A class of Recursive Permutations which is Primitive Recursive complete

Luca Paolini[a], Mauro Piccolo[a], Luca Roversi[a]

[a] *Dipartimento di Informatica – Università di Torino*

Abstract

We focus on total functions in the theory of reversible computational models. We define a class of recursive permutations, dubbed Reversible Primitive Permutations (RPP) which are computable invertible total endo-functions on integers, so a subset of total reversible computations. RPP is generated from five basic functions (identity, negation, successor, predecessor, swap), two notions of composition (sequential and parallel), one functional iteration and one functional selection. Also, RPP is closed by inversion, is expressive enough to encode Cantor pairing and the whole class of Primitive Recursive Functions.

Keywords: Reversible Computation, Unconventional Computing Models, Computable Permutations.
PACS: code, code
PACS:

1. Introduction

Mainstream computer science tends to see models of computations as tools that strongly focus on one of the two possible directions of computation. We typically think how to model the way the computation evolves from inputs to outputs while we (reasonably) overlook the behavior in the opposite direction, from outputs to inputs. Thus, in general, models of computations are not backward deterministic (and not reversible.)

For a very rough intuition about what reversible computation deals with, we start by an example. Let us think about our favorite programming language and think of implementing the addition between two natural numbers m and n. Very likely — of course without absolute certainty — we shall end up with some procedure `sum` which takes two arguments and which yields their sum. For

Email addresses: paolini@di.unito.it, luca.paolini@unito.it (Luca Paolini), piccolo@di.unito.it, mauro.piccolo@unito.it (Mauro Piccolo), roversi@di.unito.it, luca.roversi@unito.it (Luca Roversi)
URL: http://www.di.unito.it/~paolini (Luca Paolini), http://www.di.unito.it/~piccolo (Mauro Piccolo), http://www.di.unito.it/~rover (Luca Roversi)

example, sum(3,2) would yield 5. What if we think of asking for the values m and n such that sum$(m,n) = 5$? If we had implemented sum in a prolog-like language, then we could exploit its non deterministic evaluation mechanism to list all the pairs $(0,5), (1,4), (2,3), (3,2), (4,1)$ and $(5,0)$ all of which would well be a correct instance of (m,n). In a reversible setting we would obtain exactly the pair we started from. I.e., the computation would be backward deterministic. The interest for the backward determinism arose in the sixties of the last century, studying the thermodynamics of computation and the connections between information theory, computability and entropy. After then, the interest for the reversible computing has slowly but incessantly grown.

Here, some general non-exhaustive list of references follows. Reversible Turing machines are in [19, 2] and [1, 10] study some of their computational theoretic properties. Many research efforts have been focused on boolean functions and cellular automata as models of the reversible computation (see [25, 36] for instance.) Moreover, some research focused on reversible programming languages, like [11, 37]. Of course the interest to build a comprehensive knowledge about reversible computation is much wider than the ("mere") will to fully cover its computational theoretic aspects. The book [28] is a comprehensive introduction to the subject which spans from low-power-consumption reversible hardware to emerging alternative computational models like quantum [7, 38] or bio-inspired [6] of which reversibility is one the unavoidable building blocks.

Goal. The focus of this work is on a *functional model* of reversible computation. Specifically, we look for a sufficiently expressive class of recursive permutations able to represent all Dedekind-Rózsa-Robinson's Primitive Recursive Functions (PRF) [30, 34] whose relevance can be traced to the slogan "programs which terminate but do not belong to PRF are rarely of practical interest".

In analogy to PRF, we aim at getting a functional characterization of computability and proof theoretical notions – but in a reversible setting, of course — because functions are handier to compose algorithms than other models like, for example, Turing machines-oriented models which, instead, are much more convincing from the implementation view-point. The main key point of PRF that the functional characterization we are looking for must include is the possibility of representing Cantor pairing [31] which means the ability to express all interesting total properties about the traces of Turing machines and of Reversible Turing machines. Last but not least, such a functional characterization must be closed under inversion, which is a very natural to ask for in a class of permutations and of reversible computing models.

Our quest is challenging because various negative results could have prevented success. First of all, we remark that the class of all (total) recursive permutations cannot be finitely effectively enumerated (see [30, Exercise 4-6, p.55]). Since all recursive permutations can be constructively generated starting from the set of primitive recursive permutations [13], it follows that also the primitive recursive permutations cannot be enumerated. Worst, the primitive recursive permutations are not closed by inversion, as proved in [15] (see [27, 34]). For sake of completeness, we recall that similar negative results holds

also for elementary permutations: they cannot be effectively enumerated and they are not closed by inversion (see [4, 12, 14]). Summing up, no hope exists to find any (effective) description neither of the class of recursive permutations nor of the classes of primitive recursive permutations and of elementary permutations.

Literature Comparison. Our quest on the identification of the reversible analogous of PRF must be throughly related to the following works.

- Armando Matos [21] proposes variants of a programming language inspired to LOOP programs (i.e. programs of a language that Meyer and Ritchie [24] conceived to program PRF functions.) Matos is the first researcher who adopts $\mathbb{Z}$ as ground data for his analysis. We share the choice with him. The work [21] focuses on the algebraic aspects of his languages and his relations with group of matrices. Matos leaves the comparison between "the classes of SRL- and ESRL-definable functions with the class of primitive recursive functions" as future work. Also, unpublished notes by Matos exist [22]. They extend his published works by showing that *-SRL is sufficiently expressive to simulate reversible circuits [36] and that PRF-functions contain *-SRL via a suitable embedding. *-SRL is quite close to the class RPP we propose. Nevertheless, we *conjecture* that no variant of Matos' languages exists able to simulate the functional selection we have in RPP.

- [27] is a precursor of this work which we improve in at least two respects. [27] introduces the class RPRF which is closed by inversion and is PRF-complete. The main difference between RPRF and the class RPP we introduce here is that the definition of the latter is strictly more essential than the definition of the former. Specifically, Section 5 shows how to represent Cantor pairing in RPP which is the key to encode stacks of integers inside RPP and to obtain its PRF-completeness as in Section 6. Instead, the encoding of stacks inside RPRF relies on primitives which pair and unpair integers — in analogy to Cantor pairing —, but which we *explicitly* included in the definition of RPRF. Therefore, we here indirectly prove that RPRF in [27] contains (unnecessary) syntactic sugar.

  Both in [27] and in this work the PRF-completeness of RPRF and RPP relies on ancillary arguments. However, the map from PRF and RPP we propose here fully internalizes what is known as Bennett's trick. This means leaving ancillae without any garbage in them, at the end of the simulation of any $f \in \mathsf{PRF}$ as a function of RPP. This is not true for the simulation of $f \in \mathsf{PRF}$ as an element of RPRF. Technically, the difference is evident by comparing Definition 3 of Section 6 which fixes how to represent any $f \in \mathsf{PRF}$ into RPP and the corresponding definition in [27].

- Finally we focus on [9]. It introduces the class of reversible functions $\mathcal{RI}$ which is as expressive as Kleene partial recursive functions [5, 26]. Therefore, the focus of [9] is on partial reversible functions while ours is

on total ones. The expressiveness of $\mathcal{RI}$ is clearly stronger than the one of RPP. However, we see $\mathcal{RI}$ as less abstract than RPP for two reasons. On one side, the primitive functions of $\mathcal{RI}$ relies on the given specific binary representation of natural numbers. On the other, it is not evident that $\mathcal{RI}$ is the extension of a total sub-class analogous to RPP which should ideally correspond to PRF, but in a reversible setting.

Contributions. We propose a formalism that identifies a class of functions which we call Reversible Primitive Permutations (RPP) and which is strictly included in the set of total invertible functions.

Section 2 defines RPP in analogy with the definition of PRF, i.e. RPP is the closure of composition schemes on basic functions.

The functions of RPP have identical arity and co-arity and take $\mathbb{Z}$ — and not only $\mathbb{N}$ — as their domain because $\mathbb{N}$ is not a group. So, RPP is sufficiently abstract to avoid any reference to any specific encoding of numbers and strongly connects our work to Matos' one [21].

For example, in RPP we can define a `sum` that given the two integers m and n yields $m+n$ and n. Let us represent `sum` as:

$$\begin{matrix} m \\ n \end{matrix}\Big[\ \texttt{sum}\ \Big]\begin{matrix} m+n \\ n \end{matrix} \quad . \tag{1}$$

The implementation of `sum` inside RPP exploits an iteration scheme that iterates n times the successor, starting on the initial value m of the first input. RPP implies the existence of a (meta and) *effective* procedure which produces the inverse $f^{-1} \in \mathsf{RPP}$ of every given $f \in \mathsf{RPP}$. I.e., :

$$\begin{matrix} p \\ n \end{matrix}\Big[\ \texttt{sum}^{-1}\ \Big]\begin{matrix} p-n \\ n \end{matrix} \tag{2}$$

belongs to RPP and it "undoes" `sum` by iterating n times the predecessor on p. So if $p = m+n$, then $p-n = m+n-n = m$. We remark we could have internalized the operation that yields the inverse of a function inside RPP like in [21]. We choose not for sake of simplicity, so avoiding mutually recursive definitions inside RPP.

Concerning the *expressiveness*, RPP is both PRF-complete and PRF-sound. Completeness is the really relevant property between the two because this means that RPP subsumes the class PRF. This result is in Section 6. It requires quite a lot of preliminary work that one can find in Sections 3, 4 and 5 and whose goal is to encode a bounded minimalization, Cantor pairing and the stack as data-type. So, the embedding makes evident various key aspects of reversible computing. The principal ones we like to underline are how to encode Cantor pairing, the unavoidable use of ancillary variables to clone information and the compositional programming under the pattern that Bennett's trick dictates (cf. Section 7.)

Concerning RPP-soundness, which means that PRF subsumes RPP, we address the reader to [27] as a possible reference to the obvious idea about how formally defining an embedding from RPP into PRF.

So, exactly because RPP is a total class of reversible functions which is both PRF-complete and sound we think it is a reasonable candidate to become the analogous of PRF in the setting of reversible computations.

