1}, \ldots x_{n+m}}_{\text{initially all 0}})|_y,$$

or, in a more compact form $f(\overline{x}) = P(\overline{x}, \overline{0})|_y$. Here, $\overline{x}$ and $\overline{0}$ denote the initial contents of the corresponding registers.

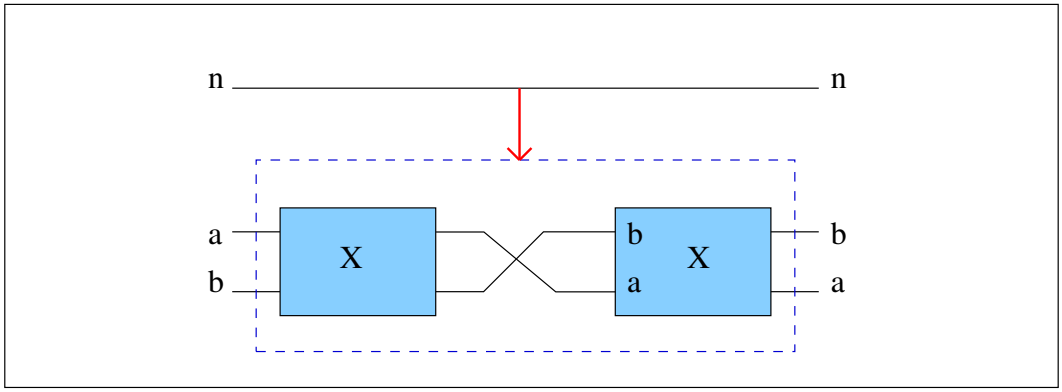


Figure 10: A representation of the SRL program "for n(for b(inc a); for a(inc b))". Although the wires (or program registers) keep their identity when going from left to right (and following the bents), this *is not a block diagram* because the lower "block" is not independent of the upper line n. The rich and powerful interdependence between n and the lower block is represented by the red arrow that means: execute m[n] times the dashed block, where m[n] denotes the contents of n. Each of the blocks denoted by X has of course an inner block (consisting of a single inc instruction).

Recovering the information

Define the SRL program Q, using $n + m + 1$ registers, as described in Figure 12, page 17 (diagram in Figure 13, page 18). The block P^{-1} recovers, at the end of the computation, the inputs of the registers, as well as the input 0's.

Figure 14, page 19, shows that this recovering technique can substantially reduce the memory used to compute a collection of function values (for the same arguments).

4.1 Applications of the recovering method

1. Functions that can be computed in the SRL language with the aid of m 0-initialised registers. These are necessarily PR (primitive recursive) functions.
2. Call them m-zeros functions.
3. Is every PR function an m-zero function?
4. If so, the IDENT decision problem is undecidable for the class of m-zero functions.
5. The values $f_1(\overline{x}),\ldots, f_p(\overline{x})$ can be computed by a $(m+p)$-zero SRL function.

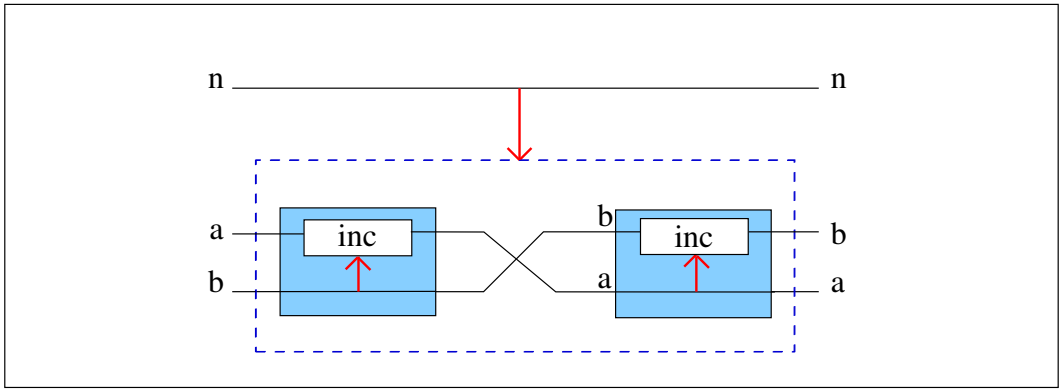


Figure 11: A still more detailed of the SRL "Fibonacci" program, "for n(for b(inc a); for a(inc b))". Neither the blue nor the "inc" rectangles are blocks of the entire program (which can not be divided in blocks).

