AperTO - Archivio Istituzionale Open Access dell'Università di Torino

A class of Recursive Permutations which is Primitive Recursive complete

This is the author's manuscript

Original Citation:

Availability:

This version is available http://hdl.handle.net/2318/1734163 since 2020-06-23T13:22:30Z

Published version:

DOI:10.1016/j.tcs.2019.11.029

Terms of use:

Open Access

Anyone can freely access the full text of works made available as "Open Access". Works made available under a Creative Commons license can be used according to the terms and conditions of said license. Use of all other works requires consent of the right holder (author or publisher) if not exempted from copyright protection by the applicable law.

(Article begins on next page)

19 April 2024

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, sign-change, successor, predecessor, swap), two notions of composition (sequential and parallel), one functional iteration and one functional selection. RPP is closed by inversion and it is expressive enough to encode Cantor pairing and the whole class of Primitive Recursive Functions.

Keywords: Reversible Computation, Unconventional Computing Models, Computable Permutations, Primitive Recursive Functions, Recursion Theory.

1. Introduction

Mainstream models of computations 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. Generally speaking, models of computations are neither backward deterministic nor 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 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

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)

list all the pairs $(0,5),(1,4),(2,3),(3,2),(4,1)$ and $(5,0)$ every 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. Since then, the interest for the reversible computing has slowly but incessantly grown.

A forcefully non-exhaustive list of references follows. Reversible Turing machines are defined in [1, 3, 13, 24] while [1, 2, 12, 25] 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 [33, 48]. Moreover, some research focused on reversible programming languages [14, 49]. Of course, the interest to build a comprehensive knowledge about reversible computation is wider than the mere interest for its computational theoretic aspects. The book [40] is a comprehensive introduction to the subject. It spans from low-power-consumption reversible hardware to emerging alternative computational models like quantum [11, 52] or bio-inspired [10] 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 Primitive Recursive Functions (PRF) [42, 46] whose relevance is sometimes traced to the slogan "programs which terminate but do not belong to PRF are rarely of practical interest."

We aim at formalizing a language that we identify as Reversible Primitive Permutations (RPP) and which retains the more interesting aspects of PRF. In analogy to PRF, our goal is to get a functional characterization of computability – but in a reversible setting, of course — because functions are handier in order to compose algorithms. Other models, like, for example, Turing machines-oriented ones are more convincing from an implementation view-point. The ability to represent Cantor pairing [43] is one of the main properties that the functional characterization we are looking for must satisfy. With Cantor pairing available it is possible to express all interesting total properties about the traces[1] of Turing machines, reversible or not. The other must-have property of our functional characterization is closure under inversion, which is something 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 played against it. First of all, we remark that the class of all (total) recursive permutations cannot be effectively enumerated (see [42, Exercise 4-6, p.55]). In [17] a constructive generation of all the recursive permutations is given starting from primitive recursive permutations. Since no enumeration exists for the latter, none can exist for the former. Worst, the class of primitive recursive permuta-

[1]Kleene's T_n predicate, Kleene's normal form theorem and technical tools related to them [9, 36, 46].

tions is not closed under inversion [19, 37, 46]. Since the above negative results hold also for the class of elementary permutations [8, 16, 18], there is no hope to find any effective description neither of the class of recursive permutations nor of the classes of primitive recursive permutations and of elementary permutations.

Comparison with the known literature. Our quest to identify the reversible analogous of PRF must be throughly related to the following works.

- The programming language SRL restricts LOOP, a language for programming PRF functions [32]. Matos introduces SRL and some of its variants in [29]. He is the first using $\mathbb{Z}$ — the natural numbers with sign — as the ground type for classes of reversible functions. We share the choice with him. The work [29] focuses on the algebraic aspects of his languages and its relations with matrix groups.

  The study of the classes of functions that SRL variants can identify is part of Matos' work. Variants of SRL are complete with respect to reversible boolean circuits [30]. Proving that some given variant of SRL is PRF-complete, i.e. that it represents all the functions in PRF, naturally ends up with the quest to encode the "test for 0" like in the proof that LOOP and PRF are equivalent [32]. We *conjecture* that no variant of Matos' languages exists able to simulate a conditional control on the flow of execution that, instead, RPP contains by definition. Of course, proving the conjecture false, would promote SRL to be (in a reasonable sense) the minimal reversible and PRF-complete language.

- The precursor of this work is [37]. It introduces the class of functions RPRF which is closed by inversion and is PRF-complete. The completeness of RPRF relies on *built-in* Cantor pairings. In this paper we show that we can get rid of *built-in* Cantor pairings inside RPP. With no *built-in* pairing operators at hand the design of RPP relies on operators which are more fine-grained as compared to the ones used for the definition of RPRF. This orients the programming style to enjoy a couple of interesting features. Inside RPP it is natural avoiding to save the entire history of a calculation within a single argument, i.e. into a single ancilla. This allows to clean the garbage at the end of the simulation of any $f \in$ PRF, something which is reminiscent of the simulation of Turing machines devised in [4].

- Finally we discuss [12]. It introduces the class of reversible functions $\mathcal{RI}$ which is as expressive as Kleene's partial recursive functions [9, 36]. Therefore, the focus of [12] 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 a 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.

Contents. 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 permutations. 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 [29].

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:

$$\begin{array}{c} m \\ n \end{array}\left[\ \texttt{sum}\ \right]\begin{array}{c} m+n \\ n \end{array} \quad . \tag{1}$$

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 $f^{-1} \in$ RPP of every given $f \in$ RPP. I.e., :

$$\begin{array}{c} p \\ n \end{array}\left[\ \texttt{sum}^{-1}\ \right]\begin{array}{c} p-n \\ n \end{array} \tag{2}$$

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 [29]. We do not internalize it to avoid mutually recursive definitions in RPP.

Concerning the *expressiveness*, RPP is closed under inversion and it 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 and Cantor pairing. The embedding relies on various key aspects of reversible computing. The principal ones are: (i) the use of ancillary variables to clone information and (ii) the compositional programming under the pattern that Bennett's trick dictates (cf. Section 7.)

Contributions. RPP is a total class of reversible functions closed under inversion, which is PRF-complete and sound. We think that it can play a key role in the setting of reversible computations analogous to that played by PRF in classical recursion theory. In particular, it can be used to devise the analogous of Kleene's normal form theorem and Kleene's predicates in the reversible setting; it can be also used to formalize a reversible arithmetic as the primitive recursive one (a.k.a. Skolem arithmetic).