As a last remark, we underline that the speculations leading us to the identification of RPP could slightly widen our foundational perspective on the "realm" of recursive computations and on their mathematical formalizations. We present our reflections in Section 7 together with (some) possible future work.

2. The class RPP of Reversible Primitive Permutations

The identification of RPP merges ideas and observations on Primitive Recursive Functions (PRF) [32, 20, 21, 26], Toffoli's class of boolean circuits [36] and Lafont's algebraic theory on circuits [16]. We quickly recall the crucial ideas we borrowed from the above papers to formalize the Definition 1.

Toffoli's boolean circuits are invertible because conceived to avoid the erasure of information. This is why the boolean circuits in [36] have identical arity and co-arity. RPP adopts this policy for the same reasons.

In analogy with PRF, the class RPP is defined by composing numerical basic functions by means of suitable composition schemes. The will to manipulate numbers suggests that the *successor* S must be in RPP. So S^{-1} — i.e. the *predecessor* P — must be in RPP as well because we want f^{-1} to be effective. This requires that the application of P to 0 remains meaningful. Satisfying this requirement and keeping the definition of RPP as much natural as possible suggested to extend both the domain and co-domain of every function in RPP to $\mathbb{Z}$ so that P applied to 0 can yield -1. In fact, working with $\mathbb{Z}$ as atomic data it is like working with $\mathbb{N}$ up to some of the natural isomorphism we can use to let $\mathbb{Z}$ and $\mathbb{N}$ to correspond.

The core of the composition schemes of RPP comes from [16]. The series composition of functions is ubiquitous in functional computational model so RPP uses one. The parallel composition of functions is natural in presence of co-arity greater than 1 so RPP relies on such a scheme. The class RPP allows to iterate functions and to select one function among three available. As one can expect, this extends its expressive power.

Preliminaries.. Let $\mathbb{Z}$ be the set of integers and let $\mathbb{Z}^n$ be its Cartesian product whose elements are n-tuples for any $n \in \mathbb{N}$. The 0-tuple is $\langle\,\rangle$. The 1-tuple is $\langle x\rangle$ or simply x. In general, we name tuples with n elements as $\underline{a}^n, \underline{b}^n, \ldots$ or simply as $\underline{a}, \underline{b}, \ldots$ if knowing the number of their components is not crucial. By definition, the concatenation $\cdot : \mathbb{Z}^i \times \mathbb{Z}^j \longrightarrow \mathbb{Z}^{i+j}$ is such that $\langle x_1, \ldots, x_j\rangle \cdot \langle y_1, \ldots, y_k\rangle = \langle x_1, \ldots, x_j, y_1, \ldots, y_k\rangle$. Whenever $j = 1$ we shall tend to write $x_1 \cdot \langle y_1, \ldots, y_k\rangle$ in place of $\langle x_1\rangle \cdot \langle y_1, \ldots, y_k\rangle$. Analogously, $\langle x_1, \ldots, x_j\rangle \cdot y_1$ will generally stand for $\langle x_1, \ldots, x_j\rangle \cdot \langle y_1\rangle$. The empty tuple is the neuter element so $\underline{x} \cdot \langle\rangle = \langle\rangle \cdot \underline{x} = \underline{x}$. In fact, we shall generally drop the explicit use of the concatenation operator '$\cdot$'. For example, this means that we tend to replace $\underline{a} \cdot x \cdot \underline{b} \cdot \underline{c}^n \cdot \underline{d}$ by $\underline{a}\, x\, \underline{b}\, \underline{c}^n\, \underline{d}$.

Definition 1 (Reversible Primitive Permutations). Reversible Primitive Permutations (abbreviates as RPP) is a sub-class of Reversible Permutations[1]. By definition, $\mathsf{RPP} = \bigcup_{k\in\mathbb{N}} \mathsf{RPP}^k$ where, for every $k \in \mathbb{N}$, the set RPP^k contains functions with identical *arity* and *co-arity* k. The classes $\mathsf{RPP}^0, \mathsf{RPP}^1, \ldots$ are defined by mutual induction.

- The *identity* I, the *negation* N, the *successor* S and the *predecessor* P belong to RPP^1. We formalize their semantics, which is the one we may expect, by means of two equivalent notations. The first is a standard functional notation that applies the function to the input and produces an output:

$$\mathsf{I}\langle x\rangle := \langle x\rangle\ , \qquad \mathsf{N}\langle x\rangle := \langle -x\rangle\ , \qquad \mathsf{S}\langle x\rangle := \langle x+1\rangle\ , \quad \mathsf{P}\langle x\rangle := \langle x-1\rangle\ .$$

  The second notation is relational. We write the function name between square brackets; the input is on the left hand side and the output in on the right hand side:

$$x\,[\,\mathsf{I}\,]\,x\ , \qquad x\,[\,\mathsf{N}\,]\,-x\ , \qquad x\,[\,\mathsf{S}\,]\,x+1\ , \qquad x\,[\,\mathsf{P}\,]\,x-1\ .$$

- The *swap* χ belongs to RPP^2. The two notations are:

$$\chi\langle x,y\rangle := \langle y,x\rangle \text{ and } \begin{matrix}x\\y\end{matrix}\left[\ \right\rangle^2_{2,1}\ \left]\begin{matrix}y\\x\end{matrix}\right. .$$

- Let $f, g \in \mathsf{RPP}^j$, for some j. The *series composition* of f and g is $(f \fatsemi g)$, belongs to RPP^j and is such that:

$$(f \fatsemi g)\,\langle x_1,\ldots,x_j\rangle = (g\circ f)\langle x_1,\ldots,x_j\rangle \text{ or}$$

$$\begin{matrix}x_1\\\cdots\\x_j\end{matrix}\left[\ f \fatsemi g\ \right]\begin{matrix}y_1\\\cdots\\y_j\end{matrix} = \begin{matrix}x_1\\\cdots\\x_j\end{matrix}\left[\ f\ \right]\left[\ g\ \right]\begin{matrix}y_1\\\cdots\\y_j\end{matrix} .$$

  We remark the use of the programming composition that applies functions rightward, in opposition to the standard functional composition (usually denoted $\circ$).

- Let $f \in \mathsf{RPP}^j$ and $g \in \mathsf{RPP}^k$, for some j and k. The *parallel composition* of f and g is $(f\|g)$, belongs to RPP^{j+k} and is such that:

$$(f\|g)\,\langle x_1,\ldots,x_j\rangle\langle y_1,\ldots,y_k\rangle = (f\langle x_1,\ldots,x_j\rangle)\cdot(g\,\langle y_1,\ldots,y_k\rangle) \text{ or}$$

$$\begin{matrix}x_1\\\cdots\\x_j\\y_1\\\cdots\\y_k\end{matrix}\left[\ f\|g\ \right]\begin{matrix}w_1\\\cdots\\w_j\\z_1\\\cdots\\z_k\end{matrix} = \begin{matrix}x_1\\\cdots\\x_j\end{matrix}\left[\ f\ \right]\begin{matrix}w_1\\\cdots\\w_j\end{matrix} \begin{matrix}y_1\\\cdots\\y_k\end{matrix}\left[\ g\ \right]\begin{matrix}z_1\\\cdots\\z_k\end{matrix} .$$

[1] For sake of precision, we remark that we use Reversible Permutations meaning Total Reversible Recursive Endo-Functions on $\mathbb{Z}^n$ for some $n \in \mathbb{N}$.

- Let $f \in \mathsf{RPP}^k$. The *finite iteration* of f is $\mathsf{It}\,[f]$, belongs to RPP^{k+1} and is such that:

$$\mathsf{It}\,[f]\,(\langle x_1,\ldots,x_k\rangle\cdot x) = (\overbrace{(f \fatsemi \ldots \fatsemi f}^{|x|})\|\mathsf{I})\,(\langle x_1,\ldots,x_k\rangle\cdot x) \text{ or}$$

$$\begin{matrix} x_1 \\ \cdots \\ x_k \\ x \end{matrix}\left[\begin{matrix}\\ \mathsf{It}\,[f] \\ \\ \end{matrix}\right]\begin{matrix} y_1 \\ \cdots \\ y_k \\ x \end{matrix}\Bigg\} = (\underbrace{f \fatsemi \ldots \fatsemi f}_{|x|})\,\langle x_1,\ldots,x_k\rangle \quad .$$

  We remark the *linearity constraint* on the *finite iteration*. The argument x which drives the iteration unfolding cannot be among the arguments of the iterated function f. Moreover, we choose to use x as the last argument of $\mathsf{It}\,[f]$ to avoid any re-indexing of the arguments of f when iterating it.

- Let $f,g,h \in \mathsf{RPP}^k$. The *selection* of one among f,g and h is $\mathsf{If}\,[f,g,h]$, it belongs to RPP^{k+1} and is such that:

$$\mathsf{If}\,[f,g,h]\;(\langle x_1,\ldots,x_k\rangle\cdot x) = \begin{cases} (f\|\mathsf{I})\,(\langle x_1,\ldots,x_k\rangle\cdot x) & \text{if } x>0 \\ (g\|\mathsf{I})\,(\langle x_1,\ldots,x_k\rangle\cdot x) & \text{if } x=0 \\ (h\|\mathsf{I})\,(\langle x_1,\ldots,x_k\rangle\cdot x) & \text{if } x<0 \end{cases} \quad \text{or}$$

$$\begin{matrix} x_1 \\ \cdots \\ x_k \\ x \end{matrix}\left[\begin{matrix}\\ \mathsf{If}\,[f,g,h] \\ \\ \end{matrix}\right]\begin{matrix} y_1 \\ \cdots \\ y_k \\ x \end{matrix}\Bigg\} = \begin{cases} f\,\langle x_1,\ldots,x_k\rangle & \text{if } x>0 \\ g\,\langle x_1,\ldots,x_k\rangle & \text{if } x=0 \\ h\,\langle x_1,\ldots,x_k\rangle & \text{if } x<0 \end{cases} \quad .$$

  We remark the *linearity constraint* imposed on the definition of *selection*. The argument x which determines which among f,g and h must be used cannot be among the arguments of f,g and h. Moreover, we choose to use x as the last argument of $\mathsf{If}\,[f,g,h]$ to avoid any re-indexing of the arguments of f,g and h, once we choose one of them. □

Proposition 1 (Finite Permutations in RPP.)**.** *Each finite permutation with arity and co-arity equal to k belongs to* RPP^k.