5 Comparison with the category of classical circuits

As we have seen, classical reversible digital circuits [FT82] and SRL programs [Mat03, Per14] are, in some sense, similar. But there are also big differences.

For digital circuits it is usual to assume that the inputs and outputs are at the extreme left and at the extreme right, respectively. The "computation" proceeds from left to tight.

A *block* of a digital circuit is a part of the circuit with the same wires at left (inputs) and the same wires at the right (outputs). No other wires "communicate" with the exterior.

A *block* of a SRL program is a part of the program consisting in a sequence of SRL instructions. A SRL instruction may, of course, include "inner" blocks, as in "for $a(P)$". The input and the output values of a block are the contents of the block registers before and after an execution of the block.

Similarities

In both digital circuits and SRL programs

- The arity (number of wires or of programming registers) of the input and output are the same.
- For blocks, the input and output arities are also equal.
- The technique described in Section 4 (page 14) for recovering the input values, possibly zero, can be applied in both cases.

Program $Q(\overline{x}, \overline{0}, 0)$
Inputs: $x_1, \ldots, x_n, x_{n+1} = 0, \ldots, x_{n+m} = 0, z = 0$
Output: final contents of register z
1 run $P(x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+m})$;
2 for y(inc z); {z now contains $f(\overline{x})$}
3 run $P^{-1}(x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+m})$;

Figure 12: Notes: y is the register x_{n+m}; z is the last argument in $Q(\overline{x}, \overline{0}, 0)$ (z is initialised with 0).
Line 1: Run $P(\overline{x}, \overline{0})$, computing $f(\overline{x})$ with the help of $m \geq 0$ additional registers initialised with 0.
Line 2: the contents of y, which is $f(\overline{x})$, is copied to z.
Line 3: the current contents of $x_1, \ldots, x_{n+m}$ are used as inputs of P^{-1}, so that, at the end of the computation all the registers $x_1, \ldots, x_{n+m}$ recover their *initial* values. The final contents of z is $f(\overline{z})$.

Differences between reversible digital circuits and SRL programs

Registers of SRL program: more "identity"

A SRL program is executed sequentially. The possible modifications in the contents of the registers is determined by the instruction itself, for instance "inc a". Thus, the semantics of SRL provides a method for obtaining the register contents as a function of time, say $m[x_i, t]$. The registers have an "identity" which is never lost during the execution. That is, anywhere in a SRL program, x_1 (for instance) refers the same memory location.

For digital circuits, the situation is similar if we agree on the correspondence between input and output wires of each basic digital units, that is, for for each indivisible black-box, such as, for instance, a Fredkin gate.

SRL programs: more complex

First notice that a digital circuit with n wires can compute only a finite number of transformations[3].

For SRL programs with n registers the number of possible transformations is unbounded (it depends on the program: consider for instance the programs "ε",

[3]Which is $(2^n)^{2^n}$.

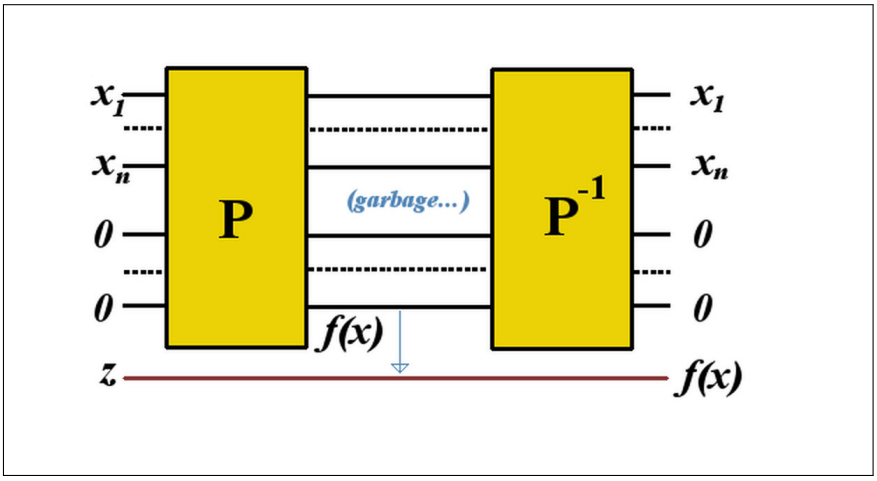


Figure 13: Diagram of the program shown in Figure 12, page 17. Note that, although auxiliary input zeros are used, all the inputs x_i ($1 \leq i \leq n$), and all the 0's except one (register z) are recovered at the end of the computation.