2. The class RPP of Reversible Primitive Permutations

The identification of RPP merges ideas and observations on Primitive Recursive Functions (PRF) [44, 28, 29, 36], Toffoli's class of boolean circuits [48] and

Lafont's algebraic theory on circuits [21]. We quickly recall the crucial ideas we borrowed from the above papers in order to formalize the Definition 1.

Toffoli's boolean circuits are invertible because they avoid erasing information. This is why the boolean circuits in [48] 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 suggests 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 on $\mathbb{N}$ up to some existing isomorphism between $\mathbb{Z}$ and $\mathbb{N}$.

The core of the composition schemes of RPP comes from [21]. 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.

2.1. 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 prefer 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 neutral 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 will often 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 (abbreviated as RPP) is a sub-class of Reversible Permutations[2]. 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 *sign-change* 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:

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

[2]For sake of precision, we focus on Total Reversible Recursive Endo-Functions on $\mathbb{Z}^n$ for some $n \in \mathbb{N}$.

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:

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

- The *binary rewiring permutation* χ belongs to RPP^2. The functional notation is the expected one:

  $$\chi\langle x,y\rangle := \langle y,x\rangle \ .$$

  We use two interchangeable relational notations:

  $$\begin{matrix}x\\y\end{matrix}\left[\ \chi\ \right]\begin{matrix}y\\x\end{matrix} \ \text{ and } \ \Big)\begin{matrix}1,2\\2,1\end{matrix}\Big(^{2} \ \begin{matrix}x\\y\end{matrix}\left[\ \Big)\begin{matrix}1,2\\2,1\end{matrix}\Big(^{2}\ \right]\begin{matrix}y\\x\end{matrix} \quad .$$

  The second one explicitly represents (i) the arity, (ii) the input sequence of the arguments which is the upper sequence of numbers; and (iii) the output sequence of arguments in the lower sequence of numbers.

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

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

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

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

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

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

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

  where $\cdot$ is the composition of sequences (cf. Section 2.1).

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

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

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

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, x is the last argument of $\mathsf{It}\,[f]$ in order to apply f to the first n arguments of $\mathsf{It}\,[f]$ itself.

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

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

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

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. □

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 will often abbreviate it as:

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

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

The following proposition certifies that identifying the inverse of a term inside RPP is a simple task. I.e., given the representation of a function f in RPP, generating f^{-1} is *effective* and such that $(y, x) \in f^{-1}$ if and only if $(x, y) \in f$.

Proposition 1 (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:*

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

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

Proof. By induction on the definition of f. □

Proposition 1 shows that RPP allows for a very smooth, syntactically driven, definition of the operation that makes the inverse of a given f effectively available. Other methods can exist. For example, [31, 41] pursue a sort of "brute force" method which is not at all syntax directed.

Of course, RPP is syntactically redundant. For example, both the parallel composition of two occurrences of the unary identity and the series-composition of two swaps yield the identity with arity 2. Moreover, the selection needs not be defined on three (higher-order) arguments as we do. Superfluous ingredients help to make the programming with RPP a bit more reasonably easy task. ⋆

Proposition 2 (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:

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

We conclude with some comments on the relevance of having functions with identical input and output arity in RPP. Toffoli's works on reversible boolean circuits influenced our choice. However, in [39] we extend a reversible language with a bijective built-in map from $\mathbb{Z} \times \mathbb{Z}$ to $\mathbb{Z}$ such that $\langle 0, 0 \rangle \mapsto 0$. Clearly, the input/output symmetry breaks up and we move from permutations to isomorphisms. The built-in bijection allows to generate as many ancillary arguments as needed for fully developing reversible computations. Once finished, ancillae will be repackaged into a single argument.

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:

$$\begin{array}{r} x_1 \\ \cdots \\ x_m \\ x_{m+1} \\ \cdots \\ x_{m+n} \end{array} \left[\begin{array}{c} \\ \\ \mathsf{w}^n(f) \\ \\ \\ \end{array} \right] \begin{array}{l} \left.\begin{array}{l} y_1 \\ \cdots \\ y_m \end{array}\right\} = f\,\langle x_1, \ldots, x_m \rangle \\ x_{m+1} \\ \cdots \\ x_{m+n} \end{array} \qquad ,$$

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, sign-change, successor and predecessor. For every $i \leq n$, the following functions in RPP^n exist:

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

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, we allow to use $\mathsf{I}, \mathsf{N}_i, \mathsf{S}_i$ and P_i in place of $\mathsf{I}^n, \mathsf{N}^n_i, \mathsf{S}^n_i$ and P^n_i.

Generalized rewiring permutations. The literature offers terminology and notation related to the reversible computing which is far from being uniform. Just as an example, the class ~~od~~ of *invertible functions* of [9, p.2] differs from the ⋆ namesake class that [27] defines. From [27] we take that a permutation on a set S is a bijection on S, i.e. an endo-bijection on S. Very often, the intended meaning of "permutation" is "finite permutation on a given (finite) set S." Generally speaking, this is not our case at least because, as we will see in Section 5, we deal with pairing inside RPP. When dealing with permutations of RPP whose only goal is to re-arrange the finite set $\{1, \ldots, k\}$ of indexes of their arguments, we talk about *rewiring permutations*.

Proposition 3 (Rewiring Permutations in RPP.)**.** *Each rewiring permutation with arity and co-arity equal to k belongs to* RPP^k.

Proof. Fixed a rewiring permutation to represent, it is enough to suitably compose *identity*, *binary rewiring permutation*, *series composition*, *parallel composition* and *weakening*. □

Let $\{i_1, \ldots, i_n\} = \{1, \ldots, n\}$ and $\rho : \mathbb{N}^n \longrightarrow \mathbb{N}^n$ The notation for a rewiring permutation in RPP^n, sending 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. is:

$$\left\rbrace \begin{smallmatrix} i_1 & ,\ldots, & i_m \\ \rho(i_1) & ,\ldots, & \rho(i_m) \end{smallmatrix} \right\lbrace^n \ . \qquad (3)$$

Generalized finite iteration. Let $f \in \mathsf{RPP}^n$ and $m \geq n+1$. Let $\langle i\rangle$, $L = \langle i_1, \ldots, i_n\rangle$ and $\langle j_1, \ldots, j_{m-n-1}\rangle$ be a partition of natural numbers in the interval $\langle 1, \ldots, m\rangle$. 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: $\mathsf{It}^m_{i,L}\,[f] := \left\rmoustache\begin{smallmatrix}1,\ldots,n,n+1,n+2,\ldots,m\\ i_1,\ldots,i_n,\ i\ ,\ j_1\ ,\ldots,j_{m-n-1}\end{smallmatrix}\right\lmoustache^m \fatsemi (\mathsf{It}\,[f] \| \mathsf{I}^{m-n-1}) \fatsemi \left(\left\rmoustache\begin{smallmatrix}1,\ldots n,n+1,n+2,\ldots,m\\ i_1,\ldots i_n,\ i\ ,\ j_1\ ,\ldots,j_{m-n-1}\end{smallmatrix}\right\lmoustache^m\right)^{-1}$.