Proof. A transposition is a permutation which exchanges two elements and keeps all others fixed. Each finite permutation is generated by compositions of transpositions. □

Notation 1. If the same value occurs in consecutive inputs or outputs, like, for example, in:

$$f\langle x_1,\ldots,x_m,\underbrace{0,\ldots,0}_{r\in\mathbb{N}},y_1,\ldots,y_n\rangle = \langle w_1,\ldots,w_p,\underbrace{0,\ldots,0}_{s\in\mathbb{N}},z_1,\ldots,z_q\rangle \ ,$$

with $m + r + n = p + s + q$, we tend to abbreviate it as:

$$f\langle x_1, \ldots, x_m, 0^r, y_1, \ldots, y_n\rangle = \langle w_1, \ldots, w_p, 0^s, z_1, \ldots, z_q\rangle \text{ or } \begin{matrix} x_1 \\ \cdots \\ x_m \\ 0^r \\ y_1 \\ \cdots \\ y_n \end{matrix} \left[\begin{matrix} \\ \\ \\ \\ \\ \\ \\ \end{matrix}\ f\ \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \end{matrix}\right] \begin{matrix} w_1 \\ \cdots \\ w_p \\ 0^s \\ z_1 \\ \cdots \\ z_q \end{matrix} \ .$$

In particular, 0^0 means that no occurrences of 0 exist. □

To come up with the identification of RPP we have been looking for a reasonably expressive class of total and *numerical* functions in which it is possible to give a gentle (as opposed to "brute force"[2]) status to the operation of inversing a given function f; we mean that the operation f^{-1}, which we standardly define as the one such that $(y, x) \in f^{-1}$ if and only if $(x, y) \in f$ is effective inside RPP.

Proposition 2 (The (syntactical) inverse f^{-1} of any f)**.** *Let $f \in \mathsf{RPP}^j$, for any $j \in \mathbb{N}$. The inverse of f is f^{-1}, belongs to RPP^j and, by definition, is:*

$$\begin{array}{l} \mathsf{I}^{-1} := \mathsf{I}\ ,\ \mathsf{N}^{-1} := \mathsf{N}\ ,\ \mathsf{S}^{-1} := \mathsf{P}\ ,\ \mathsf{P}^{-1} := \mathsf{S}\ ,\ \chi^{-1} := \chi\ , \\ (g \fatsemi f)^{-1} := f^{-1} \fatsemi g^{-1}\ ,\ (f \| g)^{-1} := f^{-1} \| g^{-1}\ , \\ (\mathsf{It}\,[f])^{-1} := \mathsf{It}\left[f^{-1}\right]\ ,\ (\mathsf{If}\,[f, g, h])^{-1} := \mathsf{If}\left[f^{-1}, g^{-1}, h^{-1}\right]\ . \end{array}$$

Then $f \fatsemi f^{-1} = \mathsf{I}$ and $f^{-1} \fatsemi f = \mathsf{I}$.

Proof. By induction on the definition of f. □

Proposition 3 (Relating *series composition* and *parallel composition*)**.** *Let $f, g \in \mathsf{RPP}^j$ and $f', g' \in \mathsf{RPP}^k$ with $j, k \in \mathbb{N}$. Then:*

$$(f \| f') \fatsemi (g \| g') = (f \fatsemi g) \| (f' \fatsemi g')\ .$$

Proof. By definition:

$$\begin{aligned} (f \| f') \fatsemi (g \| g')\, \underline{a}\, \underline{b} &= (g \| g')(f \| f')\, \underline{a}\, \underline{b} \\ &= (g \| g')\, (f \underline{a})\, (f' \underline{b}) \\ &= (g f \underline{a})(g' f' \underline{b}) \\ &= ((f \fatsemi g)\, \underline{a})((f' \fatsemi g')\, \underline{b}) \\ &= ((f \fatsemi g) \| (f' \fatsemi g'))\, \underline{a}\, \underline{b}\ . \end{aligned}$$ □

As a concluding remark, we underline that, in the course of the design of RPP, we aimed at sticking as much as we could to the traditional formalism that PRF represents. In the mean time we tried to keep a reasonable balance between conciseness and easiness of usage which can become quite cumbersome because of the "rewiring" that finite permutations imply.

[2] We remark that the inversion of total recursive functions is possible by using the brute force, see [29, 23].

3. Generalizations inside RPP

We introduce formal generalizations of elements in RPP. This helps simplifying the use of RPP as a programming notation.

Weakening inside RPP. For any given $f \in \mathsf{RPP}^m$ an infinite set $\{\mathsf{w}^n(f) \mid n \geq 1 \text{ and } \mathsf{w}^n(f) \in \mathsf{RPP}^{m+n}\}$ exists such that:

$$\begin{matrix} x_1 \\ \cdots \\ x_m \\ x_{m+1} \\ \cdots \\ x_{m+n} \end{matrix} \left[\begin{matrix} \\ \\ \mathsf{w}^n(f) \\ \\ \\ \end{matrix} \right] \begin{matrix} y_1 \\ \cdots \\ y_n \\ x_{m+1} \\ \cdots \\ x_{m+n} \end{matrix} \Bigg\} = f\,\langle x_1, \ldots, x_m \rangle \quad ,$$

for every $\mathsf{w}^n(f)$. We call $\mathsf{w}^n(f)$ *weakening* of f which we can obviously obtain by *parallel composition* of f with n occurrences of I. In general, if we need some *weakening* of a given f, then we shall not write $\mathsf{w}^n(f)$ explicitly. We shall instead use f and say that it is the *weakening* of f itself.

Generalized identity, negation, successor and predecessor. For every $i \leq n$, the following functions in RPP^n exist:

$$\begin{aligned}
\mathsf{I}^n\,\langle x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n \rangle &= \langle x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n \rangle \\
\mathsf{N}_i^n\,\langle x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n \rangle &= \langle x_1, \ldots, x_{i-1}, -x_i, x_{i+1}, \ldots, x_n \rangle \\
\mathsf{S}_i^n\,\langle x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n \rangle &= \langle x_1, \ldots, x_{i-1}, x_i + 1, x_{i+1}, \ldots, x_n \rangle \\
\mathsf{P}_i^n\,\langle x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n \rangle &= \langle x_1, \ldots, x_{i-1}, x_i - 1, x_{i+1}, \ldots, x_n \rangle \quad .
\end{aligned}$$

They are the weakening of $\mathsf{I}, \mathsf{N}, \mathsf{S}, \mathsf{P} \in \mathsf{RPP}^1$. When the arity n is clear from the context, sometimes we shall use $\mathsf{I}, \mathsf{N}_i, \mathsf{S}_i$ and P_i in place of $\mathsf{I}^n, \mathsf{N}_i^n, \mathsf{S}_i^n$ and P_i^n.

Generalized syntactic permutations. Let $\{i_1, \ldots, i_n\} = \{1, \ldots, n\}$ and $\rho : \mathbb{N}^n \longrightarrow \mathbb{N}^n$ be any permutation. There is at least one definition of the following *generalized syntactic permutation*:

$$\left(\begin{smallmatrix} i_1 & , \ldots , & i_m \\ \rho(i_1), \ldots, \rho(i_m) \end{smallmatrix} \right)^n \quad , \tag{3}$$

which belongs to RPP^n, and which sends the i_1-th input to the $\rho(i_1)$-th output, the i_2-th input to the $\rho(i_2)$-th output etc.. No restrictions exist on the order of $i_1, \ldots, i_m$. From Proposition 1, every generalized syntactic permutation is obviously a suitable composition of *identity*, *binary syntactic permutation*, *series composition*, *parallel composition* and *weakening*.

Generalized finite iteration. Let $f \in \mathsf{RPP}^n$ and $m \geq n + 1$. Let $\langle x_i \rangle$, $L = \langle x_{i_1}, \ldots, x_{i_n} \rangle$ and $\langle x_{j_1}, \ldots, x_{j_{m-n-1}} \rangle$ be three lists which partition $\langle x_1, \ldots, x_m \rangle$.

We are going to define $\mathsf{It}^m_{i,L}[f]$ such that its argument of position i drives an iteration of f from L. I.e., $\mathsf{It}^m_{i,L}[f]\langle x_1,\ldots,x_m\rangle = \langle y_1,\ldots,y_m\rangle$, where:

$$\begin{aligned} y_i &= x_i \\ \langle y_{j_1},\ldots,y_{j_{m-n-1}}\rangle &= \langle x_{j_1},\ldots,x_{j_{m-n-1}}\rangle \\ \langle y_{i_1},\ldots,y_{i_n}\rangle &= (\underbrace{f \fatsemi \ldots \fatsemi f}_{|x_i|})\, L \ . \end{aligned}$$

We observe that $\mathsf{It}^m_{i,L}[f]$ is the identity on $\langle x_{j_1},\ldots,x_{j_{m-n-1}}\rangle$. By definition:

$$\begin{aligned} \mathsf{It}^m_{i,L}[f] := & \begin{pmatrix} i_1,\ldots,i_n, & i & , & j_1 & ,\ldots,j_{m-n-1} \\ 1,\ldots,n & , n+1,n+2,\ldots, & & m & & \end{pmatrix}^m \\ & \fatsemi (\mathsf{It}[f] \| \mathsf{I}^{m-n-1}) \\ & \fatsemi \left(\begin{pmatrix} i_1,\ldots i_n, & i & , & j_1 & ,\ldots,j_{m-n-1} \\ 1,\ldots n & , n+1,n+2,\ldots, & & m & & \end{pmatrix}^m \right)^{-1} \ . \end{aligned}$$

The condition on i preserves the linearity constraint of $\mathsf{It}[f]$.

