Certifying expressive power and algorithms of Reversible Primitive Permutations with Lean

Giacomo Maletto[a], Luca Roversi[b]

[a] Università degli Studi di Torino, Dipartimento di Matematica, Italy
[b] Università degli Studi di Torino, Dipartimento di Informatica, Italy

Abstract

Reversible primitive permutations (RPP) is a class of recursive functions that models reversible computation. We present a proof, which has been verified using the proof-assistant Lean, that demonstrates RPP can encode every primitive recursive function (PRF-completeness) and that each RPP can be encoded as a primitive recursive function (PRF-soundness). Our proof of PRF-completeness is simpler and fixes some errors in the original proof, while also introducing a new, more primitive reversible iteration scheme for RPP. By keeping the formalization and semi-automatic proofs simple, we are able to identify a single programming pattern that can generate a set of reversible algorithms within RPP: Cantor pairing, integer division quotient/remainder, and truncated square root. The proof of PRF-soundness is a novel contribution. Finally, Lean source code is available for experiments on reversible computation whose properties can be certified.

Keywords: Reversible computation, Primitive recursion, Lean

1. Introduction

Studies focused on questions posed by Maxwell, regarding the solidity of the principles which Thermodynamics is based on, recognized the fundamental role that Reversible Computation can play to that purpose.

Reversible Computation is a significant area in Computer Science that encompasses various aspects such as reversible hardware design, unconventional computational models (such as quantum or bio-inspired ones), parallel computation and synchronization issues, debugging techniques, and transaction roll-back in database management systems. The book [1] is a comprehensive introduction to the subject; the book [2], focused on the low-level aspects of Reversible Computation, concerning the realization of reversible hardware, and [3], focused on how models of Reversible Computation like Reversible Turing Machines (RTM), and Reversible Cellular Automata (RCA) can be considered universal and how to prove that they enjoy such a property, are complementary to, and integrate [1].

Email addresses: `giacomo.maletto@edu.unito.it` (Giacomo Maletto), `luca.roversi@unito.it` (Luca Roversi)

This work focuses on the *functional model* RPP [4] of Reversible Computation. RPP stands for (the class of) Reversible Primitive Permutations, which can be seen as a possible reversible counterpart of PRF, the class of Primitive Recursive functions [5]. We recall that RPP, in analogy with PRF, is defined as the smallest class built on some given basic reversible functions, closed under suitable composition schemes. The very functional nature of the elements in RPP is at the base of reasonably accessible proofs of the following properties:

- RPP is PRF-complete [4]: for every function $F \in$ PRF with arity $n \in \mathbb{N}$, both $m \in \mathbb{N}$ and f in RPP exist such that f encodes F, i.e. $\texttt{f}(z, \overline{x}, \overline{y}) = (z + F(\overline{x}), \overline{x}, \overline{y})$, for every $\overline{x} \in \mathbb{N}^n$, whenever all the m variables in $\overline{y}$ are set to the value 0. Both z and the tuple $\overline{y}$ are *ancillae.* They can be thought of as temporary storage for intermediate computations of the encoding.

- RPP can be extended to become Turing-complete [6] by means of a minimization scheme analogous to the one that extends PRF to the Turing-complete class of *Partial* Recursive Functions.

- According to [7], RPP and the reversible programming language SRL [8] are equivalent, so the fix-point problem is undecidable for RPP as well [9].

We think that this study provides additional support for the idea that using recursive computational models like RPP to express Reversible Computation allows for the relatively easy certification of the correctness or other properties of RPP algorithms through proof-assistants, potentially leading to the discovery of new algorithms.

We recall that a proof-assistant is an integrated environment to formalize data-types, to implement algorithms on them, to formalize specifications and prove that they hold, increasing algorithms dependability.

Contributions. We show how to express RPP and its evaluation mechanism inside the proof-assistant Lean [10]. We can certify the correctness of every reversible function of RPP with respect to a given specification which means certifying all the main results in [4]. In more detail:

- We give a strong guarantee that RPP is PRF-complete in three macro steps. We exploit that in Lean mathlib library, PRF is proved equivalent to a class of recursive *unary* functions called `primrec`. We define a data-type `rpp` in Lean to represent RPP. Then, we certify that, for any function `f:primrec`, i.e. any unary `f` with type `primrec` in Lean, a function exists with type `rpp` that encodes `f:primrec`. Apart from fixing some bugs, our proof is fully detailed as compared to [4]. Moreover it is conceptually and technically simpler.

- We also give a strong guarantee that RPP is PRF-sound (that is, each RPP is expressible as PRF) thus completing the work in [11], by proving that the two classes of functions have the same expressivity. Again, for the proof of this fact we exploit the definitions and theorems in mathlib concerning primitive recursive functions.

- Concerning simplification, it follows from how the elements in `primrec` work. It is characterized by the following aspects:
  - we define a *new* finite reversible iteration scheme subsuming the reversible iteration schemes in RPP, and SRL, but which is more primitive;
  - we identify an algorithmic pattern which uniquely associates elements of $\mathbb{N}^2$, and $\mathbb{N}$ by counting steps in specific paths. The pattern becomes a reversible element in `rpp` once fixed the parameter it depends on. Slightly different parameter instances generate reversible algorithms whose behavior we can certify in Lean. They are truncated Square Root, Quotient/Reminder of integer division, and Cantor Pairing [12, 13]. The original proof in [4] that RPP is PRF-complete relies on Cantor Pairing, used as a stack to keep the representation of a PRF function as element of RPP reversible. Our proof in Lean replaces Cantor Pairing with a reversible representation of functions `mkpair` /`unpair` that mathlib supplies as isomorphism $\mathbb{N} \times \mathbb{N} \simeq \mathbb{N}$. The truncated Square Root is the basic ingredient to obtain reversible `mkpair`/`unpair`.

Related work. Concerning the formalization in a proof-assistant of the semantics, and its properties, of a formalism for Reversible Computation, we are aware of [14]. By means of the proof-assistant Matita [15], it certifies that a denotational semantics for the imperative reversible programming language Janus [1, Section 8.3.3] is fully abstract with respect to the operational semantics.

Concerning *functional models* of Reversible Computation, we are aware of [16] which introduces the class of reversible functions RI, which is as expressive as the *Partial* Recursive Functions. So, RI is stronger than RPP; however we see RI as less abstract than RPP for two reasons: (i) the primitive functions of RI depend on a given specific binary representation of natural numbers; (ii) unlike RPP, which we can see as PRF in a reversible setting, it is not evident to us that RI can be considered the natural extension of a total class analogous to RPP.

Finally, this work, starting from relevant parts of the BSc Thesis [17], which comes with a Lean project [18] that certifies both properties and algorithms of RPP, strictly extends [11] with the proof that RPP is PRF-sound.

Contents. Section 2 recalls the class RPP by commenting on the main design aspects that characterize its definition inside Lean. Section 3 defines and proves correct new reversible algorithms central to the proof. Section 4 recalls the main aspects of `primrec`, and illustrates the key steps to port the original PRF-completeness proof of RPP to Lean. Section 5 shows how we used the constructs present in the mathlib library to prove the PRF-soundness of RPP. Section 6 is about possible developments.

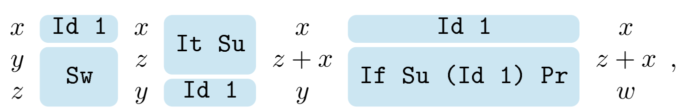
2. Reversible Primitive Permutations (RPP)

```
inductive rpp : Type
-- Base functions
| Id (n : ℕ) : rpp -- Identity
| Ne : rpp        -- Sign-change
| Su : rpp        -- Successor
| Pr : rpp        -- Predecessor
| Sw : rpp        -- Transposition or Swap
-- Inductively defined functions
| Co (f g : rpp) : rpp   -- Series composition
| Pa (f g : rpp) : rpp   -- Parallel composition
| It (f : rpp) : rpp     -- Finite iteration
| If (f g h : rpp) : rpp -- Selection
infix `‖` : 55 := Pa -- Notation for the Parallel composition
infix `;;` : 50 := Co -- Notation for the Series composition
```

Figure 1: The class RPP as a data-type `rpp` in Lean.