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 $\langle i\rangle$, $L = \langle i_1, \ldots, i_n\rangle$ and $\langle j_1, \ldots, j_{m-n-1}\rangle$ be a partition of natural numbers in the interval $\langle 1, \ldots, m\rangle$. 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:

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

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 further simplify programming inside RPP by relying on *temporary arguments*, *ancillary arguments* or *ancillae*. They are additional arguments that can: (i) hold the result of the computation; (ii) temporarily record copies of values for later use; (iii) temporarily store intermediate results. Without loss of generality, we use ancillae in a very disciplined way in order to simplify their understanding. Tipically, ancillae are initialized to zero. We cannot forget (reversibly) the content of ancillae, so they have to be carefully considered in compositions albeit sometimes we neglect their contents (tipically, just forwarding them). Notions that recall the meaning of ancillae are common to the studies on reversible and quantum computation. They are "temporary" lines, storages, channels, bits and variables [48].

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:

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

We define them as:

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

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:

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

The function $\mathsf{less}_{i,j,p,q;k}$ is expected to behave in accordance with the intended meaning whenever the arguments x_p, x_q are initially contains 0, viz. they are ancillae. Tipically, x_k will be used as ancilla initialized to zero and it is not a temporary argument. If the value of x_k is initially 0, then $\mathsf{less}_{i,j,p,q;k}$ 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.

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

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

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

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

where F is

$$\mathsf{If}^5_{5,\langle 1,2,3,4 \rangle}\left[\mathsf{If}^4_{4,\langle 1,2,3 \rangle}[\mathsf{SameSign}, \mathsf{S}^3_1, \mathsf{S}^3_1], \mathsf{If}^4_{4,\langle 1,2,3 \rangle}[\mathsf{I}^3, \mathsf{I}^3, \mathsf{S}^3_1], \mathsf{If}^4_{4,\langle 1,2,3 \rangle}[\mathsf{I}^3, \mathsf{I}^3, \mathsf{SameSign}]\right]$$

and $\quad \mathsf{SameSign} := \ \mathsf{dec}_{2;3} \fatsemi \mathsf{If}^3_{3,\langle 1,2\rangle}[\mathsf{S}^2_1, \mathsf{I}^2, \mathsf{I}^2] \fatsemi (\mathsf{dec}_{2;3})^{-1} \quad .$
The *series composition* at the lines (4) and (6) are one the inverse of the other. The leftmost function at line (4) moves the five relevant arguments in the first five positions[3]. The rightmost function at line (4) duplicates x_i and x_j into the ancillae x_p and x_q, respectively. The functions at line (6) undo what those ones at line (4) have done. In between them, at line (5) the function $\mathsf{F} \parallel \mathsf{I}^{n-5}$ receives a given $\langle x_k, x_p + |x_i|, x_q + |x_j|, x_i, x_j, x_{j_1}, \dots, x_{j_{n-5}}\rangle$ as argument and operates on its first five elements. We explain $\mathsf{F} \in \mathsf{RPP}^5$ by the following case analysis scheme:

		$x_i > 0$	$x_i = 0$	$x_i < 0$		
$\mathsf{If}^5_{5,\langle 1,2,3,4\rangle}[$	$\mathsf{If}^4_{4,\langle 1,2,3\rangle}[$	$\mathsf{SameSign}$,	S^3_1,	S^3_1	$]$	$x_j > 0$
,	$\mathsf{If}^4_{4,\langle 1,2,3\rangle}[$	I^3,	I^3,	S^3_1	$]$	$x_j = 0$
,	$\mathsf{If}^4_{4,\langle 1,2,3\rangle}[$	I^3,	I^3,	$\mathsf{SameSign}$	$]]$	$x_j < 0$

.

For example, let us assume $x_j < 0$ and $x_j < 0$. The nested selections of F eventually apply $\mathsf{SameSign}$ to a tuple of three elements $\langle x_k, x_p + |x_j|, x_q + |x_i|\rangle$. The effect of $\mathsf{dec}_{2;3}$ is to update the third element of the tuple yielding $\langle x_k, x_p + |x_j|, x_q + |x_i| - x_p - |x_j|\rangle$. So, whenever the initial value of the temporary arguments x_p and x_q is 0, we can deduce which among x_i and x_j is greater than the other. The value of x_k is incremented only when the difference is positive. $\mathsf{SameSign}$ concludes by unfolding its local subtraction.

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:

$$\begin{array}{r} x_1 \\ \dots \\ x_i \\ \dots \\ x_n \end{array} \left[\ \mathsf{mult}_{k,j;i} \ \right] \begin{array}{l} x_1 \\ \dots \\ (x_i + |x_k| \times |x_j|) \times \begin{cases} 1 & \text{if } x_k, x_j \text{ have same sign} \\ -1 & \text{if } x_k, x_j \text{ have different sign} \end{cases} \\ \dots \\ x_n \end{array} \quad .$$

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 $|x_j| \times |x_k|$ in position i which is multiplied by -1 if, and only if, x_j and x_k have different sign. We define:

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

which gives the result by first nesting the iterations of $\mathsf{inc}_{2;1}$ inside an explicit iteration (acting on 3 arguments) and then setting the correct sign.

[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.

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 $v \geq 0$ 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 $|y|$. Summarizing, $\mathsf{min}(\mathsf{F}_{i;j}) \in \mathsf{RPP}^{n+4}$ is:

$$\begin{array}{r} x_1 \\ \cdots \\ x_n \\ x_{n+1} \\ 0 \\ 0 \\ x_{n+4} \end{array} \left[\ \rotatebox{90}{$\mathsf{min}(\mathsf{F}_{i;j})$}\ \right] \begin{array}{l} x_1 \\ \cdots \\ x_n \\ x_{n+1} + \\ 0 \\ 0 \\ x_{n+4} \end{array} \begin{cases} v & \text{whenever } 0 \leq v < |x_{n+4}| \text{ is such that} \\ & \mathsf{F}_{i;j}\langle \ldots, x_{i-1}, x_i + v, x_{i+1}, \ldots\rangle = \langle \ldots, y_{j-1}, z', y_{j+1}, \ldots\rangle \text{ and } z' < 0 \\ & \text{and, for all } u \text{ such that } 0 \leq u < v, \\ & \mathsf{F}_{i;j}\langle \ldots, x_{i-1}, x_i + u, x_{i+1}, \ldots\rangle = \langle \ldots, y_{j-1}, z, y_{j+1}, \ldots\rangle \text{ and } z \geq 0; \\ |x_{n+4}| & \text{otherwise.} \end{cases}$$