Generalizing the selection. Let $f,g,h \in \mathsf{RPP}^n$ and $m \geq n+1$. Let $\langle x_i\rangle$, $L = \langle x_{i_1},\ldots,x_{i_n}\rangle$ and $\langle x_{j_1},\ldots,x_{j_{m-n-1}}\rangle$ be a partition of $\langle x_1,\ldots,x_m\rangle$. We are going to define $\mathsf{If}^m_{i,L}[f,g,h]$ such that its argument of position i determines which among f,g and h must be applied to L. I.e., $\mathsf{If}^m_{i,L}[f,g,h]\ \langle x_1,\ldots,x_m\rangle = \langle y_1,\ldots,y_m\rangle$ where:

$$\begin{aligned} y_i &= x_i \\ \langle y_{j_1},\ldots,y_{j_{m-n-1}}\rangle &= \langle x_{j_1},\ldots,x_{j_{m-n-1}}\rangle \\ \langle y_{i_1},\ldots,y_{i_n}\rangle &= \begin{cases} f\langle x_{i_1},\ldots,x_{i_n}\rangle & \text{if } x_i > 0 \\ g\langle x_{i_1},\ldots,x_{i_n}\rangle & \text{if } x_i = 0 \\ h\langle x_{i_1},\ldots,x_{i_n}\rangle & \text{if } x_i < 0 \ . \end{cases} \end{aligned}$$

We observe that $\mathsf{If}^m_{i,L}[f,g,h]$ is the identity on $\langle x_{j_1},\ldots,x_{j_{m-n-1}}\rangle$. By definition:

$$\begin{aligned} \mathsf{If}^m_{i,L}[f,g,h] := & \begin{pmatrix} i_1\ldots i_n, & i & , & j_1 & ,\ldots,j_{n-i-1} \\ 1\ldots n & , n+1,n+2,\ldots, & & m & & \end{pmatrix}^m \\ & \fatsemi (\mathsf{If}[f,g,h] \| \mathsf{I}^{m-n-1}) \\ & \fatsemi \left(\begin{pmatrix} i_1,\ldots,i_n, & i & , & j_1 & ,\ldots,j_{n-i-1} \\ 1,\ldots,n & , n+1,n+2,\ldots, & & m & & \end{pmatrix}^m \right)^{-1} \ . \end{aligned}$$

The condition on i preserves the linearity constraint of $\mathsf{If}^m_i[f,g,h]$.

4. A library of functions in RPP

We introduce functions to simplify further the constructions inside RPP by introducing the use of additional arguments to store auxiliary information. Similar notions can be found everywhere in reversible studies under many names and with little different meanings, e.g. "temporary" lines, storages, channels,

bits and variables (e.g. see [36]). Moreover, following the quantum literature sometimes the "ancillary" replaces the adjective "temporary" (e.g. see [35]).

Concretely, the functions we are going to introduce rely on additional arguments we dub as *ancillary arguments* or *ancillae*. Our ancillae have three purposes: (i) holding the result of the computation; (ii) temporarily recording copies of values for later use; (iii) temporarily storing intermediate results. Without loss of generality, we use ancillae in a very disciplined way by respecting a protocol in order to make easier the understanding. First, all ancillae are supposed to be initialized to zero (or in other words, we neglect the behaviour of functions for values of ancillae other than zero.) Last, the ancillae of kind (ii) and (iii) are set back to zero before the computation ends. Since their specific temporary use, we dub them as *temporary arguments*.

General increment and decrement. Let i, j and $m \in \mathbb{N}$ be distinct. Two functions $\mathsf{inc}_{j;i}, \mathsf{dec}_{j;i} \in \mathsf{RPP}^m$ exist such that:

$$\begin{matrix} x_1 \\ \cdots \\ x_j \\ \cdots \\ x_i \\ \cdots \\ x_m \end{matrix}\left[\begin{matrix} \\ \\ \\ \mathsf{inc}_{j;i} \\ \\ \\ \\ \end{matrix}\right]\begin{matrix} x_1 \\ \cdots \\ x_j \\ \cdots \\ x_i + |x_j| \\ \cdots \\ x_m \end{matrix} \qquad \text{and} \qquad \begin{matrix} x_1 \\ \cdots \\ x_j \\ \cdots \\ x_i \\ \cdots \\ x_m \end{matrix}\left[\begin{matrix} \\ \\ \\ \mathsf{dec}_{j;i} \\ \\ \\ \\ \end{matrix}\right]\begin{matrix} x_1 \\ \cdots \\ x_j \\ \cdots \\ x_i - |x_j| \\ \cdots \\ x_m \end{matrix} \quad .$$

We define them as:

$$\mathsf{inc}_{j;i} := \mathsf{It}^m_{j,\langle i \rangle}\,[\mathsf{S}] \qquad\qquad \mathsf{dec}_{j;i} := \mathsf{It}^m_{j,\langle i \rangle}\,[\mathsf{P}] \quad .$$

A function that compares two integers. Let k, j, i, p, q be pairwise distinct such that $k, j, i, p, q \leq n$, for a given arity $n \in \mathbb{N}$. A function $\mathsf{less}_{i,j,p,q;k} \in \mathsf{RPP}^n$ exists that implements the following relation:

$$\begin{matrix} x_1 \\ \cdots \\ x_k \\ \cdots \\ x_n \end{matrix}\left[\begin{matrix} \\ \\ \mathsf{less}_{i,j,p,q;k} \\ \\ \\ \end{matrix}\right]\begin{matrix} x_1 \\ \cdots \\ x_k + \begin{cases} 1 & \text{if } x_i < x_j \\ 0 & \text{if } x_i \geq x_j \end{cases} \\ \cdots \\ x_n \end{matrix} \quad .$$

The function $\mathsf{less}_{i,j,p,q;k}$ behaves in accordance with the intended meaning whenever the ancillae x_p, x_q are initially set to 0. Tipically, x_k will be used as ancilla initialized to zero. (x_p, x_q are temporary arguments, while x_k is not). In that case, $\mathsf{less}_{i,j,p,q;k}$ returns 0 or 1 in x_k depending on the result of comparing the values x_i and x_j. Both x_p and x_q serve to duplicate the values of x_i and x_j, respectively. The copies allow to circumvent the linearity constraints in presence of nested selections.

For sake of clarity, we label the primitive operators If and It by an exponent to emphasize the used arity.

Let $\{j_1, \ldots, j_{n-5}\} = \{1, \ldots, n\} \setminus \{k, j, i, p, q\}$. By definition:

$$\mathsf{less}_{i,j,p,q;k} := \left(\begin{smallmatrix} k,p,q,i,j,j_1,\ldots,j_{n-5} \\ 1,2,3,4,5,6,\ldots,n \end{smallmatrix}\right)^n \fatsemi \mathsf{inc}_{5;3} \fatsemi \mathsf{inc}_{4;2} \fatsemi \tag{4a}$$

$$(\mathsf{F} \parallel \mathsf{I}^{n-5}) \fatsemi \tag{4b}$$

$$\left(\mathsf{inc}_{5;3} \fatsemi \mathsf{inc}_{4;2}\right)^{-1} \fatsemi \left(\left(\begin{smallmatrix} k,p,q,i,j,j_1,\ldots,j_{n-5} \\ 1,2,3,4,5,6,\ldots,n \end{smallmatrix}\right)^n\right)^{-1} \tag{4c}$$

where

$$\begin{aligned} \mathsf{F} &:= \mathsf{If}^5[\mathsf{If}^4[\mathsf{BothPos}, \mathsf{S}_1^3, \mathsf{S}_1^3], \mathsf{If}^4[\mathsf{I}^3, \mathsf{I}^3, \mathsf{S}_1^3], \mathsf{If}^4[\mathsf{I}^3, \mathsf{I}^3, \mathsf{BothNeg}]] \\ \mathsf{BothPos} &:= \mathsf{dec}_{2;3} \fatsemi \mathsf{If}^3[\mathsf{S}_1^2, \mathsf{I}^2, \mathsf{I}^2] \fatsemi (\mathsf{dec}_{2;3})^{-1} \\ \mathsf{BothNeg} &:= \mathsf{inc}_{2;3} \fatsemi \mathsf{If}^3[\mathsf{I}^2, \mathsf{I}^2, \mathsf{S}_5^2] \fatsemi (\mathsf{inc}_{2;3})^{-1} \quad . \end{aligned}$$

The *series composition* at the lines (4a) and (4c) are one the inverse of the other. The leftmost function at line (4a) moves the five relevant arguments in the first five positions[3]. The rightmost function at line (4a) duplicates x_i and x_j into the ancillae x_p and x_q, respectively. The functions at line (4c) undo what those ones at line (4a) have done. In between them, at line (4b) a part from I^{n-5} which is straightforward, we explain $\mathsf{F} \in \mathsf{RPP}^5$ by the following case analysis scheme:

	$x_i > 0$	$x_i = 0$	$x_i < 0$		
If^5 [If^4[	BothPos,	S_1^3,	S_1^3	]	$x_j > 0$
, If^4[	I^3,	I^3,	S_1^3	]	$x_j = 0$
, If^4[	I^3,	I^3,	BothNeg	]]	$x_j < 0$

The function F receives a given $\langle x_k, x_p + x_i, x_q + x_j, x_i, x_j, x_{j_1}, \ldots, x_{j_{n-5}} \rangle$ as argument and operates on its first five elements. For example, if $x_j < 0$ and $x_j < 0$ then the nested selections of F eventually select $\mathsf{BothNeg}$ that updates the x_j to $x_j + |x_i|$. Then, $\mathsf{BothNeg}$ increments its first argument by 1 if only if $x_j + |x_i| < 0$, i.e. when $x_i < x_j$. Finally, $\mathsf{BothNeg}$ unfolds its local sum.