We use the data-type `rpp` in Figure 1, as defined in Lean, to recall from [4] that the class RPP is the smallest class of functions that contains five base functions, named as in in Figure 1, and all the functions that we can generate by the composition schemes whose name is next to the corresponding clause in Figure 1. For ease of use and readability the last two lines in Figure 1 introduce infix notations for series and parallel composition.

Example 1 (A term of type `rpp`). In `rpp` we can write `(Id 1‖Sw);;(It Su)‖(Id 1);;(Id 1‖If Su (Id 1) Pr)` which we also represent as a diagram:



where:

$$w = \begin{cases} y+1 & \text{if } z+x>0 \\ y & \text{if } z+x=0 \\ y-1 & \text{if } z+x<0 \end{cases} .$$

The inputs are the names to the left-hand side of the blocks; the outputs are to their right-hand side. The term here above is a series composition of three parallel compositions. The first one composes a unary identity `Id 1`, which leaves its unique input untouched, and `Sw`, which swaps its two arguments. Then, the x-times iteration of the successor `Su`, i.e. `It Su`, is in parallel with `Id 1`: that is why one of the outputs of `It Su` is $z+x$. Finally, `If Su (Id 1) Pr` selects which among `Su`, `Id 1`, and `Pr` to apply to the argument y, depending on the value of $z+x$; in particular, `Pr` is the function that computes the predecessor of the argument. Figure 5 will give the operational semantics which defines `rpp` formally as a class of functions on $\mathbb{Z}$, not on $\mathbb{N}$. □

```
def arity : rpp → ℕ
  | (Id n)      := n
  | Ne          := 1
  | Su          := 1
  | Pr          := 1
  | Sw          := 2
  | (f ‖ g)     := f.arity + g.arity
  | (f ;; g)    := max f.arity g.arity
  | (It f)      := 1 + f.arity -- It f has an extra argument compared to f
  | (If f g h) := 1 + max (max f.arity g.arity) h.arity
```

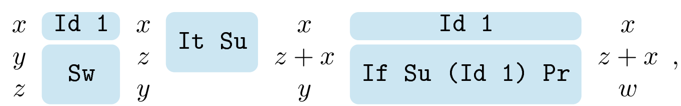
Figure 2: Arity of every `f: rpp`.

(a) `n` unary identities of RPP in parallel.

(b) Single `n`-ary identity `rpp`.

Figure 3: `n`-ary identities are base functions of `rpp`.

Remark 1 ("Weak weakening" of algorithms in `rpp`). We typically drop `Id m` if it is the last function of a parallel composition. For example, term and diagram in *Example* 1 become `(Id 1‖Sw);;(It Su);;(Id 1‖If Su (Id 1) Pr)` and:



where:

$$w = \begin{cases} y+1 & \text{if } z+x>0 \\ y & \text{if } z+x=0 \\ y-1 & \text{if } z+x<0 \end{cases} .$$

Remark 2 explains why. □

The function in Figure 2 computes the arity of any `f:rpp` from the structure of `f`, once fixed the arities of the base functions; `f.arity` is Lean dialect for the more typical notation "`arity(f)`".

Figure 3 remarks that `rpp` considers `n`-ary identities `Id n` as primitive; in RPP the function `Id n` is obtained by parallel composition of `n` unary identities.

For any given `f:rpp`, the function `inv` in Figure 4 builds an element with type `rpp`. The definition of `inv` lets the successor `Su` be inverse of the predecessor `Pr` and lets every other base function be self-dual. Moreover, the function `inv` distributes over finite iteration `It`, selection `If`, and parallel composition ‖, while it requires to exchange the order of the arguments before distributing over the series composition `;;`. The last line with `notation` suggests that $\texttt{f}^{-1}$

```
def inv : rpp → rpp
  | (Id n)      := Id n -- self-dual
  | Ne          := Ne   -- self-dual
  | Su          := Pr
  | Pr          := Su
  | Sw          := Sw   -- self-dual
  | (f ‖ g)     := inv f ‖ inv g
  | (f ;; g)    := inv g ;; inv f
  | (It f)      := It (inv f)
  | (If f g h)  := If (inv f) (inv g) (inv h)
notation f ‘⁻¹‘ := inv f
```

Figure 4: Inverse `inv f` of every `f:rpp`.

```
def ev : rpp → list ℤ → list ℤ
| (Id n)     X                 := X
| Ne         (x :: X)          := −x :: X
| Su         (x :: X)          := (x + 1) :: X
| Pr         (x :: X)          := (x − 1) :: X
| Sw         (x :: y :: X)     := y :: x :: X
| (f ;; g)   X                 := ev g (ev f X)
| (f ‖ g)    X                 := ev f (take f.arity X) ++ ev g (drop f.arity X)
| (It f)     (x :: X)          := x :: ((ev f)^[↓x] X)
| (If f g h) (0 :: X)          := 0 :: ev g X
| (If f g h) (((n : ℕ) + 1) :: X) := (n + 1) :: ev f X
| (If f g h) (−[1 + n] :: X)   := −[1 + n] :: ev h X
| _          X                 := X
notation ‘‹‘ f ‘›‘ := ev f
```

Figure 5: Operational semantics of elements in `rpp`.

is the inverse of `f`; we shall prove this fact once given the operational semantics of `rpp`.

2.1. Operational semantics of `rpp`

The function `ev` in Figure 5 interprets an element of `rpp` as a function from a list of integers to a list of integers. Originally, in [4], RPP is a class of functions with type $\mathbb{Z}^n \to \mathbb{Z}^n$. We use `list ℤ` in place of tuples of $\mathbb{Z}$ to exploit Lean library mathlib and save a large amount of formalization.

Let us give a look at the clauses in Figure 5.

The function `Id n` leaves the input list `X` untouched. `Ne` "negates", i.e. takes the opposite sign of, the head of the list, while `Su` increments, and `Pr` decrements it. `Sw` is the transposition, or swap, that exchanges the first two elements of its argument. The series composition `f;;g` first applies `f` and then `g`. The parallel composition `f‖g` splits `X` into two parts. The "topmost" one `(take f.arity X)` has as many elements as the arity of `f`; the "lowermost" one `(drop f.arity X)` contains the part of `X` that can supply the arguments to `g`. Finally, it concatenates the two resulting lists by the append `++`.

Finite iteration `It f` is new:

- it iterates `f` as many times as the value of the head `x` of the argument, if `x` contains a non negative value;
- otherwise it is the identity on the whole `x::X`.

We denote this behavior by means of `(ev f)^[↓x]`.

The selection `If f g h` chooses one among `f`, `g`, and `h`, depending on the argument head `x`: it is `g` with `x = 0`, it is `f` with `x > 0`, and `h` with `x < 0`. The last line of Figure 5 sets a handy notation for `ev`.

Remark 2 (We want to keep the definition of `ev` simple). Based on our definition, using Lean, we show that:

```
theorem ev_split (f: rpp) (X: list ℤ):
<f> X = (<f> (take f.arity X)) ++ drop f.arity X
```

holds. It is one of the most complex property to prove because it essentially says that we can apply any `<f>` to *any* `X` with at least as many elements as `arity f`.

The proof is based on two observations.

First, if `X.length >= f.arity`, i.e. `X` supplies enough arguments, then `f` operates on the first elements of `X` according to its arity. This justifies *Remark* 1.