The function $\mathsf{min}(\mathsf{F}_{i;j})$ behaves in accordance with the here above specification when: (i) the arguments x_{n+1}, x_{n+2} and x_{n+3}, that we use as temporary arguments, are set to 0, and (ii) x_{n+4} is set to contain the range y. We plan to use the ancilla x_{n+3} as flag, the ancilla x_{n+2} as a counter from 0 to x_{n+4} and, the ancilla x_{n+1} as store for v (or 0). We define:

$$\begin{aligned} &\mathsf{min}(\mathsf{F}_{i;j}) := \\ &\quad \mathsf{It}\left[(\mathsf{F}_{i;j}\|\mathsf{I}^3) \fatsemi \mathsf{If}^{n+3}_{j,\langle n+1,n+2,n+3\rangle}\left[\mathsf{I}^3, \mathsf{I}^3, (\mathsf{If}\left[\mathsf{I}^2, \mathsf{inc}_{2;1}, \mathsf{I}^2\right] \fatsemi \mathsf{S}^3_3)\right] \fatsemi (\mathsf{F}_{i;j}\|\mathsf{I}^3)^{-1} \fatsemi (\mathsf{S}^n_i\|\mathsf{S}^3_2)\right] \fatsemi \\ &\quad \mathsf{If}^{n+4}_{n+3,\langle n+1,n+4\rangle}\left[\mathsf{I}, \mathsf{inc}_{2;1}, \mathsf{I}\right] \fatsemi \\ &\quad \left(\mathsf{It}\left[(\mathsf{F}_{i;j}\|\mathsf{I}^3) \fatsemi \mathsf{If}^{n+3}_{j,\langle n+1,n+2,n+3\rangle}\left[\mathsf{I}^3, \mathsf{I}^3, \mathsf{S}^3_3\right] \fatsemi (\mathsf{F}_{i;j}\|\mathsf{I}^3)^{-1} \fatsemi (\mathsf{S}^n_i\|\mathsf{S}^3_2)\right]\right)^{-1} . \end{aligned}$$

The first line of the here above definition iterates its whole argument as many times as specified by the value of the range in x_{n+4}:

$$\begin{array}{lr} (\mathsf{F}_{i;j}\|\mathsf{I}^3) \fatsemi \mathsf{If}^{n+3}_{j,\langle n+1,n+2,n+3\rangle}\left[\mathsf{I}^3, \mathsf{I}^3, (\mathsf{If}\left[\mathsf{I}^2, \mathsf{inc}_{2;1}, \mathsf{I}^2\right] \fatsemi \mathsf{S}^3_3)\right] \fatsemi (\mathsf{F}_{i;j}\|\mathsf{I}^3)^{-1} \fatsemi (\mathsf{S}^n_i\|\mathsf{S}^3_2) & \text{1-st step,} \\ \vdots & \\ \fatsemi (\mathsf{F}_{i;j}\|\mathsf{I}^3) \fatsemi \mathsf{If}^{n+3}_{j,\langle n+1,n+2,n+3\rangle}\left[\mathsf{I}^3, \mathsf{I}^3, (\mathsf{If}\left[\mathsf{I}^2, \mathsf{inc}_{2;1}, \mathsf{I}^2\right] \fatsemi \mathsf{S}^3_3)\right] \fatsemi (\mathsf{F}_{i;j}\|\mathsf{I}^3)^{-1} \fatsemi (\mathsf{S}^n_i\|\mathsf{S}^3_2) & k\text{-th step,} \\ \vdots & \\ \fatsemi (\mathsf{F}_{i;j}\|\mathsf{I}^3) \fatsemi \mathsf{If}^{n+3}_{j,\langle n+1,n+2,n+3\rangle}\left[\mathsf{I}^3, \mathsf{I}^3, (\mathsf{If}\left[\mathsf{I}^2, \mathsf{inc}_{2;1}, \mathsf{I}^2\right] \fatsemi \mathsf{S}^3_3)\right] \fatsemi (\mathsf{F}_{i;j}\|\mathsf{I}^3)^{-1} \fatsemi (\mathsf{S}^n_i\|\mathsf{S}^3_2) & x_{n+4}\text{-th step;} \end{array}$$

let us assume that the x_j is negative at step k. We have two cases.

- The first case is with x_{n+3} equal to 0. By design, this means that x_j has never become negative before. So, x_i contains the value v we are looking for. The firs step of:

$$\mathsf{If}\left[\mathsf{I}^2, \mathsf{inc}_{2;1}, \mathsf{I}^2\right] \fatsemi \mathsf{S}^3_3 \tag{7}$$

  is to store x_{n+2} into x_{n+1}. The reason is that, by design, the increments applied to x_{n+2} and x_i are always stepwise aligned. The second step of (7) is to increment x_{n+3}, preventing any further change of the value of x_{n+1} by the subsequent steps of the iteration.

- The second case is with x_{n+3} different from 0. By design, this means that x_j has become negative in some step before the current k-th one. So, (7) skips $\mathsf{inc}_{2;1}$. The increment S^3_2 that operates on x_{n+3} is harmless because it just reinforces the idea that the value v we were looking for is "frozen" in x_{n+1}.

After any occurrence of (7) it is necessary to unfold the last application of $\mathsf{F}_{i;j}$ by means of ${\mathsf{F}_{i;j}}^{-1}$. Then, increasing x_i and x_{n+2} supplies a new value to $\mathsf{F}_{i;j}$ and keeps x_{n+2} aligned with x_i.

After the first iteration completes, the evaluation follows with the second line $\mathsf{If}^{n+4}_{n+3,\langle n+1,n+4\rangle}\,[\mathsf{I}, \mathsf{inc}_{2;1}\,, \mathsf{I}]\,\fatsemi$ in the definition of $\mathsf{min}(\mathsf{F}_{i;j})$. It takes care of two cases.

- The value of x_{n+3} is not zero: somewhere in the course of the iteration $\mathsf{F}_{i;j}$ became negative. We have set x_{n+1} in accordance with that.
- The value of x_{n+3} is zero: $\mathsf{F}_{i;j}$ became negative in the course of the iteration. By definition, $\mathsf{inc}_{2;1}$ sets x_{n+1} to the value of the range in x_{n+4}.