A function that multiplies two integers. Let k, j, i be pairwise distinct such that $k, j, i \leq n$, for any $n \in \mathbb{N}$. A function $\mathsf{mult}_{k,j;i} \in \mathsf{RPP}^n$ exists such that:

$$\begin{matrix} x_1 \\ \cdots \\ x_i \\ \cdots \\ x_n \end{matrix} \left[\begin{array}{c} \\ \mathsf{mult}_{k,j;i} \\ \\ \end{array}\right] \begin{matrix} x_1 \\ \cdots \\ x_i + \\ \cdots \\ x_n \end{matrix} \begin{cases} \overbrace{(\ x_j + \ldots + x_j)}^{|x_k|} & \text{if } x_k, x_j \text{ have same sign} \\ \overbrace{(-x_j - \ldots - x_j)} & \text{if } x_k, x_j \text{ have different sign} \end{cases} .$$

The function $\mathsf{mult}_{k,j;i}$ behaves in accordance with the intended meaning whenever the ancilla x_i is initially set to 0. In that case $\mathsf{mult}_{k,j;i}$ yields the value

[3] The order of the first five elements has been carefully devised in order to avoid index changes of arguments due to the removal of intermediate arguments to satisfy the linearity constraint of the If.

$x_j \times x_k$ in position i. We define:

$$\mathsf{mult}_{k,j;i} = \mathsf{It}^n_{k,\langle i,j\rangle}\left[\mathsf{inc}_{1;2}\right] \fatsemi \mathsf{If}^n_{k,\langle i,j\rangle}\left[\mathsf{If}^2\left[\mathsf{I},\mathsf{I},\mathsf{N}\right], \mathsf{If}^2\left[\mathsf{I},\mathsf{I},\mathsf{I}\right], \mathsf{If}^2\left[\mathsf{N},\mathsf{I},\mathsf{I}\right]\right]$$

which gives the result by first nesting finite iterations as one may expect and then setting the correct sign.

A function that encodes a bounded minimization. Let $\mathsf{F}_{i;j} \in \mathsf{RPP}^n$ with $i \neq j$ and $i, j \leq n$. For any $\langle x_1, \ldots, x_i, \ldots, x_n\rangle$ and any $y \in \mathbb{N}$, which we call *range*, we look for the minimum integer v such that both $0 \leq v$ and $\mathsf{F}_{i;j}\langle x_1, \ldots, x_{i-1}, x_i + v, x_{i+1}, \ldots, x_n\rangle$ returns a negative value in position j, when a such v exists. If v does not exist we simply return $|y|$. Formally, the function $\mathsf{min}(\mathsf{F}_{i;j}) \in \mathsf{RPP}^{n+4}$ is:

$$\begin{matrix} x_1 \\ \cdots \\ x_n \\ x_{n+1} \\ 0 \\ 0 \\ x_{n+4} \end{matrix}\left[\begin{array}{c}\mathsf{min}(\mathsf{F}_{i;j})\end{array}\right]\begin{matrix} x_1 \\ \cdots \\ x_n \\ x_{n+1} + \\ 0 \\ 0 \\ x_{n+4} \end{matrix}\begin{cases} v & \text{whenever } 0 \leq v < |x_{n+4}| \text{ is such that} \\ & \mathsf{F}_{i;j}\langle \ldots, x_{i-1}, x_i + v, x_{i+1}, \ldots\rangle = \langle \ldots, y_{j-1}, z', y_{j+1}, \ldots\rangle \text{ and } z' < 0 \\ & \text{and, for all } u \text{ such that } 0 \leq u < v, \\ & \mathsf{F}_{i;j}\langle \ldots, x_{i-1}, x_i + u, x_{i+1}, \ldots\rangle = \langle \ldots, y_{j-1}, z, y_{j+1}, \ldots\rangle \text{ and } z \geq 0; \\ |x_{n+4}| & \text{otherwise.} \end{cases}$$

The function $\mathsf{min}(\mathsf{F}_{i;j})$ behaves as just described whenever we properly set the ancillae $x_{n+1}, x_{n+2}, x_{n+3}$ to 0 (we remark that x_{n+2}, x_{n+3} are temporary arguments, so their output is expected to be set back to zero) and we set x_{n+4} to contain the range y. Specifically, x_{n+2} stores a copy of the minimum we search and x_{n+3} is the flag that we rise as soon as we find such a minimum. We define $\mathsf{min}(\mathsf{F}_{i;j})$ as follows:

$$\begin{aligned} &\mathsf{min}(\mathsf{F}_{i;j}) := \\ &\quad \mathsf{It}\left[(\mathsf{F}_{i;j} \| \mathsf{I}^3) \fatsemi \mathsf{If}^{n+3}_{j,\langle n+1,n+2\,n+3\rangle}\left[\mathsf{I}^3, \mathsf{I}^3, (\mathsf{If}\left[\mathsf{I}^2, \mathsf{inc}_{2;1}, \mathsf{I}^2\right] \fatsemi \mathsf{S}_3)\right] \fatsemi (\mathsf{F}_{i;j} \| \mathsf{I}^3)^{-1} \fatsemi (\mathsf{S}_i \| \mathsf{S}_{n+2})\right] \fatsemi \\ &\quad \mathsf{If}^{n+4}_{n+3,\langle n+1,n+4\rangle}\left[\mathsf{I}, \mathsf{inc}_{2;1}, \mathsf{I}\right] \fatsemi \\ &\quad \left(\mathsf{It}\left[(\mathsf{F}_{i;j} \| \mathsf{I}^3) \fatsemi \mathsf{If}^{n+3}_{j,\langle n+1,n+2\,n+3\rangle}\left[\mathsf{I}^3, \mathsf{I}^3, \mathsf{S}_3\right)\right] \fatsemi (\mathsf{F}_{i;j} \| \mathsf{I}^3)^{-1} \fatsemi (\mathsf{S}_i \| \mathsf{S}_{n+2})\right]\right)^{-1}. \end{aligned}$$

The first line of the above definition iterates its whole argument as many times as specified by the value of the range in x_{n+4}. When applying $\mathsf{F}_{i;j}$ to its arguments a possible case is that the j-th output of $\mathsf{F}_{i;j}$ is negative. If x_{n+3} is still equal to 0, then the current value of x_{n+2} is the result of the bounded minimization; we copy it in x_{n+1} and we set x_{n+3} to 1, i.e. freezing the value just written in x_{n+1}. Before doing anything else, it is necessary to unfold the last application of $\mathsf{F}_{i;j}$ by means of ${\mathsf{F}_{i;j}}^{-1}$. Then, in any case, it is necessary to increase both x_i and x_{n+2} because we still may need both to supply a new value to $\mathsf{F}_{i;j}$ and to look for the result. The second line adds to x_{n+1} the absolute value of x_{n+4} in case the searched minimum has not been found. Finally, the last line clean the temporary arguments x_{n+2}, x_{n+3} by undoing the computation of the the first line, but the (possible) update of the result, viz. a magisterial application of the Bennett's trick [2].

Of course, for any $k, r \leq m \in \mathbb{N}$, with $m \geq n+4$ and $r \neq k$, the following natural generalization $\mathsf{min}_{r;k}(\mathsf{F}_{i;j}) \in \mathsf{RPP}^m$ of $\mathsf{min}(\mathsf{F}_{i;j})$ exists:

$$\begin{matrix} x_1 \\ \cdots \\ \\ x_k \\ \\ \cdots \\ x_m \end{matrix}\left[\begin{matrix} \\ \mathsf{min}_{r;k}(\mathsf{F}_{i;j}) \\ \\ \\ \\ \\ \end{matrix}\right]\begin{matrix} x_1 \\ \cdots \\ \\ x_k + \\ \\ \cdots \\ x_m \end{matrix}\begin{cases} v & \text{whenever } 0 \leq v < |x_r| \text{ is such that} \\ & \mathsf{F}_{i;j}\langle \ldots, x_{i-1}, x_i + v, x_{i+1}, \ldots\rangle = \langle \ldots, y_{j-1}, z', y_{j+1}, \ldots\rangle \text{ and } z' < 0 \\ & \text{and, for all } u \text{ such that } 0 \leq u < v, \\ & \mathsf{F}_{i;j}\langle \ldots, x_{i-1}, x_i + u, x_{i+1}, \ldots\rangle = \langle \ldots, y_{j-1}, z, y_{j+1}, \ldots\rangle \text{ and } z \geq 0; \\ \\ \\ |x_r| & \text{otherwise} \quad . \end{cases}$$

Whenever the least value v exists such that both $0 \leq v < |x_r|$ such that the j-th output component of $\mathsf{F}_{i;j}\langle x_1, \ldots, x_{i-1}, x_i + v, x_{i+1}, \ldots, x_n\rangle$ is negative, the above function adds v to the k-th output argument. Otherwise, the k-th output argument will eventually be added by the absolute value of the argument r.

5. Cantor pairing functions

Pairing functions provide a mechanism to uniquely encode two natural numbers into a single natural number [31]. We show how to represent Cantor pairing functions as functions of RPP restricted on natural numbers, albeit it is possible to re-formulate the pairing function on integers (see [27].) It follows that, for sake of conciseness, we shall systematically neglect to specify the relational behavior of the coming definitions of functions negative input values.

Definition 2. Cantor pairing is a pair of isomorphisms $\mathrm{Cp} : \mathbb{N}^2 \longrightarrow \mathbb{N}$ and $\mathrm{Cu} : \mathbb{N} \longrightarrow \mathbb{N}^2$ which embed $\mathbb{N}^2$ into $\mathbb{N}$:

$$\mathrm{Cp}(x, y) = y + \sum_{i=0}^{x+y} i \tag{5}$$

$$\mathrm{Cu}(z) = \left\langle z - \sum_{i=0}^{k-1} i \;,\; (k-1) - x \right\rangle \qquad \text{where } k \text{ is the least value such that, both } k \leq z \text{ and } z - \textstyle\sum_{i=0}^{k} i < 0.$$

The pairing functions in (5) rely on the notion of triangular number $T_n = \sum_{i=0}^{n} i$, for any $n \in \mathbb{N}$.

Theorem 1. *For every $x \in \mathbb{N}$, the functions:*

$$\begin{matrix} x \\ 0 \\ 0 \end{matrix}\left[\begin{matrix} \\ \mathrm{Tn} \\ \\ \end{matrix}\right]\begin{matrix} x \\ 0 \\ (\sum_{i=0}^{x} i) \end{matrix} \qquad \textit{and} \qquad \begin{matrix} x \\ y \\ 0 \end{matrix}\left[\begin{matrix} \\ \mathrm{Cp} \\ \\ \end{matrix}\right]\begin{matrix} x \\ y \\ (\sum_{i=0}^{x+y} i) + y \end{matrix}$$

exist in RPP^3.