"inc a", "inc a; inc a", "inc a; inc a; inc a"...

Even for a fixed SRL program, the number of possible outputs may be infinite, consider the program "for n(inc a)" with initial value $a = 0$. The arity is 2, the number of possible final contents of a is unbounded (depends on the input value of n). Thus, and relatively to digital circuits, the SRL programs are enriched by

$$[\text{Infinite domain } (\mathbb{Z})] \quad + \quad [\text{loop instructions}]$$

Division in blocks

A digital circuit can often be partitioned in blocks, possibly at several "levels".

On the other hand, each loop instruction of a SRL program, say "for $n(P)$", is indivisible. Of course, the inner program P can often be divided in blocks. See Figures 9 (page 14), 10 (page 15), and 11 (page 16).

The blocks of a SRL program can be defined as we saw in Section 2 (page 5).

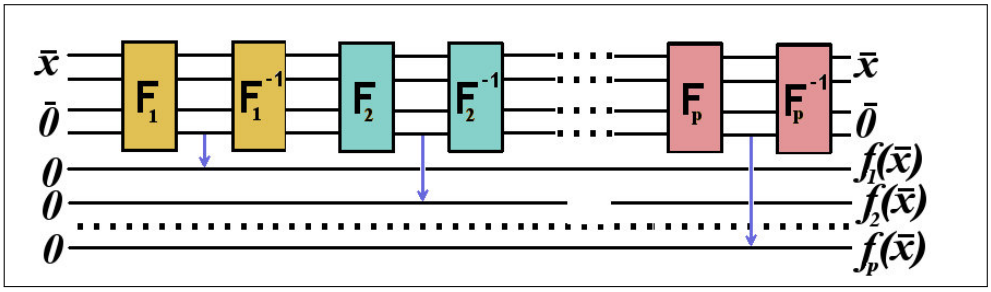


Figure 14: Recall Figure 13, page 18. Assume that the p values $f_1(\overline{x})$, $f_2(\overline{x})$,..., $f_p(\overline{x})$ are computed by the blocks F_1, F_2,..., F_p, respectively. Each of these p computations uses n "argument" registers (n is the arity of the functions) and $m \geq 0$ registers initialised with 0. By the recovering method the p values are computed using only $n+m+p$ programming registers; the straightforward method uses $p \times (n+m)$ registers.

Mainly transcribed – Theoretical foundations

In this section we begin by reviewing the Category Theory formulation of reversible classical digital circuits, see for instance [GA08, Gra06, YY09, AG05]. This part contains transcriptions and it should only be used for my personal study. The reader is invited to consult the original references.

A Reversible classical circuits

We will use the following notation:
$\mathbb{N}_2$: a two-valued set,
$[a] = \{0, 1, \ldots, a-1\}$ (for any positive integer a).

A.1 Definitions and comments

The following is a complete set of reversible digital circuits.

1. $X : \mathbb{N}_2 \to \mathbb{N}_2$, the **not** transformation (which is unary).
2. "wires"[4]: a permutation of $[a]$, or in other words, a bijection **wires** : $[a] \to [a]$. Two equivalent ways of thinking of **wires** are as a permutation of the contents and a permutation of the names.
3. Sequential composition, $\psi \circ \phi$, such that $(\psi \circ \phi)(\overline{x}) = \psi(\phi(\overline{x}))$.

[4]This is the name used in [Gra06]. A more suggestive name would be "permutation".

4. Parallel composition, $\psi \times \phi : [2a] \to [2a]$.

5. Conditional. Let $\psi, \phi : [a] \to [a]$, and consider an extra control wire c. Then, $(\psi \mid \phi) : [a+1] \to [a+1]$ implements "if c then ψ else ϕ".
   The not and the "if/then" are sufficient to implement if/then/else.

The following text is essentially from [Gra06]. For more detail on Category Theory, see for instance [Lan71].