The last line of $\mathsf{min}(\mathsf{F}_{i;j})$ undoes what the first line did, without altering the values of x_{n+1}. This is why $\mathsf{If}\,\left[\mathsf{I}^2, \mathsf{inc}_{2;1}, \mathsf{I}^2\right]$ is missing. In fact, the last line of $\mathsf{min}(\mathsf{F}_{i;j})$ is an application of Bennett's trick [3] in our functional programming setting.

5. Cantor pairing functions

Pairing functions provide a mechanism to uniquely encode two natural numbers into a single natural number [43]. 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 [37].)

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 $\mathbb{N}$:

$$\mathrm{Cp}(x, y) = x + \sum_{i=0}^{x+y} i \tag{8}$$

$$\mathrm{Cu}(z) = \left\langle z - \sum_{i=0}^{k-1} i \,\mathbf{,}\, (k-1) - (z - \sum_{i=0}^{k-1} i) \right\rangle \quad \text{where } k \text{ is the least value s.t. } k \leq z \text{ and } z < \textstyle\sum_{i=0}^{k} i.$$

The pairing functions in the above Equation (8) rely on the notion of triangular number $T_n = \sum_{i=0}^{n} i$, for any $n \in \mathbb{N}$.

Lemma 1. *For every x_1, x_2 and $x_3 \in \mathbb{N}$, the two following functions exist:*

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

and belong to RPP^2 *and* RPP^3*, respectively.*

Proof. Let T2 be $\mathsf{S}^2_1 \fatsemi \mathsf{inc}^2_{1;2}$ and let T3 $\in \mathsf{RPP}^3$ be $\mathsf{It}^3_{1,\langle 2,3\rangle}\,[\text{T2}]$. The result is immediate because we are interested in the behaviour of the operator only on positive numbers, viz. $x_1, x_2, x_3 \in \mathbb{N}$. $\square$

We shall systematically neglect to specify the relational behavior of the coming definitions of functions on negative input values.

Theorem 2. *For every $x \in \mathbb{N}$, the following function is in RPP^4:*

$$\begin{array}{c} x\\ y\\ 0\\ 0\end{array}\left[\begin{array}{c} \\ \mathrm{Cp}\\ \\ \\ \end{array}\right]\begin{array}{l} x\\ y\\ (\sum_{i=0}^{x+y} i)+x\\ 0\end{array}\quad .$$

Proof. Let Cp be $\mathsf{inc}^4_{1;2} \fatsemi \mathsf{inc}^4_{1;4} \fatsemi \left\rbrace\begin{smallmatrix}1,2,3,4\\2,3,1,4\end{smallmatrix}\right\lbrace^4 \fatsemi (\text{T3}\|\mathsf{I}) \fatsemi \mathsf{dec}^4_{1;2} \fatsemi \left\rbrace\begin{smallmatrix}1,2,3,4\\4,1,3,2\end{smallmatrix}\right\lbrace^4 \fatsemi \mathsf{dec}^4_{1;2}$. $\square$

We now turn to representing Cu. The computation of the difference between a given z and the triangular number $\sum_{i=0}^{k} i$, limited by z, is at the core of Cu.

Lemma 3 (Subtracting a triangular number)**.** *For every x_3 and $x_4 \in \mathbb{N}$, the following function exists:*

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

and belongs to RPP^4.

Proof. Let $H_{3;4}$ be:

$$\left(\left\rbrace\begin{smallmatrix}1,2,3,4\\3,2,1,4\end{smallmatrix}\right\lbrace^4 \fatsemi (\text{T3}\|\mathsf{I})\right) \fatsemi \mathsf{dec}^4_{3;4} \fatsemi \left(\left\rbrace\begin{smallmatrix}1,2,3,4\\3,2,1,4\end{smallmatrix}\right\lbrace^4 \fatsemi (\text{T3}\|\mathsf{I})\right)^{-1} \quad .$$

By Lemma 1 the proof is done. $\square$

We can supply $H_{3;4}$ to the minimization min in order for it to yield a series of "attempts" whose goal is to find when $H_{3;4}$ becomes negative in its 4th output:

$$\begin{array}{ccccc} \text{1-st attempt} & \text{2-nd attempt} & & j\text{-th attempt} & \\ \begin{array}{c} 0\\ 0\\ 0\\ z\end{array}\left[\begin{array}{c} \\ H_{3;4}\\ \\ \\ \end{array}\right]\begin{array}{l} 0\\ 0\\ 0\\ z-\sum_{i=0}^{0} i\end{array} & \begin{array}{c} 0\\ 0\\ 1\\ z\end{array}\left[\begin{array}{c} \\ H_{3;4}\\ \\ \\ \end{array}\right]\begin{array}{l} 0\\ 0\\ 1\\ z-\sum_{i=0}^{1} i\end{array} & \cdots & \begin{array}{c} 0\\ 0\\ j\\ z\end{array}\left[\begin{array}{c} \\ H_{3;4}\\ \\ \\ \end{array}\right]\begin{array}{l} 0\\ 0\\ j\\ z-\sum_{i=0}^{j} i\end{array} & \cdots \end{array}$$

The representation of Cu relies on the search here above.

Theorem 4 (Representing Cu : $\mathbb{N}^2 \longrightarrow \mathbb{N}$ in RPP)**.** *A function* Cu $\in \mathsf{RPP}^8$ *exists such that, for every $z \in \mathbb{N}$:*

$$\begin{array}{c} z\\ 0\\ 0\\ 0^5\end{array}\left[\begin{array}{c} \\ \mathrm{Cu}\\ \\ \\ \end{array}\right]\begin{array}{l} z\\ z-(\sum_{i=0}^{v-1} i)\\ (v-1)-(z-(\sum_{i=0}^{v-1} i))\\ 0^5\ \ ,\end{array}$$

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 pairing and $(v-1) - (z - (\sum_{i=0}^{v-1} i))$ is the second one.