Proof. We define $\mathrm{T2} \in \mathsf{RPP}^2$ and $\mathrm{T3} \in \mathsf{RPP}^3$ satisfying,

$$\begin{matrix} x_1 \\ x_2 \end{matrix}\left[\begin{matrix} \\ \mathrm{T2} \\ \\ \end{matrix}\right]\begin{matrix} x_1 + 1 \\ x_2 + |x_1 + 1| \end{matrix} \qquad \begin{matrix} x_1 \\ x_2 \\ x_3 \end{matrix}\left[\begin{matrix} \\ \mathrm{T3} \\ \\ \end{matrix}\right]\begin{matrix} x_1 \\ x_2 + x_1 \\ x_3 + x_2 x_1 + \sum_{i=0}^{x_1} i \end{matrix} .$$

by T2 $:= \mathsf{S}^2_1 \fatsemi \mathsf{inc}^2_{1;2}$ and T3 $:= \mathsf{It}^3_{1,\langle 2,3\rangle}$ [T2]. Therefore, we can define the functions as follows Tn $:=$ T3 $\fatsemi \mathsf{dec}^3_{1;2}$ and Cp $:= \mathsf{inc}^3_{2;1} \fatsemi \left\rangle {}^{1,2,3}_{1,3,2} \right\langle^3 \fatsemi$ Tn. □

We now show that an operator kCu exists which we can think of as being the kernel of Cu. It allows to compute the least value k that generates the (least) triangular number $\sum_{i=0}^{k} i$ which let $z - (\sum_{i=0}^{k} i) < 0$.

Lemma 2 (The kernel kCu)**.** *A function* kCu $\in \mathsf{RPP}^9$ *exists such that:*

$$\begin{matrix} x_1 \\ 0 \\ 0^7 \end{matrix} \left[\begin{matrix} \\ \text{kCu} \\ \\ \end{matrix} \right] \begin{matrix} x_1 \\ 0 + \textit{least } v < x_1 \textit{ such that } x_1 - (\sum_{i=0}^{v} i) < 0 \\ 0^7 \end{matrix}$$

for every $x_1 \in \mathbb{N}$.

Proof. There exists $H_{3;4} \in \mathsf{RPP}^4$ such that:

$$\begin{matrix} 0 \\ 0 \\ x_3 \\ x_4 \end{matrix} \left[\begin{matrix} \\ H_{3;4} \\ \\ \end{matrix} \right] \begin{matrix} 0 \\ 0 \\ x_3 \\ x_4 - \sum_{i=0}^{x_3} i \end{matrix}$$

for every $x_3, x_4 \in \mathbb{N}$. By Theorem 1, we define

$$H_{3;4} = \left(\left\rangle {}^{1,2,3,4}_{3,2,1,4} \right\langle^4 \fatsemi (\text{Tn} \| \mathsf{I}) \right) \fatsemi \mathsf{dec}^4_{3;4} \fatsemi \left(\left\rangle {}^{1,2,3,4}_{1,2,3,4} \right\langle^4 \fatsemi (\text{Tn} \| \mathsf{I}) \right)^{-1} .$$

The final step is using $H_{3;4}$ as the parameter of the bounded minimization:

$$\begin{aligned} \text{kCu} = \; & \left(\left\rangle {}^{1,2,3,4,5,6,7,8,9}_{8,2,3,4,5,6,7,1,9} \right\langle^9 \fatsemi (\mathsf{min}(H_{3;4}) \| \mathsf{I}^1) \right) \fatsemi \mathsf{inc}^9_{5;9} \fatsemi \\ & \left(\left\rangle {}^{1,2,3,4,5,6,7,8,9}_{8,2,3,4,5,6,7,1,9} \right\langle^9 \fatsemi (\mathsf{min}(H_{3;4}) \| \mathsf{I}^1) \right)^{-1} \fatsemi \left\rangle {}^{1,2,3,4,5,6,7,8,9}_{1,9,3,4,5,6,7,8,2} \right\langle^9 . \end{aligned}$$

□

Theorem 3 (Representing Cu : $\mathbb{N}^2 \longrightarrow \mathbb{N}$ in RPP)**.** *A function* Cu $\in \mathsf{RPP}^{11}$ *exists such that, for every* $z \in \mathbb{N}$*:*

$$\begin{matrix} z \\ 0 \\ 0 \\ 0^8 \end{matrix} \left[\begin{matrix} \\ \text{Cu} \\ \\ \end{matrix} \right] \begin{matrix} z \\ z - (\sum_{i=0}^{v-1} i) \\ (v-1) - (z - (\sum_{i=0}^{v-1} i)) \\ 0^8 \end{matrix} ,$$

where v *is the least value such that* $v \leq z$ *and* $z - (\sum_{i=0}^{v} i) < 0$*. So* $z - (\sum_{i=0}^{v-1} i)$ *is the first component of the pair that* z *represents under Cantor's pairing and* $(v-1) - (z - (\sum_{i=0}^{v-1} i))$ *is the second one.*

Proof. It is enough to follow the standard pattern: (i) start by rearranging inputs; (ii) find v using kCu $\in \mathsf{RPP}^9$; (iii) compute $\sum_{i=0}^{v-1} i$ by Tn $\in \mathsf{RPP}^3$; (iv) compute $(v-1) - (z - (\sum_{i=0}^{v-1} i))$ and $z - (\sum_{i=0}^{v-1} i)$; (v) a permutation which rearranges the outputs as required concludes the definition. □

We remark that Theorem 1, Lemma 2 and Theorem 3 can be extended to a functions that operate on values of $\mathbb{Z}$, not only of $\mathbb{N}$, suitably managing signs. We refer to [27] for some discussions on the point.

6. Expressiveness of RPP

We begin by showing how to represent stacks as natural numbers in RPP. I.e., given $x_1, x_2, \ldots, x_n \in \mathbb{N}$ we can encode them into $\langle x_n, \ldots, \langle x_2, x_1 \rangle \ldots \rangle \in \mathbb{N}$ by means of a sequence of push on a suitable ancillae, while a corresponding sequence of pop can decompose it as usual.

Proposition 4 (Representing stacks in RPP)**.** *Functions* $\mathsf{push}, \mathsf{pop} \in \mathsf{RPP}^{13}$ *exist such that:*

$$\begin{matrix} s \\ x \\ 0^{11} \end{matrix}\left[\begin{array}{c}\mathsf{push}\end{array}\right]\begin{matrix} \mathrm{Cp}(s,x) \\ 0 \\ 0^{11} \end{matrix} \qquad\qquad \begin{matrix} \mathrm{Cp}(s,x) \\ 0 \\ 0^{11} \end{matrix}\left[\begin{array}{c}\mathsf{pop}\end{array}\right]\begin{matrix} s \\ x \\ 0^{11} \end{matrix} \quad ,$$

for every value $s \in \mathbb{N}$ *(the "stack") and* $x \in \mathbb{N}$ *(the element one has to push on or pop out the stack.)*

Proof. The function $\mathrm{zClean} := (\mathsf{I}^2 \| \,\mathrm{Cu}) \fatsemi \mathsf{dec}^{13}_{4;1} \fatsemi \mathsf{dec}^{13}_{5;2} \fatsemi (\mathsf{I}^2 \| \,\mathrm{Cu})^{-1}$ is such that:

$$\begin{matrix} s \\ x \\ \mathrm{Cp}(s,x) \\ 0^{10} \end{matrix}\left[\begin{array}{c}\mathrm{zClean}\end{array}\right]\begin{matrix} 0 \\ 0 \\ \mathrm{Cp}(s,x) \\ 0^{10} \end{matrix}$$

so, $\mathsf{push} := (\mathrm{Cp} \| \mathsf{I}^{10}) \fatsemi \mathrm{zClean} \fatsemi \left(\begin{smallmatrix}1,2,3\\3,2,1\end{smallmatrix}\right)^3 \| \mathsf{I}^{10})$ and $\mathsf{pop} := \mathsf{push}^{-1}$. □

6.1. RPP *is* PRF*-complete*

RPP is expressive enough to represent the class PRF of Primitive Recursive Functions [5, 26], which we recall for easy of reference. PRF is the smallest class of functions on natural numbers that:

- contains the functions $\mathsf{0}^n(x_1, \ldots, x_n) := 0$, the successors $\mathsf{S}^n_i(x_1, \ldots, x_n) := x_i$ and the projections $\mathsf{P}^n_i(x_1, \ldots, x_n) := x_i$ for all $1 \leq i \leq n$,
- is closed under composition, viz. PRF includes the function $f(\overrightarrow{x}) := h(g_1(\overrightarrow{x}), \ldots, g_m(\overrightarrow{x})))$ whenever there are $g_1, \ldots, g_m, h \in \mathsf{PRF}$ of suitable arity, and
- is closed under primitive recursion, viz. PRF includes the function f defined by means of the schema $f(\overrightarrow{x}, 0) := g(\overrightarrow{x})$ and $f(\overrightarrow{x}, y+1) := h(f(\overrightarrow{x}, y), \overrightarrow{x}, y)$ whenever there are $g, h \in \mathsf{PRF}$ of suitable arity.

In the following, PRF^n denotes the class of functions $f \in \mathsf{PRF}$ with arity n.