A *category* $\mathcal{C}$ is a collection, or class, of objects $(a, b, c, ...)$ with a collection of unique *morphisms*, also called *arrows*, between them.

For any two morphisms $f \in a \to b$, $g \in b \to c$, there exists a unique *composition* morphism, $g \circ f \in a \to c$, which is associative. There is also the additional constraint that a distinguished identity morphism must exist for every object.

A *monoidal category* is a category $\mathcal{C}$, as above, equipped with a binary functor $\otimes \in \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, called a *tensor*, with a unit object I. A monoidal category must have three natural isomorphisms, which express the fact that the tensor operation must be associative, have a left and right identity. Associativity is given by:

$$\alpha_{a,b,c} \in (a \otimes b) \otimes c \ \to \ a \otimes (b \otimes c),$$

the left-identity by:

$$\lambda_a \in I \otimes a \ \to \ a,$$

and the right identity by:

$$\rho_a \in a \otimes I \to a.$$

Additionally, these three natural transformations are subject to certain *coherence conditions*, which are given by [Lan71].

A *strict monoidal category* is a monoidal category in which the three natural transformations α, λ, and ρ are all the identity transformation.

Finally, a *groupoid* is a category in which every morphism is an isomorphism, i.e. there exists an inverse for every morphism, such that the composition of a morphism and its inverse gives the identity morphism.

[...] Reversible circuits, [...], will be reformulated in the category $\mathbf{FCC}^{\simeq}$, the category of reversible finite classical computations. [...] The purpose of this reformulation is to make precise the informal construction of reversible computations given previously.

Reversible computations are modelled here as a category, where for every morphism $\varphi \in (\mathbf{FCC}^{\simeq}\, a\, b)$ there is an inverse $\varphi^{-1} \in (\mathbf{FCC}^{\simeq}\, b\, a)$, such that φ and φ^{-1} are isomorphisms, and a and b are finite sets.

The morphisms represent computations, and the requirement for the existence of an inverse computation, such that there is an isomorphism, ensures the computation is reversible. As every morphism in $\mathbf{FCC}^{\simeq}$ is an isomorphism, it follows that $\mathbf{FCC}^{\simeq}$ is in fact a groupoid. Any isomorphic objects are assumed to be equal, i.e. $\mathbf{FCC}^{\simeq}$ is strict. It follows from this that $(\mathbf{FCC}^{\simeq}\, a\, b) = \{\,\}$ if $a \neq b$, and consequently homsets of $\mathbf{FCC}^{\simeq}$ are denoted $\mathbf{FCC}^{\simeq}\, a = \mathbf{FCC}^{\simeq}\, a\, a$; the source and target bit-vectors must be of the same size (have the same number of wires) for the computation to be reversible.

$\mathbf{FCC}^{\simeq}$ has a strict monoidal structure $(I, \otimes)$, where $I = 0$ and $a \otimes b = a + b$ [a and b are "thought" as disjoint]. A special object of Booleans is defined as $\mathbb{N}_2$, with $\mathbb{N}_2 = 1$;

the monoid of addition lifts to a strict monoidal structure on $\mathbf{FCC}$.

As computations take place on bit-vectors, which are collections of Booleans, only objects generated from $(I = 0, \mathbb{N}_2 = 1, \otimes = +)$ are interesting; if $\mathbb{N}_2$ represents a wire, then $\mathbb{N}_2 \otimes \mathbb{N}_2$ is two wires, etc. Hence natural numbers $a \in \mathbb{N}$ can be used to denote the object $\mathbb{N}_2^a$. This gives $I = 0$, $\mathbb{N}_2 = 1$, and $a \otimes b = a + b$, as stated previously. The objects of the category $\mathbf{FCC}^{\simeq}$ are therefore the initial segment of a, as defined previously, $[a] = \{i \in N \mid i < a\}$. Note that $\mathbf{FCC}^{\simeq}$ is the free symmetric monoidal category on one object: $\mathbb{N}_2$.

The morphisms of the category $\mathbf{FCC}^{\simeq}a$ are the circuits of arity a, [...] which can be characterised inductively [...]