Proof. Let Cu be:

$$\left\rfloor\begin{smallmatrix}1,2,3,4,5,6,7,8\\4,2,3,1,5,6,7,8\end{smallmatrix}\right\lfloor^{8} \fatsemi \mathsf{inc}^{8}_{4;8} \fatsemi \mathsf{min}(H_{3;4})$$
$$\fatsemi \mathsf{P}^{8}_{5} \fatsemi \left\rfloor\begin{smallmatrix}1,2,3,4,5,6,7,8\\1,2,5,4,3,6,7,8\end{smallmatrix}\right\lfloor^{8} \fatsemi (H_{3;4}\|\mathsf{I}^4) \fatsemi \mathsf{dec}^{8}_{4;3} \fatsemi \left\rfloor\begin{smallmatrix}1,2,3,4,5,6,7,8\\8,4,3,2,5,6,7,1\end{smallmatrix}\right\lfloor^{8}$$

The *series composition* $\left\rfloor\begin{smallmatrix}1,2,3,4,5,6,7,8\\4,2,3,1,5,6,7,8\end{smallmatrix}\right\lfloor^{8} \fatsemi \mathsf{inc}^{8}_{4;8}$ sets the input for $\mathsf{min}(H_{3;4})$ which receives the tuple $\langle 0,0,0,z,0,0,0,z\rangle$ and yields $\langle 0,0,0,z,k,0,0,z\rangle$ where k is the least value such that $k \leq z$ and $z < \sum_{i=0}^{k} i$. The predecessor P^{8}_{5} sets its 5th argument to the value $k-1$ which is required to correctly apply Cu (see Definition 2). The next rewiring sets the tuple $\langle 0,0,k-1,z,0,0,0,z\rangle$, argument of $H_{3;4}\|\mathsf{I}^4$ which produces $\langle 0,0,k-1,z-\sum_{i=0}^{k-1} i,0,0,0,z\rangle$. The value $z-\sum_{i=0}^{k-1} i$ is the first component of the pair of numbers we want to extract from the first argument of the whole Cu. The second component is $k-1-(z-\sum_{i=0}^{k-1} i)$ we obtain by applying $\mathsf{dec}^{8}_{4;3}$. The last rewiring rearranges the values as required. $\square$

As a remark, Theorem 2, Lemma 3 and Theorem 4 can be extended to functions that operate on $\mathbb{Z}$, not only on $\mathbb{N}$, suitably managing signs [37].

6. Expressiveness of RPP

We start to show 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 expected.

Proposition 4 (Representing stacks in RPP). *Functions* push, pop $\in \mathsf{RPP}^{10}$ *exist such that:*

$$\begin{array}{r|c|l} s & & \mathrm{Cp}(s,x)\\ x & \mathsf{push} & 0\\ 0^8 & & 0^8 \end{array} \qquad\qquad \begin{array}{r|c|l} \mathrm{Cp}(s,x) & & s\\ 0 & \mathsf{pop} & x\\ 0^8 & & 0^8 \end{array} \quad ,$$

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 zClean $:= (\mathsf{I}^2\|\,\mathrm{Cu}) \fatsemi \mathsf{dec}^{10}_{4;1} \fatsemi \mathsf{dec}^{10}_{5;2} \fatsemi (\mathsf{I}^2\|\,\mathrm{Cu})^{-1}$ is such that:

$$\begin{array}{r|c|l} s & & 0\\ x & & 0\\ \mathrm{Cp}(s,x) & \text{zClean} & \mathrm{Cp}(s,x)\\ 0^7 & & 0^7 \end{array}$$

so, push $:= (\mathrm{Cp}\|\mathsf{I}^6) \fatsemi \text{zClean} \fatsemi (\left\rfloor\begin{smallmatrix}1,2,3\\3,2,1\end{smallmatrix}\right\lfloor^{3}\|\mathsf{I}^6)$ and pop $:= \mathsf{push}^{-1}$. $\square$

6.1. RPP *is* PRF*-complete*

RPP is expressive enough to represent the class PRF of Primitive Recursive Functions [9, 36], which we recall for easy of reference. PRF is the smallest class of functions on natural numbers that:

- contains the functions $\mathtt{0}^n(x_1, \ldots, x_n) := 0$, the successors $\mathtt{S}_i^n(x_1, \ldots, x_n) := x_i + 1$ and the projections $\mathtt{P}_i^n(x_1, \ldots, x_n) := x_i$ 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 $x, x_1, \ldots, x_n \in \mathbb{N}$:

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

Three observations are worth doing. Definition 3 relies on $a + 1$ arguments used as ancillae. 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 [37, Def.3.1, p.236] because it gets rid of an output line: this line was a "waste bin" containing part of the computation trace that, eventually, was useless. The encoding proposed in this paper does not need the "waste bin" anymore.

Theorem 5 (RPP is PRF-complete)**.** *If* $f \in \mathsf{PRF}^n$ *then* $\boxed{f}$ *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 $\mathtt{0}^n$ then let $\boxed{\mathtt{0}^n} := \mathsf{I}^{n+1}$, where $a = 0$.
- If f is $\mathtt{S}_i^n$ then let $\boxed{\mathtt{S}_i^n} := \mathsf{inc}_{i+1;1}^{n+1} \fatsemi (\mathsf{S} \| \mathsf{I}^n)$, where $a = 0$.
- If f is $\mathtt{P}_i^n$ $(1 \leq i \leq n)$ then, let $\boxed{\mathtt{P}_i^n} := \mathsf{inc}_{i+1;1}^{n+1}$, 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 $\boxed{g_1} \in \mathsf{RPP}^{n+a_1+1}$, ..., $\boxed{g_k} \in \mathsf{RPP}^{n+a_k+1}$ and $\boxed{h} \in \mathsf{RPP}^{k+a_0+1}$ for $a_0, \ldots, a_k \in \mathbb{N}$.

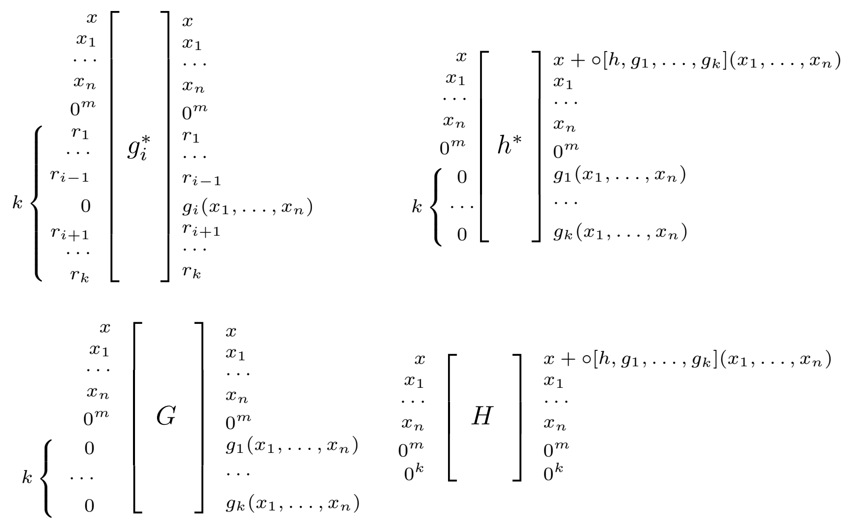

Table 1: Composition component relations

Let $m = \max\{a_0, \dots, a_k\}$. By using $\overline{\underline{g_i}}$, 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 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 $\overline{\underline{\circ[h, g_1, \dots, g_k]}}$ and we define it as $h^* \fatsemi G^{-1}$.

