A Formal Verification of Reversible Primitive Permutations

Giacomo Maletto

1. The definition

1.1 Reversible computing

Reversible computing is a model of computation in which every process can be run backwards. Simply put, in a reversible setting any program takes inputs and gives outputs (like usual), but can also go the other way around: provided the output it can reconstruct the input. In a mathematical sense, every function is expected to be invertible.

Why do we care about such a thing?

Firstly, having a programming language in which every function (or even a subset of functions) is reversible could lead to interesting and practical applications.

But we can also imagine reversible computers, in which the underlying architecture is inherently reversible: Toffoli gates provides a way to do so. The opposite of reversibility is loss of information, which (for thermodynamic reasons) leads to loss of energy and heat dissipation. This means that a non-reversible gate dissipates energy each time information is discarded, while in principle a reversible computer wouldn't.

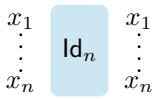
Lastly, reversible computing is directly related to quantum computing, as each operation in a quantum computer must be reversible.

1.2 Reversible Primitive Permutations

In the article I decided to formalize, the authors focus on providing a functional model of reversible computation. They develop an inductively defined set of functions, called **Reversible Primitive Permutations** or **RPP**, which are expressive enough to represent all Primitive Recursive Functions (we talk about what this means in section ?). Here is the definition that we will use:

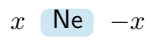
Definition 1 (Reversible Primitive Permutations)**.** The class of **Reversible Primitive Permutations** or RPP is the smallest subset of functions $\mathbb{Z}^n \to \mathbb{Z}^n$ satisfying the following conditions:

- The n-ary **identity** $\mathsf{Id}_n(x_1, \ldots, x_n) = (x_1, \ldots, x_n)$ belongs to RPP, for all $n \in \mathbb{N}$.

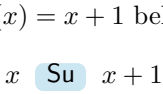


  The meaning of these diagrams should be fairly obvious: if the values on the left of a function are provided as inputs to that function, we get the values on the right as outputs.

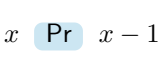
- The **sign-change** $\mathsf{Ne}(x) = -x$ belongs to RPP.



- The **successor function** $\mathsf{Su}(x) = x + 1$ belongs to RPP.



- The **predecessor function** $\mathsf{Pr}(x) = x - 1$ belongs to RPP.

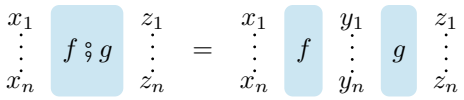


- The **swap** $\mathsf{Sw}(x, y) = (y, x)$ belongs to RPP.

- If $f : \mathbb{Z}^n \to \mathbb{Z}^n$ and $g : \mathbb{Z}^n \to \mathbb{Z}^n$ belongs to RPP, then the **series composition** $(f \fatsemi g) : \mathbb{Z}^n \to \mathbb{Z}^n$ belongs to RPP and is such that:

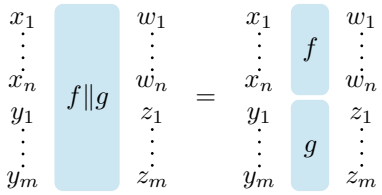
$$(f \fatsemi g)(x_1, \ldots, x_n) = g(f(x_1, \ldots, x_n)) = (g \circ f)(x_1, \ldots, x_n).$$

  We remark that $f \fatsemi g$ means that f is applied first, and then g, in opposition to the standard functional composition (denoted by $\circ$).



- If $f : \mathbb{Z}^n \to \mathbb{Z}^n$ and $g : \mathbb{Z}^m \to \mathbb{Z}^m$ belongs to RPP, then the **parallel composition** $(f \| g) : \mathbb{Z}^{n+m} \to \mathbb{Z}^{n+m}$ belongs to RPP and is such that:

$$(f \| g)(x_1, \ldots, x_n, y_1, \ldots, y_m) = (f(x_1, \ldots, x_n), g(y_1, \ldots, y_m)).$$