Definition 3 (RPP-definability of any $f \in \mathsf{PRF}$**).** Let $n, a \in \mathbb{N}$ and $f \in \mathsf{PRF}^n$. We say that f is RPP^{n+a+1}-definable whenever there is a function $d_f \in \mathsf{RPP}^{n+a+1}$ such that, for every $x, x_1, \ldots, x_n \in \mathbb{N}$:

$$\begin{matrix} x \\ x_1 \\ \cdots \\ x_n \\ 0^a \end{matrix}\left[\begin{array}{c} \\ d_f \\ \\ \end{array}\right]\begin{matrix} x + f(x_1, \ldots, x_n) \\ x_1 \\ \cdots \\ x_n \\ 0^a \end{matrix} .$$

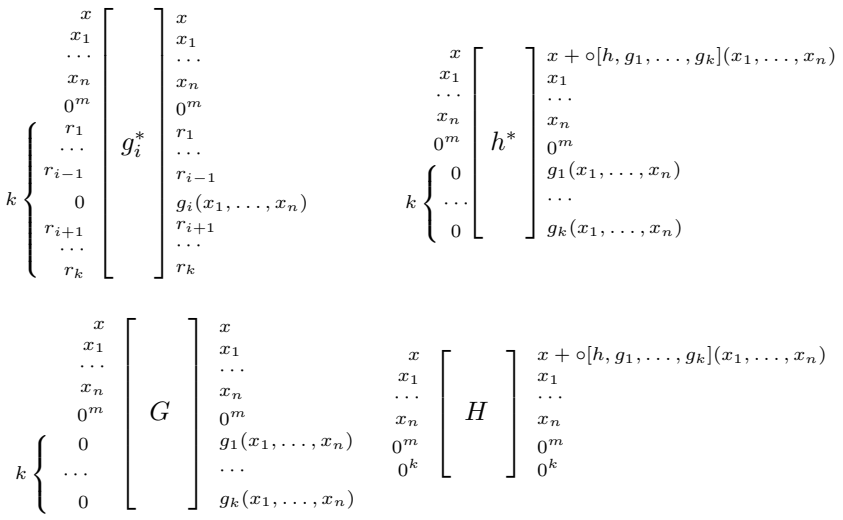

Table 1: Composition component relations

Three observations are worth doing. Definition 3 relies on $a+1$ ancillae (a temporary arguments.) If $f \in \mathsf{PRF}^n$ is RPP^{n+a+1}-definable, then f is also RPP^{n+k}-definable for any $k \geq a+1$ by using *weakening* (cf. Section 3). Definition 3 improves the namesake one in [27, Def.3.1, p.236] because it gets rid of an output line which plays the role of a "waste bin" able to record a computation trace.

Theorem 4 (RPP is PRF-complete)**.** *If $f \in \mathsf{PRF}^n$ then $\underline{\overline{f}}$ is RPP^{n+a+1}-definable, for some $a \in \mathbb{N}$.*

Proof. The proof is given by induction on the definition of the primitive recursive function f.

- If f is $\mathsf{0}^n$ then let $\underline{\overline{\mathsf{0}^n}} := \mathsf{I}^{n+1}$, where $a = 0$.
- If f is S_i^n then let $\underline{\overline{\mathsf{S}_i^n}} := \mathsf{inc}_{i+1;1}^{n+1} \fatsemi (\mathsf{S} \| \mathsf{I}^n)$, where $a = 0$.
- If f is P_i^n $(1 \leq i \leq n)$ then, let $\underline{\overline{\mathsf{P}_i^n}} := \mathsf{inc}_{i+1;1}^{n+1}$, where $a = 0$.
- Let $f = \circ[h, g_1, \ldots, g_k]$ where $g_1, \ldots, g_k \in \mathsf{PRF}^n$ and $h \in \mathsf{PRF}^k$, for some $k \geq 1$. Therefore, by inductive hypothesis, there are $\underline{\overline{g_1}} \in \mathsf{RPP}^{n+a_1+1}$, ..., $\underline{\overline{g_k}} \in \mathsf{RPP}^{n+a_k+1}$ and $\underline{\overline{h}} \in \mathsf{RPP}^{k+a_0+1}$ for $a_0, \ldots, a_k \in \mathbb{N}$.

  Let $m = \max\{a_0, \ldots, a_k\}$. By using $\underline{\overline{g_i}}$, it is easy to build $g_i^* \in \mathsf{PRF}^{n+m+k+1}$ computing the reversible permutation described in the top-left of Table 1. The sequential composition can be used to compute the permutation G described in the bottom-left of Table 1. By using G and h, it is easy to compute

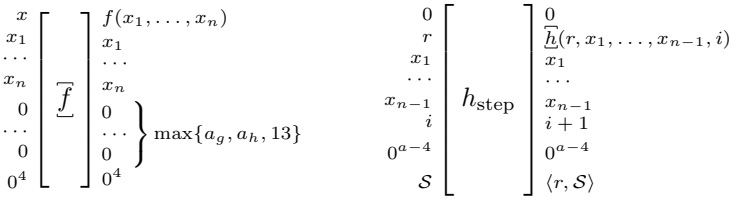

Table 2: Primitive Recursion Components

h^* in the top-right of Table 1. Last, $H \in \mathsf{RPP}^{n+(m+k)+1}$ in the bottom-right of Table 1 is $\circ[h, g_1, \ldots, g_k]$ and we define it as $h^* \fatsemi G^{-1}$.

H improves the representation of composition sketched in [27] because it reduces the number of ancillae by re-using them to compute one after the other $g_1, \ldots, g_k \in \mathsf{PRF}^n$ and $h \in \mathsf{PRF}^k$.

- Let $f \in \mathsf{PRF}^n$ where $n \geq 1$ be defined by means of the primitive recursion on $g \in \mathsf{PRF}^{n-1}$ and $h \in \mathsf{PRF}^{n+1}$. By inductive hypothesis there are $\overline{g} \in \mathsf{RPP}^{(n-1)+a_g+1}$ and $\overline{h} \in \mathsf{RPP}^{(n+1)+a_h+1}$ for $a_g, a_h \in \mathbb{N}$.

  The definition of $\overline{f}$ requires: (i) a temporary argument to store the result of the previous recursive call; (ii) a temporary argument to stack intermediate results; (iii) a temporary argument to index the current iteration-step; (iv) a temporary argument to contain the final result which is not modified when the computation is undone for cleaning temporary values; and (v) in different times, a_g temporary arguments to compute $\overline{g}$, a_h temporary arguments to compute $\overline{h}$, 13 ancillae to push and pop (see Proposition 4.)

  Let $a := 4 + max\{a_g, a_h, 13\}$. Our goal is to define $\overline{f} \in \mathsf{RPP}^{n+a+1}$ which behaves as described in the left of Table 2:

  1. A first block of operations which we call F_1 reorganizes the inputs of $\overline{f}$ in Table 2 to obtain $0, 0, x_1, \ldots, x_{n-1}, \overbrace{0, \ldots, 0}^{a-2}, x_n, x$.
  2. The result of the previous step becomes the input of $\mathsf{I}^1 \| \overline{g} \| \mathsf{I}^{a-a_g}$ which yields $0, g(x_1, \ldots, x_{n-1}), x_1, \ldots, x_{n-1}, \overbrace{0, \ldots, 0}^{a-2}, x_n, x$. Note that x_n is the argument that must to drive the iteration.
  3. The previous point supplies the arguments to the x_n-times applications of the recursive step:

$$h_{\text{step}} := (\overline{h} \| \mathsf{I}^{a-(a_h+2)}) \fatsemi \mathsf{S}^{n+a-1}_{n+2} \fatsemi \mathsf{push}^{n+a-1}_{2;n+a-1} \fatsemi (\chi \| \mathsf{I}^{n+a-3})$$

  by means of $\mathsf{It}\,[h_{\text{step}}] \| \mathsf{I}^1$. The relation that h_{step} implements is depicted in Table 2. Thus, h_{step} takes $n + a - 1$ input arguments only among the $n+a+1$ available because the first one serves to the identity and the other

one drives the iteration. The function of h_{step} first applies $\underline{\overline{h}}$ and then reorganizes arguments for the next iteration. Specifically, (i) it increments the step-index in the argument of position $(n+2)$, (ii) it pushes the result of the previous step, which is in position 2, on top of the stack , which is in its last ancilla, (iii) it exchanges the first two arguments, the first one containing the result of the last iterative step and the second one containing a fresh zero produced by push.

4. We get $0, f(x_1, \ldots, x_n), x_1, \ldots, x_{n-1}, x_n, \overbrace{0, \ldots, 0}^{a-2}, x_n, x$ from the previous point. We add the result to the last line by means of $\mathsf{inc}_{2;n+a+1}$ by yielding $0, f(x_1, \ldots, x_n), x_1, \ldots, x_{n-1}, x_n, \overbrace{0, \ldots, 0}^{a-2}, x_n, x + f(x_1, \ldots, x_n)$.
5. We then conclude by unwinding the first three steps.

Summing up, $\underline{\overline{h}}$ is:

$$F_1 \fatsemi \big(\mathsf{I}^1 \| \overline{g} \| \mathsf{I}^{a-a_g}\big) \fatsemi \big(\mathsf{lt}\,[h_{\text{step}}] \| \mathsf{I}^1\big) \fatsemi \mathsf{inc}_{2;n+a+1} \fatsemi \big(\mathsf{lt}\,[h_{\text{step}}] \| \mathsf{I}^1\big)^{-1} \fatsemi \big(\mathsf{I}^1 \| \overline{g} \| \mathsf{I}^{a-a_g}\big)^{-1} \fatsemi {F_1}^{-1} \quad .$$

□

6.2. RPP *is* PRF*-sound*

The mere intuition should support the evidence that every $f \in \mathsf{RPP}$ has a representative inside PRF we can obtain via the bijection which exists between $\mathbb{Z}$ and $\mathbb{N}$. More precisely, every RPP-permutation can be represented as a PRF-endofunction on tuples of natural numbers which encode integers. Details on how formalizing the embedding of RPP into PRF are in [27].

7. Conclusions

7.1. The intensioanl nature of the functions in RPP

Let A be the Ackermann function, example of total computable function with two arguments that cannot belong to PRF because its growth rate is too high. Kuznekov shows that a primitive recursive function F exists with input arity 1 such that $F^{-1}(x) = A(x, x)$. It is worth to remark that the inverse of F is not primitive recursive, because A is not. References are not immediate, because the original result [15] is in Russian: some details about Kuznekov's proof are in [27, 33]. Moreover in [34, Exercise 5.7, p.25] Kuznecov's result is slightly reformulated by the statement saying that "primitive recursive functions do not form a group under composition."