H improves the representation of composition sketched in [37] because it reduces the number of ancillae by re-using them to compute one after the other $g_1, \dots, g_k \in \mathsf{PRF}^n$ 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 $\overline{\underline{g}} \in \mathsf{RPP}^{(n-1)+a_g+1}$ and $\overline{\underline{h}} \in \mathsf{RPP}^{(n+1)+a_h+1}$ for $a_g, a_h \in \mathbb{N}$.

  The definition of $\overline{\underline{f}}$ 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, as many temporary arguments as a_g to compute $\overline{\underline{g}}$, as many temporary arguments as a_h to compute $\overline{\underline{h}}$ and 10 ancillae to push and pop elements in the course of the computation (see Proposition 4.)

  Let $a := 4 + max\{a_g, a_h, 10\}$. Our goal is to define $\overline{\underline{f}} \in \mathsf{RPP}^{n+a+1}$ which behaves as described in the left of Table 2:

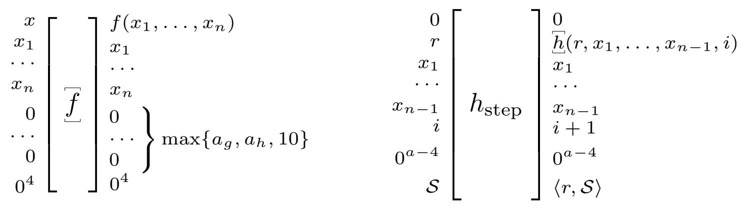

Table 2: Primitive Recursion Components

1. A first block of operations which we call F_1 reorganizes the inputs of $\underline{\overline{f}}$ in Table 2 to obtain $0, 0, x_1, \dots, x_{n-1}, \overbrace{0, \dots, 0}^{a-2}, x_n, x$.
2. The result of the previous step becomes the input of $\mathsf{I}^1 \| \underline{\overline{g}} \| \mathsf{I}^{a-a_g}$ which yields $0, g(x_1, \dots, x_{n-1}), x_1, \dots, x_{n-1}, \overbrace{0, \dots, 0}^{a-2}, x_n, x$. We notice that x_n is the argument that drives the iteration.
3. The previous point supplies the arguments to the x_n-times applications of the recursive step:
$$h_{\text{step}} := (\underline{\overline{h}} \| \mathsf{I}^{a-(a_h+2)}) \fatsemi \mathsf{S}^{n+a-1}_{n+2} \fatsemi \mathsf{push}^{n+a-1}_{2;n+a-1} \fatsemi (\chi \| \mathsf{I}^{n+a-3})$$
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 $0, f(x_1, \dots, x_n), x_1, \dots, x_{n-1}, x_n, \overbrace{0, \dots, 0}^{a-2}, x_n, x$ from the previous point. We add the result to the last line by means of $\mathsf{inc}_{2;n+a+1}$ by yielding $0, f(x_1, \dots, x_n), x_1, \dots, x_{n-1}, x_n, \overbrace{0, \dots, 0}^{a-2}, x_n, x + f(x_1, \dots, x_n)$.
5. We then conclude by unwinding the first three steps.

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

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

$\square$

6.2. RPP *is* PRF*-sound*

The mere intuition should support the evidence that every $f \in$ 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 [37].

7. Conclusions

In this paper we introduce the class RPP of functions which (i) is closed under inversion, (ii) is both PRF-complete and PRF-sound, (iii) provides a reasonable balance between conciseness and easiness of usage, suitable for recursion theoretical analysis, (vi) is a good candidate for extensions able to encompass recursive bijections which are not strictly total endo-functions — and recursive partial functions and (v) which is a good starting point to formalize classical results about the recursion theory in reversible settings.

We add some final comments on intensional aspects of RPP.

7.1. RPP *and Ackermann*

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. Kuznecov 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 [19] is in Russian: some details about Kuznecov's proof are in [37, 45] [39, 45] . Moreover in [46, 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 5, the function:

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

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

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

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. RPP *and no-cloning*

A form of no-cloning theorem holds in the setting of reversible computing in analogy with the no-colning theorem for the quantum computing.

Theorem 6 (No-cloning theorem [30])**.** *Let $\mathbb{Z}^{k+2} \to \mathbb{Z}^{k+2}$ represent the class of permutations with arity $k+2$, for any fixed $k \in \mathbb{N}$. Let $i \neq j$ belong to the initial segment $[1, k+2]$ of $\mathbb{N}$ such that i and j identify two arguments of the permutations in $\mathbb{Z}^{k+2} \to \mathbb{Z}^{k+2}$. No permutation can always return the same value on the arguments of position i and j, independently of their values.*

Proof. Permutations are bijections. So they are surjections which necessarily range over the whole co-domain. □

Quoting from [35, p.530]: "...the no-cloning theorem states that quantum mechanics does not allow unknown quantum states to be copied exactly, and places severe limitations on our ability to make approximate copies ...[*however*] the no-cloning theorem does not prevent all quantum states from being copied, it simply says that non-orthogonal quantum states cannot be copied."

The moral is that cloning is not available if we deal with quantum computations in general. However, in some cases programming strategies exist to circumvent Theorem 6 and we can use them for cloning when we move inside reversible computing. Cloning strategies boil down to using ancillary variables constrained to assume specific values. As a simple instance, the general increment $\mathsf{inc}_{j;i}$ of Section 4 clones — builds an exact a copy of — the j-th argument in its i-th argument exactly when this latter initially assumes value 0.

7.3. *Ancillae in* RPP

In the previous subsection we refer to ancillae as tools to circumvent no-cloning. For remarking once more that ancillae are harmless as far as reversible computing is concerned we focus on reversible computation as formalized by means of reversible Turing-machines (RTM) [1, 2, 3, 13]. This is like saying that a function is reversible when, and only when, some RTM exists that computes it. Any RTM crucially relies on an infinite tape that contains infinite blank cells: they supply an unbounded amount of ancillae that the computations can use at will. An RTM allows to duplicate data at the cost of using states to recall where and how many copies are generated, in order to revert the computation. This does not break the property of being in front of reversible computations. By the way, interesting discussions exist on how taking advantage from limited irreversible erasures on the ancillary tape when simulating irreversible computations inside reversible ones [4, 5, 25].