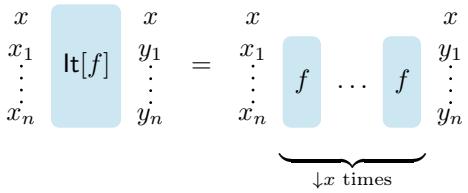
- If $f : \mathbb{Z}^n \to \mathbb{Z}^n$ belongs to RPP, then then **finite iteration** $\mathsf{It}[f] : \mathbb{Z}^{n+1} \to \mathbb{Z}^{n+1}$ belongs to RPP and is such that:

$$\mathsf{It}[f](x, x_1, \ldots, x_n) = (x, (\overbrace{f \circ \cdots \circ f}^{\downarrow x \text{ times}})(x_1, \ldots, x_n))$$

  where $\downarrow(\cdot) : \mathbb{Z} \to \mathbb{N}$ is defined as

$$\downarrow x = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}.$$

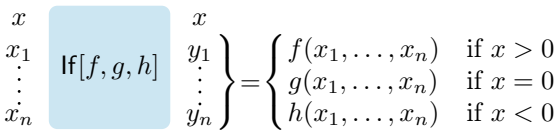
  This means that the function f is applied $\downarrow x$ times to $(x_1, \ldots, x_n)$.



- If $f, g, h : \mathbb{Z}^n \to \mathbb{Z}^n$ belongs to RPP, then the **selection** $\mathsf{If}[f, g, h] : \mathbb{Z}^{n+1} \to \mathbb{Z}^{n+1}$ belongs to RPP and is such that:

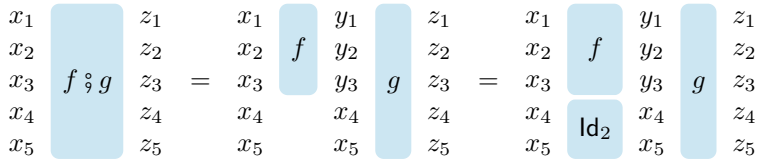
$$\mathsf{If}[f, g, h](x, x_1, \ldots, x_n) = \begin{cases} (x, f(x_1, \ldots, x_n)), & \text{if } x > 0 \\ (x, g(x_1, \ldots, x_n)), & \text{if } x = 0 \\ (x, h(x_1, \ldots, x_n)), & \text{if } x < 0 \end{cases}.$$

  We remark that the argument x which determines which among f, g and h must be used cannot be among the arguments of f, g and h, as that would break reversibility.



Remark 1. If we have two functions of different arity, for example $f : \mathbb{Z}^3 \to \mathbb{Z}^3$ and $g : \mathbb{Z}^5 \to \mathbb{Z}^5$, then we will still write $f \fatsemi g$ to mean the function with arity $\max(3, 5) = 5$ given by $(f \| \mathsf{Id}_2) \fatsemi g$. In general, the arity of the "smaller" function

can be enlarged by a suitable parallel composition with the identity. The same goes for the arguments of the selection $\mathsf{If}[f, g, h]$.

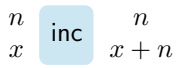


1.3 Some examples

In order to get accustomed to this definition, let's see some examples.

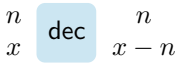
Increment and decrement Let's try to imagine what addition should look like in RPP. Of course, addition is usually thought of as a function which takes two inputs and yields their sum: something like $\mathsf{add}(x, y) = x + y$. But notice that this operation is not reversible: given only the output (the value $x + y$) it is impossible to obtain the original values (x, y). As we will see, every function in RPP is reversible, so we will not be able to define addition in this way.

Instead, we can define a function inc in RPP which, given $n \in \mathbb{N}$ and $x \in \mathbb{Z}$, yields

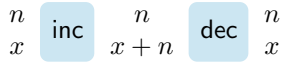


If n is negative the output is just (n, x). The fact that the above diagram is only valid for $n \in \mathbb{N}$ might bother some of you; we'll explain later why it is so, and how we can also make it work for $n \in \mathbb{Z}$.

For now let's focus on the output: we don't just have $x + n$ but also n, and indeed, given both n and $x + n$ we can reconstruct n (obviously) and x (by $(x + n) - n$). As a matter of fact, the following function dec also belongs to RPP:



and if we try to compose inc and dec we get this remarkable result:



and similarly for $\mathsf{dec} \fatsemi \mathsf{inc}$. So indeed dec is the inverse of inc, and we can write $\mathsf{dec} = \mathsf{inc}^{-1}$.