Second, if `X.length < f.arity` holds, i.e. `X` has not enough elements, then `f X` has an unspecified behavior; this might sound odd, but it simplifies the certified proofs of must-have properties of `rpp`. □

2.2. The functions `inv h` *and* `h` *are each other inverse*

Once defined `inv` in Figure 4 and `ev` in Figure 5 we can prove:

```
theorem inv_co_l (h : rpp) (X : list ℤ) : <h ;; h^-1> X = X
theorem inv_co_r (h : rpp) (X : list ℤ) : <h^-1 ;; h> X = X
```

certifying that `h` and h^{-1} are each other inverse. We start by focusing on the main details to prove `theorem inv_co_l` in Lean. The proof proceeds by (structural) induction on `h`, which generates 9 cases, one for each clause that defines `rpp`. One can go through the majority of them smoothly. Some comments about two of the more challenging cases follow.

Parallel composition. Let `h` be some parallel composition, whose main constructor is `Pa`. The step-wise proof of `inv_co_l` is:

```
<f‖g;;(f‖g)^{-1}> X
    = <f‖g;;f^{-1}‖g^{-1}> X        -- by definition
(!) = <(f;;f^{-1})‖(g;;g^{-1})> X  -- lemma pa_co_pa, arity_inv below
    = <f;;f^{-1}>(take f.arity X) ++ <g;;g^{-1}>(drop f.arity X)
                                 -- by definition
    = take f.arity X ++ drop f.arity X -- by ind. hyp.
    = X                          -- property of ++ (append),
```

where the equivalence `(!)` holds because we can prove both:

```
lemma pa_co_pa (f f' g g' : rpp) (X : list ℤ) :
 f.arity = f'.arity → <f‖g ;; f'‖g'> X = <(f;;f') ‖ (g;;g')> X ,
lemma arity_inv (f : rpp) : f^{-1}.arity = f.arity .
```

Proving `lemma arity_inv`, i.e. that the arity of a function does not change if we invert it, assures that we can prove `lemma pa_co_pa`, i.e. that series and parallel compositions smoothly distribute reciprocally.

Iteration. Let `h` be a finite iterator whose main constructor is `It`. The goal to prove is `<It f;;It f^{-1}> x::X = x::X` which reduces to `<f^{-1}>^[↓x] (<f>^[↓x] X') = X'`, where, we recall, the notation `<f>^[↓x]` means "`<f>` applied `x` times, if `x` is positive". Luckily this last statement is both formalized as `function.left_inverse g^[n] f^[n]`, available in the library mathlib of Lean.

To conclude, let us see how the proof of `inv_co_r` works. It does not copy-cat the one of `inv_co_l`. It relies on proving:

```
lemma inv_involute (f : rpp) : (f^{-1})^{-1} = f ,
```

which says that applying `inv` twice is the identity, and on using `inv_co_l`:

```
<f^{-1} ;; f> X = X -- which, by inv_involute, is equivalent to
<f^{-1} ;; (f^{-1})^{-1}> X = X -- which holds because it is an
instance of (inv_co_l f^{-1}) .
```

A less general, but semantically more appropriate version of `inv_co_l` and `inv_co_r` could be:

```
theorem inv_co_l (f : rpp) (X : list ℤ) :
            f.arity ≤ X.length → <f ;; f^{-1}> X = X
theorem inv_co_r (f : rpp) (X : list ℤ) :
            f.arity ≤ X.length → <f^{-1} ;; f> X = X
```

because, recalling *Remark* 2, the application `(f X)` makes sense when `f.arity ≤ X.length`. Fortunately, the way we defined `rpp` allows us to state `inv_co_l` or `inv_co_r` in full generality with no reference to `f.arity ≤ X.length`.

2.3. Changes from the original definition

The definition of `rpp` in Lean is really very close to the original RPP, but not identical. The goal is to simplify the overall task of formalization and certification. The brief list of changes follows.

- As already outlined, `It` and `If` use the head of the input list to iterate or choose: taking the head of a list with pattern matching is obvious. In [4], it is the last element in the input tuple that drives iteration and selection of RPP.

- `Id n`, for any `n`:$\mathbb{N}$, is primitive in `rpp` and derived in RPP.

- Using `list` $\mathbb{Z} \to$ `list` $\mathbb{Z}$ as the domain of the function that interprets any given element `f:rpp` avoids letting the type of `f:rpp` depend on the arity of `f`. To know the arity of `f` it is enough to invoke `arity f`. Finally, we observe that getting rid of a dependent type like, say, `rpp n`, allows us to escape situations in which we would need to compare equal but not definitionally equal types like `rpp (n+1)` and `rpp (1+n)`.

- The new finite iterator `It f (x::t): list` $\mathbb{Z}$ *subsumes* the finite iterators `ItR` in RPP, and `for` in SRL. This means that `It` is more primitive, but equally expressive and simpler for Lean to prove that its definition is terminating.

  More specifically, we recall that:

  - `ItR f` $(x_0, x_1, \ldots, x_{n-2}, x)$ simply evaluates to `f(f(...f(`$x_0, x_1, \ldots, x_{n-2}$`)...))` with $|x|$ occurrences of `f`;
  - `for(f) x` is slightly more complex:
    1. it evaluates to `f(f(...f(`$x_0, x_1, \ldots, x_{n-2}$`)...))`, with `x` occurrences of `f`, if `x > 0`;
    2. it evaluates to $f^{-1}(f^{-1}(\ldots f^{-1}(x_0, x_1, \ldots, x_{n-2})\ldots))$, with `-x` occurrences of f^{-1}, if `x < 0`;
    3. it behaves like the identity if `x = 0`.

  We know how to define both `ItR` and `for` in terms of `It`:

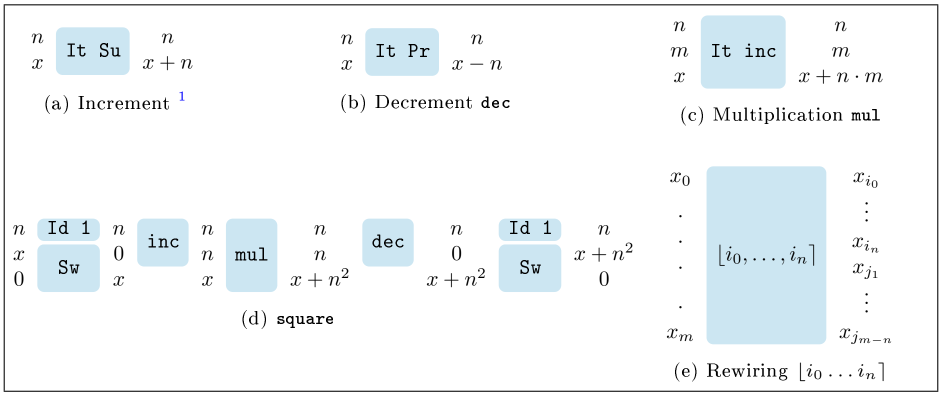
$$\texttt{ItR f} = \texttt{(It f);;Ne;;(It f);;Ne} \tag{1}$$

$$\texttt{for(f)} = \texttt{(It f);;Ne;;(It f}^{-1}\texttt{);;Ne} \; . \tag{2}$$

Example 2 (How does (1) work?). Whenever `x > 0`, the leftmost `It f` in (1) iterates `f`, while the rightmost one does nothing because `Ne` in the middle negates `x`. On the contrary, if `x < 0`, the leftmost `It f` does nothing and the iteration is performed by the rightmost iteration, because `Ne` in the middle negates `x`. In both cases, the last `Ne` restores `x` to its initial sign. But this is the behavior of `ItR`, as we wanted. □

3. RPP algorithms central to our proofs

[1] Note that using our definition, the variable `n` must be non-negative in order to have the shown behavior, otherwise the function acts as the identity. This is why it's called *increment* and not *addition*.