Turning our focus back on RPP we emphasize that our assumptions are analogous to those ones just recalled when working with RTM for simulating standard Turing-machines. We recall that RPP contains permutations only (Definition 1) and that no zero-constant function exist in it. The formalization of how the simulation works follows a path which is standard when comparing computational models [9, 28, 36]. For example, representing a function by a Turing-machine requires to fix how supplying data on the initial tape and where initially positioning its head. Concerning RPP, its simulation of PRF exploits permutations purposefully devised to behave as required when we supply the value 0 to some specific arguments, the ancillae. Summarizing, we define restriction-less permutations which behave as required as soon as the simulation conventions are satisfied.

8. Future work

We briefly discuss some of the possible directions we can follow to develop this work.

The formalization of RPP adhere to a standard recursion theory approach to identify computational classes. We see RPP as a formalism which lies at the same level of abstraction as the imperative programming languages SRL, and similar, that Matos introduces for dealing with reversible computations [29, 30]. So, we are focused on aspects more closely related to the design of paradigmatic programming languages. For example, an open question is whether RPP and SRL are equivalent. Answering it amounts to show that the selection — a built-in axiom of RPP — can be simulated in SRL.

Despite RPP is not a real programming language it can be a good place to start from for conceiving one. Even though the design of RPP does not start with the aim of computing with isomorphisms among types, like the point-free languages of [15, 34], most of the programming examples we give with it are point-free, especially when we iterate or select functions. Moreover, multiple input and output arity can be at the base of the representation and manipulation of immutable data structures like arrays. In the lines of [47], no obstacle seems to exist against the introduction of primitives that hide ancillae with the aim of simplifying programming. Finally, RPP looks flexible enough to incorporate specific bijections with different input and output arity like in [50]. This would extend RPP to a class of functions which are not necessarily permutations.

On the model theoretic point of view, it is obvious that RPP and ~~of~~ Lafont's ⋆
circuit classes in [21] share the same construction principles. Besides basic functions and two composition schemes, RPP has iteration and selection schemes by means of which, for example, we can give compressed descriptions of Lafont's circuits. Since RPP is PRF-complete a relation with Burroni's category of primitive recursive functions in [6] has to exist. Formalizing it will require to investigate ~~on~~ the link between the formalizations of the natural numbers ⋆
that the two approaches pursue. The categorical approach in [6] relies on a Peano-Lawvere axiom [23]. Instead, RPP see numbers in a Peano style, even though RPP does not contain any function which is constantly equal to 0.

Of course, the categorical structure in [6] is not the only one we can focus on to say what a model of RPP is. Models of RPP might well be instances of reversible PRO already used to study feedback-free reversible circuits [6, 7, 20, 21, 22]. Also, we would not be surprised we could use the category PInj of sets and functional injective relations as model of RPP, already used in [38] as a model of Janus, a reversible imperative programming language [26, 51].

Finally there is a quite close resemblance between the stack mechanism we implement to show that RPP is PRF-complete and the pebble game used to assess how efficient the simulation of a reversible computation by means of an irreversible one is [25, 4]. Given that RPP suggests a programming style where a computation can be undone as soon as the result it produces is at hand, and observing that information is piled up by coding it numerically, we plan to study if this features keep the overhead of the PRF simulation by means of RPP within an interesting range of space complexity.

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] H. B. Axelsen and R. Glück. On reversible turing machines and their function universality. *Acta Informatica*, 53(5):509–543, Aug 2016.

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

[4] C. H. Bennett. Time space trade-offs for reversible computation. *SIAM Journal on Computing*, 18(4):766–776, 1989.

[5] H. Buhrman, J. Tromp, and P. Vitányi. Time and space bounds for reversible simulation. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Automata, Languages and Programming*, pages 1017–1027, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[6] A. Burroni. Récursivité graphique (1e partie) : catégorie des fonctions récursives primitives formelles. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 27(1):49–79, 1986.

[7] A. Burroni. Higher-dimensional word problems with applications to equational logic. *Theoretical Computer Science*, 115(1):43 – 62, 1993.

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

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

[10] 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.

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

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

[13] G. Jacopini, P. Mentrasti, and G. Sontacchi. Reversible turing machines and polynomial time reversibly computable functions. *SIAM J. Discrete Math.*, 3(2):241–254, 1990.

[14] 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.

[15] R. P. James, A. Sabry, and J. Street. Theseus: A high level language for reversible computing. In *Work-in-progress Report for the Conference on Reversible Computation*, 2014.

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

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

[18] 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.

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

[20] Y. Lafont. Equational reasoning with 2-dimensional diagrams. In H. Comon and J.-P. Jounnaud, editors, *Term Rewriting*, pages 170–195, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

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

[22] Y. Lafont. Diagram rewriting and operads. In *Operads 2009*, volume 26 of *Séminaires et Congrès*, pages 163–179. Soc. Math. France, 2013.

[23] F. W. Lawvere. An elementary theory of the category of sets. *Proceedings of the National Academy of Sciences*, 52(6):1506–1511, 1964.

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

[25] M. Li and P. Vitányi. Reversibility and adiabatic computation: trading time and space for energy. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 452(1947):769–789, 1996.

[26] C. Lutz. Janus: a time-reversible language. *Letter to R. Landauer.*, 1986.

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

[28] A. I. Mal'cev. *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.

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

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

[31] 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.

[32] 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.

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

[34] S.-C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In D. Kozen, editor, *Mathematics of Program Construction*, pages 289–313, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[35] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, New York, NY, USA, 10th edition, 2011.

[36] 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.

[37] 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.

[38] L. Paolini, M. Piccolo, and L. Roversi. A certified study of a reversible programming language. In T. Uustalu, editor, *TYPES 2015 postproceedings*, volume 69 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2017.

[39] L. Paolini, M. Piccolo, and L. Roversi. On a class of reversible primitive recursive functions and its turing-complete extensions. *New Generation Computing*, 36(3):233–256, Jul 2018.

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

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

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

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

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

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

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

[47] M. K. Thomsen, R. Kaarsgaard, and M. Soeken. Ricercar: A language for describing and 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.

[48] 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.

[49] 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.

[50] T. Yokoyama, H. B. Axelsen, and R. Glück. Towards a reversible functional language. In A. D. Vos and R. Wille, editors, *Reversible Computation - Third International Workshop, RC 2011, Gent, Belgium, July 4-5, 2011. Revised Papers*, volume 7165 of *Lecture Notes in Computer Science*, pages 14–29. Springer, 2011.

[51] T. Yokoyama and R. Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 144–153, New York, NY, USA, 2007. ACM.

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