B Disjoint union and coproduct

Recall a possible definition of disjoint union $A \biguplus B$, also denoted $A + B$:

$$A \biguplus B = \{(0, a) : a \in A\} \cup \{(1, b) : b \in B\}.$$

The following is from the Wikipedia,
`https://en.wikipedia.org/wiki/Coproduct`.

In Category Theory the disjoint union is defined as a coproduct in the category of sets.

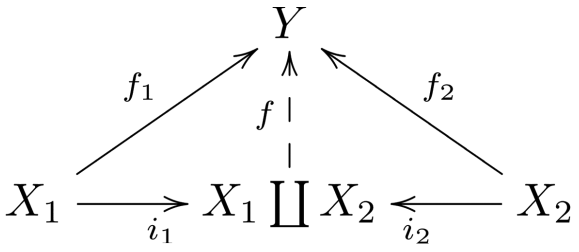
In Category Theory, the coproduct, or categorical sum, is a category-theoretic construction which includes as examples
– the disjoint union of sets and of topological spaces,
– the free product of groups, and
– the direct sum of modules and vector spaces.
The coproduct of a family of objects is essentially the "least specific" object to which each object in the family admits a morphism. It is the category-theoretic dual notion to the categorical product, which means the definition is the same as the product but with all arrows reversed. Despite this seemingly innocuous change in the name and notation, coproducts can be and typically are dramatically different from products.

B.1 Coproduct

Let $\mathcal{C}$ be a category and let X_1 and X_2 be objects in that category. An object is called the *coproduct* of these two objects, written $X1 \coprod X2$ or $X1 \oplus X2$ or sometimes simply $X1 + X2$, if there exist morphisms $i_1 : X_1 \to X_1 \coprod X_2$ and $i_2 : X_2 \to X_1 \coprod X_2$ satisfying a universal property:

> for any object Y and morphisms $f_1 : X_1 \to Y$ and $f_2 : X_2 \to Y$, there exists a unique morphism $f : X_1 \coprod X_2 \to Y$ such that $f_1 = f \cdot i_1$ and $f_2 = f \cdot i_2$.

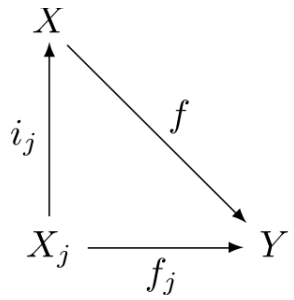
That is, the following diagram commutes:



The unique arrow f making this diagram commute may be denoted

$f_1 \coprod f_2$ or $f_1 \oplus f_2$ or $f_1 + f_2$ or $[f_1, f_2]$. The morphisms i_1 and i_2 are called *canonical injections*, although they need not be injections nor even monic.

The definition of a coproduct can be extended to an arbitrary family of objects indexed by a set J. The coproduct of the family $\{X_j : j \in J\}$ is an object X together with a collection of morphisms $i_j : X_j \to X$ such that, for any object Y and any collection of morphisms $f_j : X_j \to Y$, there exists a unique morphism f from X to Y such that $f_j = f \cdot i_j$. That is, the following diagrams commute (for each $j \in J$):



B.2 Examples

The coproduct in the category of sets is simply the disjoint union with the maps i_j being the inclusion maps. Unlike direct products, coproducts in other categories are not all obviously based on the notion for sets, because unions don't behave well with respect to preserving operations (e.g. the union of two groups need not be a group), and so coproducts in different categories can be dramatically different from each other. For example, the coproduct in the category of groups, called the free product, is quite complicated. On the other hand, in the category of Abelian groups (and equally for vector spaces), the coproduct, called the direct sum, consists of the elements of the direct product which have only finitely many nonzero terms. (It therefore coincides exactly with the direct product in the case of finitely many factors.)

In the case of topological spaces coproducts are disjoint unions with their disjoint union topologies. That is, it is a disjoint union of the underlying sets, and the open sets are sets open in each of the spaces, in a rather evident sense. In the category of pointed spaces, fundamental in homotopy theory, the coproduct is the wedge sum (which amounts to joining a collection of spaces with base points at a common base point).

Despite all this dissimilarity, there is still, at the heart of the whole thing, a disjoint union: the direct sum of Abelian groups is the group generated by the

"almost" disjoint union (disjoint union of all nonzero elements, together with a common zero), similarly for vector spaces: the space spanned by the "almost" disjoint union; the free product for groups is generated by the set of all letters from a similar "almost disjoint" union where no two elements from different sets are allowed to commute.