(a) Increment [1] (b) Decrement `dec` (c) Multiplication `mul` (d) `square` (e) Rewiring $\lfloor i_0 \ldots i_n \rceil$

Figure 6: Some useful functions of `rpp`

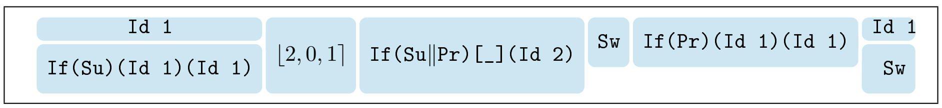


Figure 7: Algorithm scheme `step[_]`. The algorithm we can obtain from it depends on how we fill the hole `[_]`.

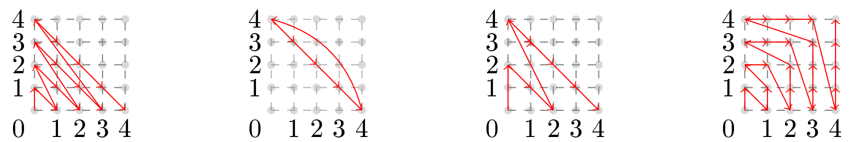
Figure 6 recalls definition, and behavior of some `rpp` functions already introduced in [4].

It is worth commenting on how the function *rewiring* $\lfloor i_0 \ldots i_n \rceil$ works. Let $\{i_0, \ldots, i_n\} \subseteq \{0, \ldots, m\}$ be a set of $n+1$ distinct indices between 0 and m, and $\{j_1, \ldots, j_{m-n}\} = \{0, \ldots, m\} \setminus \{i_0, \ldots, i_n\}$ which we assume such that $j_k < j_{k+1}$, for every $1 \leq k < m-n$. By definition, $\lfloor i_0, \ldots, i_n \rceil (x_0, \ldots, x_m) = (x_{i_0}, \ldots, x_{i_n}, x_{j_1}, \ldots, x_{j_{m-n}})$, i.e. rewiring brings every input with index in $\{i_0, \ldots, i_n\}$ in front of all the inputs with index in $\{j_1, \ldots, j_{m-n}\}$, preserving the order.

3.1. The algorithm scheme `step[_]`

Figure 7 identifies the *new algorithm scheme* `step[_]`. Depending on how we fill the hole `[_]`, we get step functions that, once iterated, draw paths in $\mathbb{N}^2$.

On top of the functions in Figures 6, and 7 we build Cantor Pairing/Unpairing, Quotient/Reminder of integer division, and truncated Square Root. Suitable instances of `step[_]` allow us to visit $\mathbb{N}^2$ as in Figures 8a, 8b, and 8c, respectively. The pairing function `mkpair`, which behaves as in Figure 8d, and which is an alternative to Cantor Pairing/Unpairing, has a more complex definition; it will be a necessary ingredient of our main proof.



(a) Cantor (b) Quot./Rem. (c) Square root (d) `mkpair`

Figure 8: Paths in $\mathbb{N}^2$ that generate algorithms in `rpp`.

Cantor (Un-)Pairing. The standard definition of Cantor Pairing $\mathsf{cp} : \mathbb{N}^2 \to \mathbb{N}$ and Un-pairing $\mathsf{cu} : \mathbb{N} \to \mathbb{N}^2$, two bijections one inverse of the other, is:

$$\mathsf{cp}(x, y) = \sum_{i=1}^{x+y} i + x = \frac{(x+y)(x+y+1)}{2} + x \tag{3}$$

$$\mathsf{cu}(n) = \left(n - \frac{i(1+i)}{2}, \frac{i(3+i)}{2} - n\right) \ , \tag{4}$$

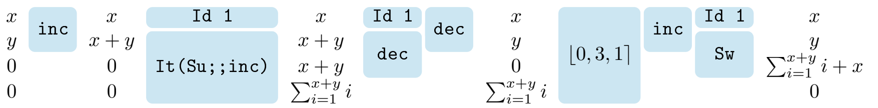
where $i = \left\lfloor \frac{\sqrt{8n+1}-1}{2} \right\rfloor$.

Figure 9 has all we need to define Cantor Pairing `cp:rpp`, and Un-pairing `cu:rpp`. In Figure 9a, `cp_in` is the natural algorithm in `rpp` to implement (3). As expected, the input pair (x, y) is part of `cp_in` output, a fact that the suffix "`_in`" recalls in the name of the function. In order to drop (x, y) from the output of `cp_in`, and to obtain `cp` as in Figure 9e, we apply *Bennett's trick* using `cu_in` $^{-1}$, i.e. the inverse of `cu_in`, whose definition is completely new, as compared to the corresponding one defined previously in [4]. The intuition behind `cu_in` is as follows. Let us fix any point $(x, y) \in \mathbb{N}^2$. We can realize that, starting from the origin, if we follow as many steps as the value $\mathsf{cp}(x, y)$ in Figure 8a, we stop exactly at (x, y). A standard functional notation for the function that, given the current point (x, y), identifies the next one to move to in the path of Figure 8a is:

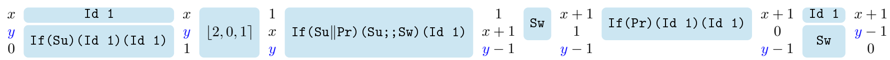
$$\mathrm{step}(x, y) = \begin{cases} (x+1, y-1) & y > 0 \\ (0, x+1) & y = 0 \end{cases} .$$

We implement $\mathrm{step}(x, y)$ in `rpp` as `step[Su;;Sw]`. Figures 9b, and 9c represent two runs of `step[Su;;Sw]` to give visual evidence that `step[Su;;Sw]` implements $\mathrm{step}(x, y)$. Colored occurrences of y show the relevant part of the computational flow. Note that we cannot implement $\mathrm{step}(x, y)$ by using the conditional directly on y, because in the computation we also want to modify the value of y. Finally, as soon as we get `cu_in` by iterating `step[Su;;Sw]` as in Figure 9d, we can define `cp` (Figure 9e), and `cu` (Figure 9f).

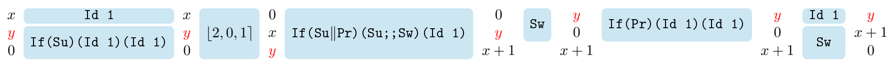
Quotient and reminder. Let us focus on the path in Figure 8b. It starts at $(0, n)$ (with $n = 4$), and, at every step, the next point is in *direction* $(+1, -1)$. When it reaches $(n, 0)$ (with $n = 4$), instead of jumping to $(0, n+1)$, as in Figure 8a,



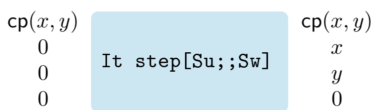
(a) Function `cp_in`



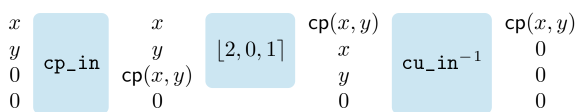
(b) Function `step[Su;;Sw]`: detailed behavior with $y > 0$.



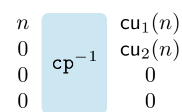
(c) Function `step[Su;;Sw]`: detailed behavior with $y = 0$.



(d) Function `cu_in`

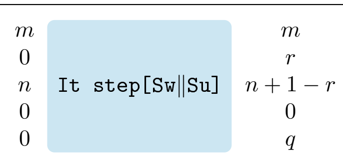


(e) The function `cp`

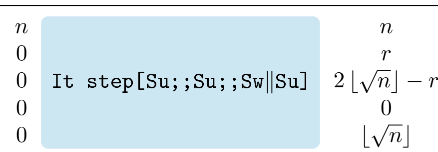


(f) The function `cu`

Figure 9: Cantor Pairing and Un-pairing.