But we haven't said how to actually define inc. Well, just like this:

$$\mathsf{inc} = \mathsf{It}[\mathsf{Su}]$$

This means that we apply the successor function Su to the value x, for $\downarrow n$ times. If $n \in \mathbb{N}$ then $\downarrow n = n$, so we effectively add n to the value x. If instead n is negative then $\downarrow n = 0$ and nothing changes.

Can you guess how dec is defined?

In a very similar manner, using the predecessor function:

$$\mathsf{dec} = \mathsf{It}[\mathsf{Pr}]$$

and as we will shortly see, finding the inverse is not something that we have to do by hand.

Multiplication and square We now turn our attention to multiplication. The elementary-school way to define multiplication is by repeated addition, and we can define mul exactly like that:

$$\mathsf{mul} = \mathsf{It}[\mathsf{inc}].$$

As inc had arity 2, mul has arity $2+1=3$. If $n, m \in \mathbb{N}$ and $x \in \mathbb{Z}$ then we have

$$\begin{matrix} n \\ m \\ x \end{matrix} \boxed{\mathsf{mul}} \begin{matrix} n \\ m \\ x + n \cdot m \end{matrix}$$

because we're essentially "incrementing by m" n times; so in this case we preserve both inputs and increase a certain variable x.

What is the inverse mul^{-1}? Does it perform division? Well, the truth is rather disappointing:

$$\begin{matrix} n \\ m \\ x \end{matrix} \boxed{\mathsf{mul}^{-1}} \begin{matrix} n \\ m \\ x - n \cdot m \end{matrix}$$

We will see a way to calculate division in RPP, but this is not it.

We're now ready to define the function square which is used to calculate the square of a number:

$$\mathsf{square} = (\mathsf{Id}_1 \| \mathsf{Sw}) \fatsemi \mathsf{inc} \fatsemi \mathsf{mul} \fatsemi \mathsf{dec} \fatsemi (\mathsf{Id}_1 \| \mathsf{Sw}).$$

That might look like a very complicated expression; thankfully we can make use of diagrams to show what each step does. Given $n \in \mathbb{N}$ and $x \in \mathbb{Z}$ we have

$$\begin{matrix} n \\ x \\ 0 \end{matrix} \boxed{\mathsf{Sw}} \begin{matrix} n \\ 0 \\ x \end{matrix} \boxed{\mathsf{inc}} \begin{matrix} n \\ n \\ x \end{matrix} \boxed{\mathsf{mul}} \begin{matrix} n \\ n \\ x + n \cdot n \end{matrix} \boxed{\mathsf{dec}} \begin{matrix} n \\ 0 \\ x + n \cdot n \end{matrix} \boxed{\mathsf{Sw}} \begin{matrix} n \\ x + n \cdot n \\ 0 \end{matrix}$$

so we add the result $n \cdot n$ to a variable x; we also require an additional value initialized to 0. We will make frequent use of variables initially set to 0 and which come back to 0 after the calculation; these are traditionally called **ancillary arguments** or **ancillaes**, from the latin term used to describe female house slaves in ancient Rome.

You might be wondering what would happen if $n < 0$ or the ancilla was different from 0. The truth is, we don't really care. We will often specify the behaviour of these functions given some initial values, and we won't need to know what happens for different initial values because we'll never use those functions in other ways.

1.4 Calculating the inverse

Earlier we hinted at the fact that every function in RPP is invertible and the inverse belongs to RPP; furthermore, we don't need to perform the calculation manually, case by case. In other words, there is an *effective procedure* which produces the inverse $f^{-1} \in \mathsf{RPP}$ given any $f \in \mathsf{RPP}$.