B.3 A Theorem from "ProofWiki"

Theorem 1 *Let* **Set** *be the category of sets. Let S and T be sets. Then their disjoint union $S \bigcup\!\!\!\!\!+\, T$ is a coproduct in* **Set**.

Note. Here $f \coprod g$ will be denoted by $[f, g]$.

Proof. We have the implicit mappings $i_1 : S \to S \bigcup\!\!\!\!\!+\, T$ and $i_2 : T \to S \bigcup\!\!\!\!\!+\, T$ defined by:

$$\begin{aligned} i_1(s) &= (s, 1) \\ i_2(t) &= (t, 2). \end{aligned}$$

Now given a set V and mappings $f : S \to V$ and $g : T \to V$, there is to be a unique $[f, g] : S \bigcup\!\!\!\!\!+\, T \to V$ such that:

$$\begin{aligned} f &= [f, g] \cdot i_1 \\ g &= [f, g] \cdot i_2. \end{aligned}$$

Define $[f, g]$ by:

$$[f, g](x, \delta) = \begin{cases} f(x) & \text{if } \delta = 1 \\ g(x) & \text{if } \delta = 2. \end{cases}$$

Now it is immediate that $[f, g]$ so defined satisfies the two conditions above. Furthermore, these conditions fix $[f, g]$ uniquely, since every $(x, \delta) \in S \bigcup\!\!\!\!\!+\, T$ has $\delta = 1$ or $\delta = 2$. Hence the result, by definition of coproduct. □

B.4 Discussion

The coproduct construction given above is actually a special case of a colimit in category theory. The coproduct in a category $\mathcal{C}$ can be defined as the colimit of any functor from a discrete category J into $\mathcal{C}$. Not every family $\{X_j\}$ will have a coproduct in general, but if it does, then the coproduct is unique in a strong sense: if $i_j : X_j \to X$ and $k_j : X_j \to Y$ are two coproducts of the family $\{X_j\}$, then (by the definition of coproducts) there exists a unique isomorphism $f : X \to Y$ such that $fi_j = k_j$ for each j in J.

As with any universal property, the coproduct can be understood as a universal morphism. Let $\Delta : \mathcal{C} \to \mathcal{C} \times \mathcal{C}$ be the diagonal functor which assigns to each

object X the ordered pair (X, X) and to each morphism $f : X \to Y$ the pair (f, f). Then the coproduct $X + Y$ in $\mathcal{C}$ is given by a universal morphism to the functor Δ from the object (X, Y) in $\mathcal{C} \times \mathcal{C}$.

The coproduct indexed by the empty set (that is, an empty coproduct) is the same as an initial object in $\mathcal{C}$.

If J is a set such that all coproducts for families indexed with J exist, then it is possible to choose the products in a compatible fashion so that the coproduct turns into a functor $\mathcal{C}J \to \mathcal{C}$. The coproduct of the family $\{X_j\}$ is then often denoted by $\coprod_j X_j$, and the maps i_j are known as the *natural injections*.

Letting $\mathbf{Hom}_C(U, V)$ denote the set of all morphisms from U to V in $\mathcal{C}$ (that is, a hom-set in $\mathcal{C}$), we have a natural isomorphism

$$\mathbf{Hom}_{\mathcal{C}}\left(\coprod_{j\in J} X_j, Y\right) \cong \prod_{j\in J} \mathbf{Hom}_{\mathcal{C}}(X_j, Y),$$

given by the bijection which maps every tuple of morphisms

$$(f_j)_{j\in J} \in \prod_{j\in J} \mathbf{Hom}_{\mathcal{C}}(X_j, Y)$$

(a product in **Set**, the category of sets, which is the Cartesian product, so it is a tuple of morphisms) to the morphism

That this map is a surjection follows from the commutativity of the diagram: any morphism f is the coproduct of the tuple

$$(f \cdot i_j)_{j\in J}.$$

That it is an injection follows from the universal construction which stipulates the uniqueness of such maps. The naturality of the isomorphism is also a consequence of the diagram. Thus the contravariant hom-functor changes coproducts into products. Stated another way, the hom-functor, viewed as a functor from the opposite category C^{op} to **Set** is continuous; it preserves limits (a coproduct in $\mathcal{C}$ is a product in C^{op}).