(a) Function that computes q and r such that $m = q(n+1) + r$. We obtain it by iterating `step[Sw‖Su]`.



(b) Function that computes $\lfloor\sqrt{n}\rfloor$ and $r = n - \lfloor\sqrt{n}\rfloor^2$. We obtain it by iterating `step[Su;;Su;;Sw‖Su]`.

Figure 10: Quotient/Reminder and Square root.

it lands again on $(0, n)$. The idea is to keep looping on the same diagonal. This behavior can be achieved by iterating `step[Sw‖Su]`. Figure 10a shows that we are doing modular arithmetic. Globally, it takes $n+1$ steps from $(0, n)$ to itself by means of `step[Sw‖Su]`. Specifically, if we assume we have performed m steps along the diagonal, and we are at point (x, y), we have that $x \equiv m \pmod{n+1}$ and $0 \leq x \leq n$. So, if we increase a counter by one each time we reset our position to $(0, n)$ we can calculate quotient and reminder.

Truncated Square root. Let us focus on the path in Figure 8c. It starts at $(0, 0)$. Whenever it reaches $(x, 0)$ it jumps to $(0, x+2)$, otherwise the next point is in *direction* $(+1, -1)$. The behavior can be achieved by iterating `step[Su;;Su;;Sw‖Su]` as in Figure 10b. In order to compute $\lfloor \sqrt{n} \rfloor$, besides implementing the above path, the function `step[Su;;Su;;Sw‖Su]` counts in k the number of jumps occurred so far along the path. In particular, starting from $(0, 0)$, the first jump occurs in the first step; the next one in the $(1+3)$th, then the $(1+3+5)$th, then the $(1+3+5+7)$th etc. Since we know that $1+3+\cdots+(2k-1) = k^2$ for any k, letting n be the number of iterations (and hence the numbers of steps) we have that k is such that $k^2 \leq n < (k+1)^2$; i.e. $k = \lfloor \sqrt{n} \rfloor$.

Remark 4. The value $2 \lfloor \sqrt{n} \rfloor - r$ can be canceled out by adding r, and subtracting $\lfloor \sqrt{n} \rfloor$ twice. What we *cannot* eliminate is the "remainder" $r = n - \lfloor \sqrt{n} \rfloor^2$ because the *function* Square root cannot be inverted in $\mathbb{Z}$, and the algorithm cannot forget it.

The mkpair *function.* Figure 8d shows the behavior of the function mkpair. It is very similar to the one of cp, but it uses an alternative algorithm described in [19]. Here we do not describe it in detail because it's just a composition of sums, products and square roots, just discussed here above.

3.2. A note on the mechanization of proofs

We recall once more that everything defined here above has been proved correct in Lean (see [18] for the details). For example, once defined `sqrt` in Lean, the following lemma:

```
lemma sqrt_def (n : ℕ) (X : list ℤ) :
  <sqrt>(n::0::0::0::0::X) =
            n::(n-√n*√n)::(√n+√n-(n-√n*√n))::0::√n::X
```

shows that `sqrt` behaves as expected, for any `n`.

In order to prove the here above lemma, or similar ones, we make use of the *tactic* `simp`, i.e. a Lean command that builds proofs. The tactic `simp` can automatically simplify expressions until trivial identities show up. What is meant by "simplify" is that theorems which state an equality with form `Left_hand_side = Right_hand_side`, like in `sqrt_def`, can be marked with the attribute `@[simp]`; the very useful consequence is that every time `simp` is invoked in a subsequent proof, if the equality to be proved contains an instance of `Left_hand_side`, then it will be substituted with `Right_hand_side`, often making it simpler to conclude a proof.

```
inductive primrec:(ℕ → ℕ) → Prop
| zero: primrec (λ (n:ℕ), 0)
| succ: primrec succ
| left: primrec (λ (n:ℕ), (unpair n).fst)
| right: primrec (λ (n:ℕ), (unpair n).snd)
| pair {F G}: primrec F → primrec G → primrec (λ (n:ℕ), mkpair (F n) (G n))
| comp {F G}: primrec F → primrec G → primrec (λ (n:ℕ), F (G n))
| prec {F G}: primrec F → primrec G → primrec
(unpaired (λ (z n:ℕ), nat.rec (F z) (λ (y IH:ℕ), G (mkpair z (mkpair y IH))) n))
```

Figure 11: `primrec` defines PRF in mathlib of Lean.

So, `@[simp]` introduces an incremental and quite handy mechanism to automate proofs: the more available proofs exist, the more we can, in principle, label as `@[simp]`, widening the possibility to automatically prove further properties.

4. Proving in Lean that RPP is PRF-complete

We formally show in Lean that the class of functions we can express as (algorithms) in `rpp` contains at least the class PRF of Primitive Recursive Functions; we say that "`rpp` is PRF-complete". The definition of PRF that we take as reference is one of the two available in Lean mathlib library. Once recalled and commented it briefly, we shall proceed with the main aspects of the PRF-completeness of `rpp`.

4.1. Primitive Recursive Functions `primrec` in mathlib

Figure 11 recalls the definition of PRF from [20] available in mathlib that we take as reference. It is an inductively defined Proposition `primrec` that requires a *unary* function with type $\mathbb{N} \rightarrow \mathbb{N}$ as argument. Specifically, `primrec` is the least collection of functions $\mathbb{N} \rightarrow \mathbb{N}$ with a given set of base elements, closed under some composition schemes.

Base functions of `primrec`. The *constant* function `zero` yields 0 on every of its inputs. The `succ`*essor* gives the natural number next to the one taken as input. The two *projections* `left`, and `right` take an argument `n`, and extract a left, or a right, component from it as `n` was the result of pairing two values `x,y:`$\mathbb{N}$. The functions that `primrec` relies on to encode/decode pairs on natural numbers as a single natural one are `mkpair:`$\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$, and `unpair:`$\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$. The first one builds the value `mkpair x y`, i.e. the number of steps from the origin to reach the point with coordinates `(x,y)` in the path of Figure 8d. The function `unpair:`$\mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N}$ takes the number of steps to perform on the same path. Once it stops, the coordinates of that point are the two natural numbers we are looking for. So, `mkpair`/`unpair` are an alternative to Cantor Pairing/Un-pairing.

Composition schemes. Three schemes exist in `primrec`, each depending on parameters `f,g:primrec`. The scheme `pair` builds the function that, taken a value `n:ℕ`, gives the unique value in ℕ that encodes the pair of values `F n`, and `G n`; everything we might pack up by means of `pair`, we can unpack with `left`, and `right`.

The scheme `comp` composes `F,G:primrec`.

The *primitive recursion* scheme `prec` can be "unfolded" to understand how it works. This reading will ease the description of how to encode it in `rpp`. Let `F`, `G` be two elements of `primrec`. We see `prec` as encoding the function:

$$H[\mathtt{F},\mathtt{G}](x) = R[\mathtt{G}]\big(\mathtt{F}\big((x)_1\big),(x)_2\big) \tag{5}$$

where: (i) $(x)_1$ denotes `(unpair x).fst`, (ii) $(x)_2$ denotes `(unpair x).snd`, and (iii) $R[\mathtt{G}]$ behaves as follows:

$$\begin{aligned} R[\mathtt{G}](z,0) &= z \\ R[\mathtt{G}](z,n+1) &= \mathtt{G}\big(«z,«n,R[\mathtt{G}](z,n)»»\big)\ , \end{aligned} \tag{6}$$

defined using the built-in recursive scheme `nat.rec` on ℕ, and $«a,b»$ denotes (`mkpair` a b).