By Theorem 4, the function:

$$\begin{matrix} w \\ z \\ 0^k \end{matrix} \left[\begin{array}{c} \\ \overline{\underline{F}} \\ \\ \end{array} \right] \begin{matrix} w + F(z) \\ z \\ 0^k \end{matrix}$$

exists and belongs to RPP, for some k. Proposition 2 implies that $\overline{\underline{F}}^{-1}$ is in RPP, is computable and is such that:

$$\begin{matrix} w \\ z \\ 0^k \end{matrix}\left[\begin{matrix} \\ \overline{\underline{F}} \\ \\ \end{matrix}\right]\begin{matrix} w + F(z) \\ z \\ 0^k \end{matrix}\left[\begin{matrix} \\ \overline{\underline{F}}^{-1} \\ \\ \end{matrix}\right]\begin{matrix} w \\ z \\ 0^k \end{matrix} \quad .$$

This highlights the strongly intensional nature of the reversible functions inside RPP. The inversion of a permutation p undoes what p executes, so $\overline{\underline{F}}^{-1}$ algorithmically searches the value x such that $A(x, x) = z$, for any given argument z we can pass to F, by using $F(z)$ as bound for the iteration.

7.2. Weakened no-cloning

We start recalling a theorem from stated in [22].

Theorem 5 (No-cloning theorem on ESRL [22])**.** *Consider permutations in $\mathbb{Z}^{k+2} \to \mathbb{Z}^{k+2}$ where $k \in \mathbb{N}$. No permutation can always assume the same value on two output lines independently of the input values.*

Proof. Permutations are bijections, thus they must necessarily range over the whole co-domain. □

However, programming strategies exist to partially circumvent Theorem 5 in our setting, exactly like strategies exist in quantum computing for developing quantum error correcting codes. Such strategies boil down to using ancillary variables constrained to assume specific values so yielding the cloning of information in a limited way. As a simple instance, we note that the general increment defined in Section 4 , when we assume x_i to be zero, allows to clone — to copy — the value of an argument.

7.3. Programming languages and programming patterns

RPP also suggests the design of a programming language with a specific programming pattern that one can already identify in many of the "programming examples" inside the preceding sections. Generally speaking, every function of RPP can be divided into three parts. The first part transforms the inputs in outputs of which only a part is effectively the one which contains what we aim at computing. That set of outputs can be recorded by the second part of the function for future use. "Recording" means cloning them on a suitable set of further output lines. Finally, the third part undoes what the first part computed, so getting back to the inputs. This is what the literature knows as Bennett's trick. However, what we advocate, is that it can be made into a pattern of reference to design programs in a reversible setting.

7.4. Dictionary

Sometimes, there is some confusion about the mathematical notions related to reversibility. Our paper rests on standard definitions provided by Nicolas Bourbaki. We quote the Definition 10 of [3]:

> Let f be a mapping of A into B. The mapping f is said to be injective, or an injection, if any two distinct elements of A have distinct images under f. The mapping f is said to be surjective, or a surjection if $f(A) = B$. If f is both injective and surjective, it is said to be bijective, or a bijection.

and some phrases, that can be found few lines below:

> If f is bijective, we sometimes say that f puts A and B in one-to-one correspondence. A bijection of A onto A is called a permutation of A.

We are convinced that the definitions above are standard, but we remark that injection is sometimes used to mean inclusion (see [8, p.31]) and one-to-one is used to mean Bourbaki's injective function.

The notion of invertible function is much less standard and is used in many ways in literature. Such an adjective is associated either to injective maps whenever the domain-codomain of the inverse does not matter (cf. [5, p.2]), or to bijective maps (whenever domain-codomain matter, e.g. in [18] invertible is intended as the conjunction of left and right invertible). It is worth notice that, in both cases, the inversion of a computable function is computable.

Yet, we remark that many algebraic and categorical notions are related to the above ones (see [17]) and, finally, that reversible function means function that can be defined via aReversible Turing Machine.

7.5. Future work

Firstly, a comparison between the expressiveness of ESRL and RPP is underway. Secondly, we would like to extend RPP in many ways without leaving total computations. One extension would consist on adding primitives able to hide ancillary variables by fixing their value, so to simplify programming in the lines of [35]. Another would allow the definition of functions with different arity and co-arity, by including some specific bijections as primitives. This way we would extend RPP to a class with functions that would not necessarily be permutations. Thirdly, of course, we are working on a natural extension of RPP which is complete with respect to Kleene partial recursive functions.

References

[1] H. B. Axelsen and R. Glück. What do reversible programs compute? In *14th International Conference on Foundations of Software Science and Computational Structures*, volume 6604 of *Lecture Notes in Computer Science*, pages 42–56. Springer, 2011.

[2] C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Develop.*, 17:525–532, 1973.

[3] N. Bourbaki. *Elements of mathematics. Theory of sets.* Addison-Wesley, 1968. Translated from the French.

[4] F. B. Cannonito and M. Finkelstein. On primitive recursive permutations and their inverses. *Journal of Symbolic Logic*, 34(4):634–638, 12 1969.

[5] N. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

[6] P. Giannini, E. Merelli, and A. Troina. Interactions between computer science and biology. *Theoretical Computer Science*, 587:1 – 2, 2015. Interactions between Computer Science and Biology.

[7] S. Guerrini, S. Martini, and A. Masini. Towards A theory of quantum computability. *CoRR*, abs/1504.02817, 2015.

[8] P. Halmos. *Naive Set Theory*. Undergraduate Texts in Mathematics. Springer, 1960.

[9] G. Jacopini and P. Mentrasti. Generation of invertible functions. *Theor. Comput. Sci.*, 66(3):289–297, 1989.

[10] G. Jacopini and G. Sontacchi. General recursive functions in a very simply interpretable typed lambda-calculus. *Theor. Comput. Sci.*, 121(1&2):169–178, 1993.

[11] R. P. James and A. Sabry. Information effects. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 73–84, 2012.

[12] I. Kalimullin. *Computability and Models: Perspectives East and West*, chapter On Primitive Recursive Permutations, pages 249–258. Springer, 2003.

[13] V. V. Koz'minykh. On the representation of partial recursive functions as superpositions. *Algebra and Logic*, 11(3):153–167, 1972.

[14] V. V. Koz'minykh. Representation of partial recursive functions with certain conditions in the form of superpositions. *Algebra and Logic*, 13(4):238–240, 1974.

[15] A. V. Kuznecov. On primitive recursive functions of large oscillation. *Doklady Akademii Nauk SSSR*, 71:233–236, 1950. In russian.

[16] Y. Lafont. Towards an algebraic theory of boolean circuits. *Journal of Pure and Applied Algebra*, 184(2–3):257–310, 2003.

[17] S. Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer New York, 1998.

[18] S. Lane and G. Birkhoff. *Algebra*. Chelsea Publishing Series. Chelsea Publishing Company, 1999.

[19] Y. Lecerf. Machines de turing réversibles. *Comptes Rendus Hebdomadaires des Séances de L'académie des Sciences*, 257:2597–2600, 1963.

[20] A. I. Malcev. *Algorithms and recursive functions*. Wolters-Noordhoff, 1970. Translated from the first Russian ed. by Leo F. Boron, with the collaboration of Luis E. Sanchis, John Stillwell and Kiyoshi Iseki.

[21] A. B. Matos. Linear programs in a simple reversible language. *Theor. Comput. Sci.*, 290(3):2063–2074, 2003.

[22] A. B. Matos. Register reversible languages.
Available at `http://www.dcc.fc.up.pt/~acm/questionsv.pdf`, January 2016.

[23] J. McCarthy. The inversion of functions defined by turing machines. In C. Shannon and J. McCarthy, editors, *Automata Studies, Annals of Mathematical Studies*, 34, pages 177–181. Princeton University Press, 1956.

[24] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proceedings of the 1967 22Nd National Conference*, ACM '67, pages 465–469, New York, NY, USA, 1967. ACM.

[25] K. Morita. Reversible computing and cellular automata – a survey. *Theoretical Computer Science*, 395(1):101 – 131, 2008.

[26] P. Odifreddi. *Classical recursion theory: the theory of functions and sets of natural numbers*. Studies in logic and the foundations of mathematics. North-Holland, 1989.

[27] L. Paolini, M. Piccolo, and L. Roversi. A class of reversible primitive recursive functions. *Electronic Notes in Theoretical Computer Science*, 322(18605):227–242, 2016.

[28] K. S. Perumalla. *Introduction to Reversible Computing*. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2013.

[29] J. Robinson. General recursive functions. *Proceedings of the American Mathematical Society*, 1:703–718, 1950.

[30] H. Rogers. *Theory of recursive functions and effective computability*. McGraw-Hill series in higher mathematics. McGraw-Hill, 1967.

[31] A. L. Rosenberg. *The Pillars of Computation Theory: State, Encoding, Nondeterminism*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[32] P. Rozsa. *Recursive functions*. Academic Press, 1967.

[33] S. G. Simpson. Foundations of mathematics. Departement of Mathematics, University of Pennsylvania. `http://www.personal.psu.edu/t20/notes/fom.pdf`, 2009.

[34] R. Soare. *Recursively Enumerable Sets and Degrees: A Study of Computable Functions and Computably Generated Sets.* Perspectives in Mathematical Logic. Springer, 1987.

[35] M. K. Thomsen, R. Kaarsgaard, and M. Soeken. Ricercar: A language for describing andÂ rewriting reversible circuits with ancillae and its permutation semantics. In J. Krivine and J.-B. Stefani, editors, *Reversible Computation: 7th International Conference, RC 2015, Grenoble, France, July 16-17, 2015, Proceedings*, number 9138 in Lecture Notes in Theoretical Computer Science, pages 200–215, 2015.

[36] T. Toffoli. Reversible computing. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 632–644. Springer, 1980.

[37] T. Yokoyama, H. B. Axelsen, and R. Glück. Principles of a reversible programming language. In A. Ramírez, G. Bilardi, and M. Gschwind, editors, *Proceedings of the 5th Conference on Computing Frontiers, 2008, Ischia, Italy, May 5-7, 2008*, pages 43–54. ACM, 2008.

[38] M. Zorzi. On quantum lambda calculi: a foundational perspective. *Mathematical Structures in Computer Science*, 2014. http://dx.doi.org/10.1017/S0960129514000425.