If J is a finite set, say $J = \{1, \ldots, n\}$, then the coproduct of objects $X_1, \ldots, X_n$ is often denoted by $X_1 \oplus \ldots \oplus X_n$. Suppose all finite coproducts exist in $\mathcal{C}$, coproduct functors have been chosen as above, and 0 denotes the initial object

of $\mathcal{C}$ corresponding to the empty coproduct. We then have natural isomorphisms

$$\begin{aligned} &X \oplus (Y \oplus Z) \;\cong\; (X \oplus Y) \oplus Z \;\cong\; X \oplus Y \oplus Z \\ &X \oplus 0 \;\cong\; 0 \oplus X \;\cong\; X \\ &X \oplus Y \;\cong\; Y \oplus X. \end{aligned}$$

These properties are formally similar to those of a commutative monoid; a category with finite coproducts is an example of a symmetric monoidal category.

If the category has a zero object Z, then we have unique morphism $X \to Z$ (since Z is terminal) and thus a morphism $X \oplus Y \to Z \oplus Y$. Since Z is also initial, we have a canonical isomorphism $Z \oplus Y \cong Y$ as in the preceding paragraph. We thus have morphisms $X \oplus Y \to X$ and $X \oplus Y \to Y$, by which we infer a canonical morphism $X \oplus Y \to X \times Y$. This may be extended by induction to a canonical morphism from any finite coproduct to the corresponding product. This morphism need not in general be an isomorphism; in **Grp** it is a proper epimorphism while in **Set**$\star$ (the category of pointed sets) it is a proper monomorphism. In any pre-additive category, this morphism is an isomorphism and the corresponding object is known as the biproduct. A category with all finite biproducts is known as an semi-additive category.

If all families of objects indexed by J have coproducts in $\mathcal{C}$, then the coproduct comprises a functor $\mathcal{C}^j \to \mathcal{C}$. Note that, like the product, this functor is covariant.

C Monoidal categories

from the Wikipedia, "Cartesian monoidal category"
In mathematics, a monoidal category (or tensor category) is a category $\mathcal{C}$ equipped with a bifunctor

$$\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$$

that is associative up to a natural isomorphism, and an object I that is both a left and right identity for $\otimes$, again up to a natural isomorphism. The associated natural isomorphisms are subject to certain coherence conditions, which ensure that all the relevant diagrams commute.

The ordinary tensor product makes vector spaces, Abelian groups, R-modules, or R-algebras into monoidal categories. Monoidal categories can be seen as a generalisation of these and other examples. Every (small) monoidal category may also be viewed as a "categorification" of an underlying monoid, namely the monoid whose elements are the isomorphism classes of the category's objects and whose binary operation is given by the category's tensor product.

In category theory, monoidal categories can be used to define the concept of a monoid object and an associated action on the objects of the category. They are

also used in the definition of an enriched category.

Monoidal categories have numerous applications outside of category theory proper. They are used to define models for the multiplicative fragment of intuitionistic linear logic. They also form the mathematical foundation for the topological order in condensed matter. Braided monoidal categories have applications in quantum information, quantum field theory, and string theory.

C.1 Formal definition

A monoidal category is a category $\mathcal{C}$ equipped with a monoidal structure. A monoidal structure consists of the following:

1. a bifunctor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ called the tensor product or monoidal product,
2. an object I called the unit object or identity object,
3. three natural isomorphisms subject to certain coherence conditions expressing the fact that the tensor operation
   - is associative: there is a natural (in each of three arguments A, B, C isomorphism α, called associator, with components $\alpha A, B, C : (A \otimes B) \otimes C \cong A \otimes (B \otimes C)$,
   - has I as left and right identity: there are two natural isomorphisms λ and ρ, respectively called left and right unitor, with components $\lambda_A : I \otimes A \cong A$ and $\rho_A : A \otimes I \cong A$.