4.2. The main point of the proof

In order to formally state what we mean for `rpp` to be PRF-complete, in Lean we need to say when, given `F:ℕ → ℕ`, we can *encode* it by means of some `f:rpp`. This is done by means of the following definition:

```
def encode (F:ℕ → ℕ) (f:rpp) :=
    ∀ (z:ℤ) (n:ℕ), <f> (z::n::repeat 0 (f.arity-2))
                        = (z+(F n))::n::repeat 0 (f.arity-2)
```

which says that, fixed `F:ℕ → ℕ`, and `f:rpp`, the statement `(encode F f)` holds if the evaluation of `<f>`, applied to any argument `(z::n::0::...::0)` with as many occurrences of trailing 0s as `f.arity-2`, gives a list with form `((z+(F n))::n::0::...::0)` such that:

(i) the first element is the original value `z` increased with the result `(F n)` of the function we want to encode;

(ii) the second element is the initial `n`;

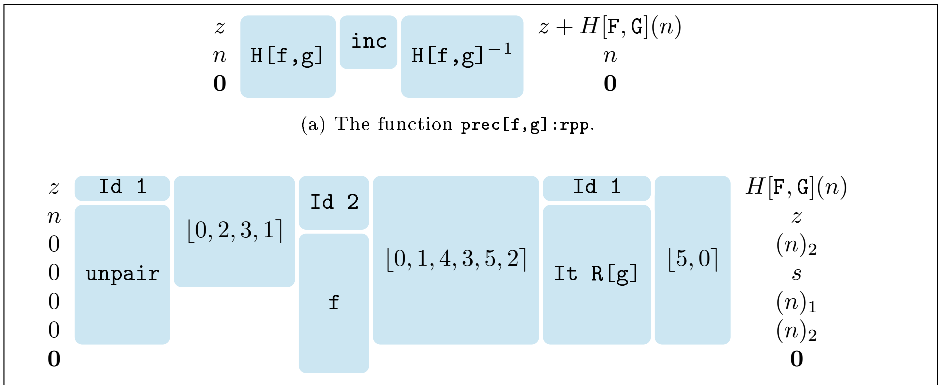
(iii) trailing 0s are again as many as `f.arity-2`.

In Lean we can prove:

```
theorem completeness (F:ℕ → ℕ):
            primrec F → ∃ f:rpp, encode F f
```

which says that we know how to build `f:rpp` which `encode`s `F`, for every well formed `F:ℕ → ℕ`, i.e. such that `primrec F` holds.

The proof proceeds by induction on the proposition `primrec`, which generates 7 sub-goals. We illustrate the main arguments to conclude the most interesting case which requires to encode the composition scheme `prec`.



(a) The function `prec[f,g]:rpp`.

(b) The function `H[f,g]` with parameters `f`, `g`.

Figure 12: Encoding `prec` of Figure 11 in `rpp`.

Remark 5. Many aspects of the proof that we here detail out, "forced" by Lean, so to say, were simply missing in the original PRF-completeness proof for RPP in [4]. □

The inductive hypothesis to show that we can encode `prec` is that, for any given `F,G:`$\mathbb{N} \to \mathbb{N}$ such that `(primrec F):Prop`, and `(primrec G):Prop`, both `f,g:rpp` exist such that `(encode F f)`, and `(encode G g)` hold. This means that both:

$$\begin{aligned} \texttt{f}\ (z::n::\mathbf{0}) &= (z + \texttt{F}\ n)::n::\mathbf{0} \\ \texttt{g}\ z::n::\mathbf{0} &= (z + \texttt{G}\ n)::n::\mathbf{0} \end{aligned}$$

hold, where $\mathbf{0}$ stands for a sufficiently long list of 0s. Moreover, Figure 12a, in which the assumption is that $z = 0$, defines `prec[f,g]:rpp` such that:

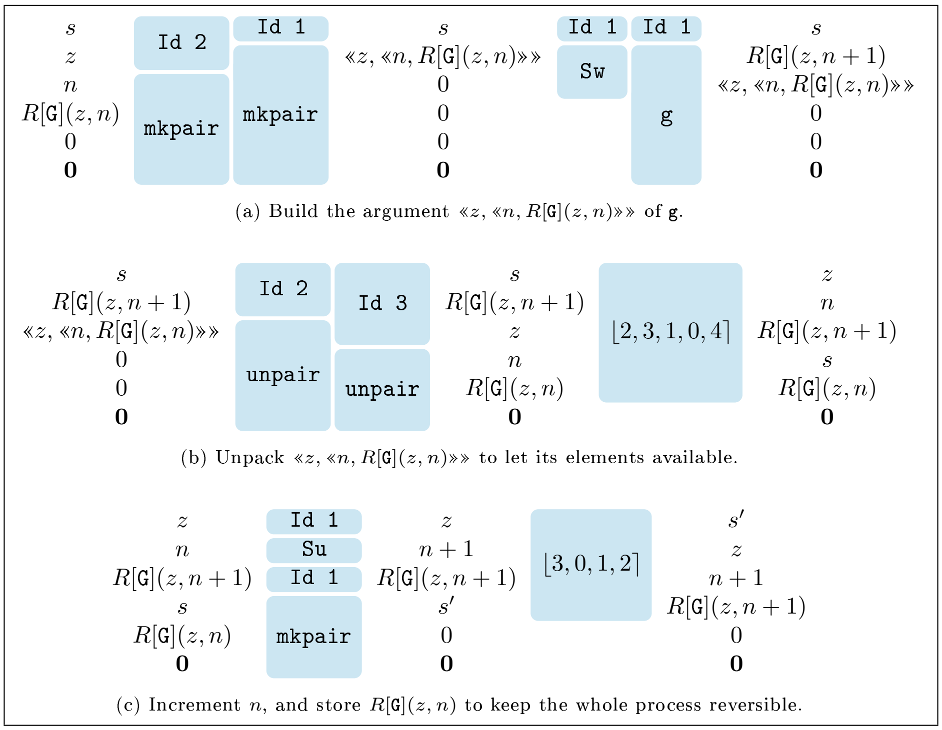
(i) `(encode (prec F G) prec[f,g]):Prop` holds, and

(ii) `H[f,g]` encodes $H[\texttt{F},\texttt{G}]$

as in (5). Finally, the term `It R[g]` in `H[f,g]` encodes (6) by iterating `R[g]` from the initial value given by `f`.

Figure 13 splits the definition of `R[g]` into three logical parts. Figure 13a packs everything up by means of `mkpair` to build the argument $R[\texttt{G}](z, n)$ of `g`; by induction we get $R[\texttt{G}](z, n + 1)$. In Figure 13b, `unpair` unpacks «z, «$n, R[\texttt{G}](z, n)$»» to expose its components to the last part. Figure 13c both increments n, and packs $R[\texttt{G}](z, n)$ into s, by means of `mkpair`, because $R[\texttt{G}](z, n)$ has become useless once obtained $R[\texttt{G}](z, n+1)$ from it. Packing $R[\texttt{G}](z, n)$ into s, so that we can eventually recover it, *is mandatory*. We cannot "replace" $R[\texttt{G}](z, n)$ with 0 because that would not be a reversible action.

Remark 6. The function `cp` in Figure 9e can replace `mkpair` in Figure 13c as a bijective map $\mathbb{N}^2$ into $\mathbb{N}$. Indeed, the original PRF-completeness of RPP relies on `cp`. We favor `mkpair` to take the most out of mathlib. □



(a) Build the argument $«z, «n, R[\mathtt{G}](z,n)»»$ of `g`.

(b) Unpack $«z, «n, R[\mathtt{G}](z,n)»»$ to let its elements available.

(c) Increment n, and store $R[\mathtt{G}](z,n)$ to keep the whole process reversible.