Proposition 1 (The inverse f^{-1} of any f)**.** *Let $f : \mathbb{Z}^n \to \mathbb{Z}^n$ belong to* RPP*. Then the inverse $f^{-1} : \mathbb{Z}^n \to \mathbb{Z}^n$ exists, belongs to* RPP *and, by definition, is*

- $\mathsf{Id}_n^{-1} = \mathsf{Id}_n$
- $\mathsf{Ne}^{-1} = \mathsf{Ne}$
- $\mathsf{Su}^{-1} = \mathsf{Pr}$
- $\mathsf{Pr}^{-1} = \mathsf{Su}$
- $\mathsf{Sw}^{-1} = \mathsf{Sw}$
- $(f \fatsemi g)^{-1} = g^{-1} \fatsemi f^{-1}$
- $(f \| g)^{-1} = f^{-1} \| g^{-1}$
- $\mathsf{It}[f]^{-1} = \mathsf{It}[f^{-1}]$
- $\mathsf{If}[f, g, h]^{-1} = \mathsf{If}[f^{-1}, g^{-1}, h^{-1}]$

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

Proof. By induction on the definition of f. □

Well, that was rather succint.

We invite the reader to check that every listed inverse does indeed make sense; for example, the function $\mathsf{It}[f](x, y_1, \ldots, y_n)$ applies $\downarrow x$ times the function f to the argument $(y_1, \ldots, y_n)$. If we want to "undo" this effect we just need to apply $\downarrow x$ times f^{-1} to the same argument, so $\mathsf{It}[f]^{-1} = \mathsf{It}[f^{-1}]$.

Of course, that reasoning only works if in turn f is also invertible and $f^{-1} \in \mathsf{RPP}$. This is the reason that the proof is by induction: given an arbitrary RPP, if we unfold one step of the definition we get one of the cases listed. We apply the appropriate step and then by inductive hypothesis we can assume that in turn its sub-terms are invertible.

Notice that in this way, if we try to calculate inc^{-1} we really do get dec, as we had hoped for.

Since RPP is inductively defined, any proposition involving RPP functions can be proven using induction. Not only that, but any function which has for an argument a generic RPP can be defined recursively, and indeed we can also see $(\cdot)^{-1} : \mathsf{RPP} \to \mathsf{RPP}$ as a recursive function. Now that we delve into the Lean theorem prover we will see that induction and recursion can be seen as really the same thing, and that's just one of many similarities between functions and proofs.

1.5 First steps with Lean

In Lean we primarily do three things:

1. define data structure
2. define functions
3. prove theorems about data and functions

What sets Lean apart from your average functional programming language (like Haskell) is the third item on the list. Now we will instead focus on the first and second points.

Data is defined using the `inductive` keyword. Here is the typical example of data structure:

```
inductive weekday
| monday
| tuesday
| wednesday
| thursday
| friday
| saturday
| sunday
```

This defines a *type* called `weekday`. Days of the week like `monday`, `tuesday`, etc. are elements of the type `weekday`. We can see this using the `#check` command:

```
#check tuesday -- this outputs "weekday"
```

Everything in Lean has a type. For example, natural numbers have type ℕ:

```
#check 3 -- ℕ
```

Even type themselves have a type[1]. Lean's type system is very expressive, and makes it possible to work with complex math in Lean.

We can define functions over the type `weekday` - for example, the function `next`:

```
-- opening the scope weekday (you can ignore this)
open weekday

-- special characters like → will abound
```

[1]Types in Lean have a role similar to sets in math. Standard math axioms (like ZFC) dictates that everything is a set, including sets themselves. This basic notion can lead to some contradictory statements, like the famous Russell's paradox (let's consider the set of all sets that do not contain themselves; does this set contain itself?) and if one is not careful in defining types of types, the same thing could happen with type theory. But in fact, type theory was invented in the beginning of the 20th century by Bertrand Russell precisely to avoid Russell's paradox. The approach used in Lean is to define a cumulative hierarchy of universes, so that it's impossible to invoke objects like "the type of all types" or a type having itself as an element.

```
-- and are easy to write using a text editor
def next : weekday → weekday
| monday    := tuesday
| tuesday   := wednesday
| wednesday := thursday
| thursday  := friday
| friday    := saturday
| saturday  := sunday
| sunday    := monday

#reduce next wednesday -- this outputs "thursday"
```

(Almost) every expression - like `next (next thursday)` or `3 * 5 + 2` - have a corresponding *reduced form* (respectively `saturday` and `17`) which can be displayed using the `#reduce` command, and is obtained by repeatedly applying functions to their arguments, until the full computation is carried out. In this sense, things like `next wednesday` and `next (next tuesday)` (or `2 + 2` and `1 + 3`) are *definitionally equivalent*, because they're reduced to the same expression.