The coherence conditions for these natural transformations are similar to those of coproducts. For instance,
for all A, B, C, and D in $\mathcal{C}$, the pentagon diagram

$$\begin{array}{ccccc} ((A \otimes B) \otimes C) \otimes D & \xrightarrow{\alpha_{A,B,C} \otimes 1_D} & (A \otimes (B \otimes C)) \otimes D & \xrightarrow{\alpha_{A,B \otimes C,D}} & A \otimes ((B \otimes C) \otimes D) \\ \downarrow{\scriptstyle \alpha_{A \otimes B,C,D}} & & & & \downarrow{\scriptstyle 1_A \otimes \alpha_{B,C,D}} \\ (A \otimes B) \otimes (C \otimes D) & & \xrightarrow[\alpha_{A,B,C \otimes D}]{} & & A \otimes (B \otimes (C \otimes D)) \end{array}$$

commutes.

A strict monoidal category is one for which the natural isomorphisms α, λ and ρ are identities. Every monoidal category is monoidally equivalent to a strict monoidal category.

C.2 Examples

Any category with finite products can be regarded as monoidal with the product as the monoidal product and the terminal object as the unit. Such a category is sometimes called a Cartesian monoidal category. For example:

Set, the category of sets with the Cartesian product, any particular one-element set serving as the unit.

Cat, the category of small categories with the product category, where the category with one object and only its identity map is the unit.

Dually, any category with finite coproducts is monoidal with the coproduct as the monoidal product and the initial object as the unit. Such a monoidal category is *called co-cartesian monoidal.*

R-Modules, the category of modules over a commutative ring R, is a monoidal category with the tensor product of modules $\otimes R$ serving as the monoidal product and the ring R (thought of as a module over itself) serving as the unit.

References

[AG05] Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. In *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, pages 249–258, 2005.

[Bae06] John Baez. Classical versus quantum computation, 2006. See also "n-Category Café", `https://golem.ph.utexas.edu/category/`.

[BS09] John Baez and Mike Stay. Physics, Topology, Logic and Computation: a Rosetta stone, 2009. `arXiv.org > quant-ph > arXiv:0903.0340`.

[FT82] Edward Fredkin and Tommaso Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21:219–253, 1982.

[GA08] Alexander Green and Thorsten Altenkirch. From reversible to irreversible computations. *Electronic Notes in Theoretical Computer Science, URL:* `www.elsevier.nl/locate/entcs`, 210:65–74, 2008. Also in QPL06, Workshop on Quantum Physics and Logic.

[Gra06] Jonathan Grattage. *A functional quantum programming language.* PhD thesis, University of Nottingham, UK, 2006.

[Lan71] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, 1971.

[Mat03] Armando B. Matos. Analysis of a simple reversible language. *Theoretical Computer Science*, 290(3):2063–2074, 2003.

[Mat17a] Armando B. Matos. On the additional memory used to "reversibilize" computations, Preliminary Notes. 2017.

[Mat17b] Armando B. Matos. Register reversible languages II. (work in progress), 2017.

[MRP17] Armando B. Matos, Luca Roversi, and Luca Paolini. For each positive integer k there is a SRL program $p_k(n, a, b)$ such that, in the computation $p_k(n, 0, 0)$, the final contents of a, of b and of n are greater than an exponential tower of k 2's with n at the top. 2017.

[Per14] Kalyan Perumalla. *Introduction to Reversible Computing.* CRC Press, 2014.

[PPR16a] Luca Paolini, Mauro Piccolo, and Luca Roversi. A class of recursive permutations which is primitive recursive complete, 2016. Submitted.

[PPR16b] Luca Paolini, Mauro Piccolo, and Luca Roversi. A class of reversible primitive recursive functions, 2016. Dipartimento di Informatica, Università degli Studi di Torino, Corso Svizzera 185, 10149 Torino.

[Sel11] Peter Selinger. A survey of graphical languages for monoidal categories. In Bob Coecke, editor, *New Structures for Physics*, volume 813 of *Lecture Notes in Physics*, pages 289–355. Springer, 2011.

[Sob17] Pawel Sobocinski. Graphical Linear Algebra, 2017. 22nd Estonian Winter School in Computer Science.

[UPY14] Irek Ulidowski, Iain Phillips, and Shoji Yuen. Concurrency and reversibility. In *Proceedings of Sixth International Conference on Reversible Computation. Lecture Notes in Computer Science, volume 8507*, pages 1–14. Springer-Verlag, 2014.

[YY09] Tomoo Yokoyama and Tetsuo Yokoyama. Functoriality in reversible circuits (work in progress). *ENTCS. Elsevier*, 2009.