Figure 13: Encoding $R[\mathtt{G}]$ in (6) as `R[g]:rpp`.

5. Proving in Lean that RPP is PRF-sound

We formally show in Lean that every function we can express as (algorithm) in rpp can be expressed as an element of PRF, the class of Primitive Recursive Functions; we say that "rpp is PRF-sound". This means that, through a suitable embedding of list $\mathbb{Z}$ in $\mathbb{N}$ and thus seeing each <f>:list $\mathbb{Z} \to$ list $\mathbb{Z}$ as a function of type $\mathbb{N} \to \mathbb{N}$, this is always primitive recursive. In Lean terms, we can prove:

```
theorem rpp_primrec (f:rpp) : primrec <f>
```

As far as we know, no full proof of this fact was present before, [6] included. In order to show it, we make heavy use of previously established theorems present in Lean mathlib library.

5.1. *The extended definition of* primrec *in* mathlib

Section 4 recalls the meaning for a function of type $\mathbb{N} \to \mathbb{N}$ to be primrec. We are now interested in expressing a function f:$\alpha \to \beta$, i.e. with some given domain of type α, and co-domain of type β, as a primitive recursive function. If we somehow "link" both α, and β to $\mathbb{N}$, we can leverage our previous definitions and results.

Three main steps do the job:

1. First, we require that both α, and β be encodable, notion defined in Lean by means of:

   ```
   class encodable (α : Type*) :=
   (encode : α → ℕ)
   (decode [] : ℕ → option α)
   (encodek : ∀ a, decode (encode a) = some a)
   ```

   It means that *computable immersions* encode exist with type $\alpha \to \mathbb{N}$ (and $\beta \to \mathbb{N}$). The inverse function decode need only be defined for those n : $\mathbb{N}$ which are in the image of the immersion: for this reason, decode has return type option α, a type in which all elements are of the form none or some a for a : α; the elements of n : $\mathbb{N}$ not in the image can just be mapped to none.

2. Second, it is important to work always with the same instance of *computable immersion* as we proceed with the development of the various proofs: different immersions may differ by some automorphism of $\mathbb{N}$ which may not be primitive recursive. This however is guaranteed by the Lean class mechanism, which is able to simultaneously infer when a new type is encodable based on previous theorems, and fixes just one embedding for each such type.

3. Third, we notice that it may happen that the composition `encode ∘ decode` is not primitive recursive, which is undesirable. To fix this, we make it a requirement with the `primcodable` class:

   ```
   class primcodable (α : Type*) extends encodable α :=
     (prim [] : nat.primrec (λ n, encodable.encode (decode n)))
   ```

   and we require α, and β to be `primcodable`.

The definition of `primrec` can be extended to functions `f:α → β` whose types α, and β are `primcodable`. Specifically, for `f:α → β` to be `primrec` requires that the composition `encode ∘ f ∘ decode : ℕ → ℕ` is primitive recursive. This is how we can express this requirement in Lean:[2]

```
def primrec {α β} [primcodable α] [primcodable β]
(f:α → β):Prop := nat.primrec (λ n, encode ((decode α n).map f))
```

The relevant consequence of all this formalization is that Lean automatically deduces that `list ℤ` is `primcodable`; this follows from the `class` mechanism, from the fact that $\mathbb{Z}$ is `primcodable`, and by knowing that if a type α is `primcodable`, then so is `list α`.

Once everything is set up as described, we can eventually prove `theorem rpp_primrec` above, i.e. that for every `f:rpp`, the function `<f>:list ℤ → list ℤ` is `primrec`. We proceed by induction on `f`, by tackling the base cases `Id`, `Ne`, `Su`, `Pr`, `Sw` and the inductive cases `Co`, `Pa`, `It`, `If`.

5.2. Inductive cases

We illustrate the details of the case of parallel composition `f‖g`. Let `f`, and `g` be such that `<f>` and `<g>` are `primrec`. The goal is to prove that `f‖g` is `primrec`. In Lean, this amounts to prove the following lemma:

```
lemma rpp_pa {f g:rpp} (hf:primrec <f>) (hg:primrec <g>) :
        primrec <f‖g>
```

It starts by applying the definition of the parallel composition. For every fixed `l: list ℤ`, we have:

```
<f‖g> l = (<f> (take f.arity l))++(<g> (drop f.arity l))
```

So, we are left with the problem of proving that the right-hand side of the equation is `primrec`. We break down the problem into three sub-problems:

1. prove that the `append` operation `++` is `primrec`;

2. prove that the functions `take`, and `drop` are are `primrec`;

[2] The fact that `decode` has return type `option α` makes this expression more complicated: the function `map f` needs to be used.

3. prove that the composition of primitive recursive functions is `primrec`.

That `append` is `primrec`$_2$[3] is already proven in mathlib:

```
theorem list_append :
    primrec₂ ((++) : list α → list α → list α)
```

Furthermore, mathlib has proofs to demonstrate that the composition of two `primrec` elements or the application of one `primrec`$_2$ element to two `primrec` elements remains within the `primrec` set:

```
theorem comp {f:β → σ} {g:α → β}
    (hf:primrec f) (hg:primrec g) : primrec (λ a, f (g a))

theorem primrec₂.comp
    {f:β → γ → σ} {g:α → β} {h:α → γ}
    (hf:primrec₂ f) (hg:primrec g) (hh:primrec h) :
    primrec (λ a, f (g a) (h a))
```

So the sub-problems enumerated here above at points 1, and 3, are concluded.

For now let us assume that we also know how to deal with point 2, i.e. we have proved theorems `list_take` and `list_drop`. Under that assumption, we can conclude by writing:

```
lemma rpp_pa {f g : rpp} (hf : primrec <f>) (hg : primrec <g>) :
    primrec <f ‖ g> :=
(list_append.comp
    (comp hf (list_take.comp (const f.arity) primrec.id))
    (comp hg (list_drop.comp (const f.arity) primrec.id))).of_eq
$ λ l, by refl
```

The explanation of what this means is the following: what comes before the expression `.of_eq` is the statement that a certain "auxiliary" function, which we can call `F` for simplicity, is `primrec`. Figure 14 represents the structure of `F`: each block both defines part of the function and states that that part is `primrec`, at the same time. What comes after `.of_eq` is instead a proof that `F` is equal to `<f ‖ g>` for all inputs `l`: this is a definitional equality, so it can be proved easily in Lean tactics mode with `by refl`. Finally, `of_eq` is a theorem which given the hypotheses

- `F` is `primrec` (what's before `.of_eq`)
- `F` is equal to `<f ‖ g>` for all inputs (what's after `.of_eq`)

concludes that also `<f ‖ g>` is `primrec`, which is what we wanted to show.

[3]For functions which take two arguments, `primrec`$_2$ is used instead of `primrec`.

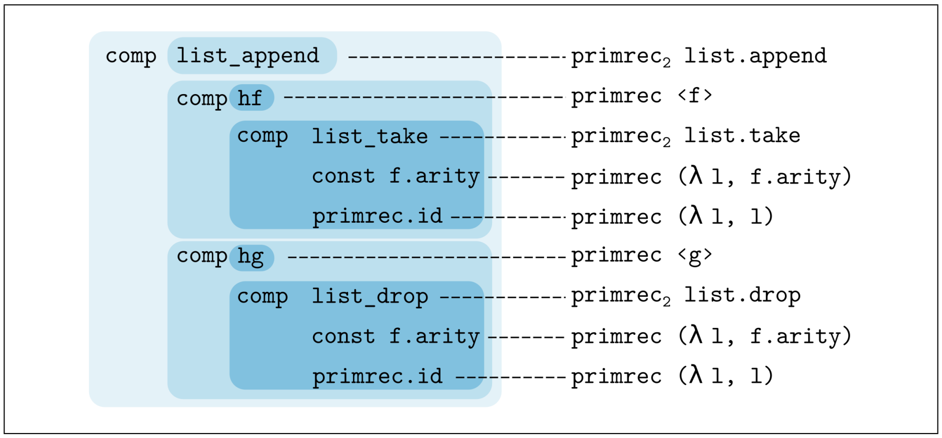


Figure 14: Diagram representing `rpp_pa`. For example, the first "`comp`" block means that, given the fact that `list.take` is `primrec`$_2$ and `(λ l, f.arity)`, `(λ l, l)` are `primrec`, then the composition `list.take f.arity l` is `primrec`.

We are eventually left with point 2 of the proof of `lemma rpp_pa`, i.e. the proofs of `lemma list_take`, and `lemma list_drop`.

Let us start by focusing on:

```
lemma list_take : primrec₂ list.take
```

in which, we recall, `list.take` is defined as:

```
def take : ℕ → list α → list α
| 0        a        := []
| (succ n) []       := []
| (succ n) (x :: r) := x :: take n r
```

i.e. a function recursive in both its arguments. The built-in Lean recursion principles for ℕ, and `list` α are both proven to be `primrec` in mathlib through theorems `nat_elim` and `list_rec`; unfortunately we cannot use them simultaneously for free in order to reason by induction on `take`.

We overcome the problem into two steps:

1. we define an “auxiliary” function `take2` in terms of the known function `foldl`, already proven to be `primrec`, and prove that `take2` is `primrec`;

2. we prove that `take2` is equal to `take` for all inputs, and conclude using `of_eq`.

The proof of equivalence is established through the use of the “special” induction principle `list.reverse_rec_on` which decomposes a list into its final element and all preceding elements, rather than the head and tail, feature that helps to reason with `take2`’s definition.

Once proven `list_take`, we can focus on the proof of `list_drop`. The key step is `lemma reverse_drop` here below:

```
lemma reverse_drop {α : Type*} (n : ℕ) (l : list α) :
    (l.drop n) = reverse (l.reverse.take (l.length - n))
```

Clearly, it expresses `list.drop` in terms of `list.take`, so the proof that `list.drop` is `primrec` proceeds smoothly and this concludes our overview of how the proof of `lemma rpp_pa` works.

Proving that `Co`, `It`, and `If` are `primrec` gets simpler to handle because the relevant functions are already proven to be `primrec`.

5.3. Base cases

The base cases are handled in a similar way, by building each function from simpler ones. In particular, the operations `Ne`, `Su`, `Pr` which respectively represent negation $x \mapsto -x$, successor $x \mapsto x+1$, predecessor $x \mapsto x-1$, all represent functions of type $\mathbb{Z} \to \mathbb{Z}$. Instead of focusing specifically on those functions, we found that it was actually easier to start from more basic functions close to the definition of integers in Lean, and progressively build more complex functions following exactly their definition and development in the mathlib library. We now focus on those more basic functions.

Let us look at the definition of integers:

```
inductive ℤ : Type
| of_nat : ℕ → ℤ
| neg_succ_of_nat : ℕ → ℤ
```

It is based on the two functions/constructors `of_nat`, and `neg_succ_of_nat` which can be proven to be `primrec` almost directly by unfolding the definitions of the embedding $\mathbb{Z} \to \mathbb{N}$ and noticing that through the compositions, the functions become two known functions `nat_bit0, nat_bit1 : ℕ → ℕ` which are already proven to be `primrec` in mathlib.

Other than `of_nat` and `neg_succ_of_nat`, the last important building block for functions of type $\mathbb{Z} \to \mathbb{Z}$ is the "Cases Principle" `int.cases_on` for integers:

```
int.cases_on : Π {f:ℤ → Type} (z:ℤ),
    (Π (n:ℕ), f (int.of_nat n)) →
    (Π (n:ℕ), f (int.neg_succ_of_nat n)) → f z
```

It states that if a function is defined for natural numbers and for negative numbers, then it is defined for all numbers. The reason this is important is that almost all basic functions with domain $\mathbb{Z}$ are defined by cases, breaking down the case where the input number is natural and where it is negative. We can express the fact that this cases principle is `primrec` in the following way:[4]

[4]The statement was slightly modified for simplicity. The original statement can be found in [18].

```
lemma int_cases {f:α → ℤ} {g h:α → ℕ → β}
    (hf : primrec f) (hg : primrec₂ g) (hh : primrec₂ h) :
    primrec (λ a, int.cases_on (f a) (g a) (h a))
```

This means that given three primrec/$\texttt{primrec}_2$ functions hf, hg, hh, we can compose them with the "Cases Principle" to get a new function, which the lemma states is primrec. We remark that all other cases/recursion/induction principles in mathlib are stated in a similar fashion. The proof, as usual, is based on the fact that more elementary operations are already proven to be primrec in mathlib.

6. Conclusion and developments

We give a concrete example of reversible programming in a proof-assistant. We think it is a valuable operation because programming reversible algorithms is not as much wide-spread as classical iterative/recursive programming, in particular by means of a tool that allows us to certify the result. Other proof assistants have been considered, and in fact the same theorems have also been proved in Coq, but we found that the use of the mathlib library together with the simp tactic made our experience with Lean much smoother. Furthermore, our work can migrate to Lean 4 whose stable release is announced in the near future. Lean 4 exports its source code as efficient C code [21]; our and other reversible algorithms can become efficient extensions of Lean 4, or standalone, and C applications.

The most application-oriented obvious goal to mention is to keep developing a Reversible Computation-centered certified software stack, spanning from a programming formalism more friendly than rpp, down to a certified emulator of Pendulum ISA, passing through compilator, and optimizer whose properties we can certify. For example, we can also think of endowing Pendulum ISA emulators with energy-consumption models linked to the entropy that characterize the reversible algorithms we program, or the Pendulum ISA object code we can generate from them.

A more speculative direction, is to keep exploring the existence of programming schemes in rpp able to generate functions, other than Cantor Pairing, etc., which we can see as discrete space-filling functions, whose behavior we can describe as steps, which we count, along a path in some space.

References

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

[2] A. De Vos, Reversible Computing - Fundamentals, Quantum Computing, and Applications, Wiley, 2010.

[3] K. Morita, Theory of Reversible Computing, Monographs in Theoretical Computer Science. An EATCS Series, Springer, 2017. doi:10.1007/978-4-431-56606-9.

[4] L. Paolini, M. Piccolo, L. Roversi, A class of recursive permutations which is primitive recursive complete, Theor. Comput. Sci. 813 (2020) 218–233. doi:10.1016/j.tcs.2019.11.029.

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

[6] L. Paolini, M. Piccolo, L. Roversi, On a class of reversible primitive recursive functions and its turing-complete extensions, New Generation Computing 36 (3) (2018) 233–256. doi:10.1007/s00354-018-0039-1.

[7] A. B. Matos, L. Paolini, L. Roversi, On the expressivity of total reversible programming languages, in: I. Lanese, M. Rawski (Eds.), Reversible Computation, Springer International Publishing, Cham, 2020, pp. 128–143.

[8] A. B. Matos, Linear programs in a simple reversible language, Theor. Comput. Sci. 290 (3) (2003) 2063–2074. doi:10.1016/S0304-3975(02)00486-3.

[9] A. Matos, L. Paolini, L. Roversi, The fixed point problem of a simple reversible language, TCS 813 (2020) 143–154. doi:https://doi.org/10.1016/j.tcs.2019.10.005.

[10] L. de Moura, S. Kong, J. Avigad, F. van Doorn, J. von Raumer, The lean theorem prover (system description), in: A. P. Felty, A. Middeldorp (Eds.), Automated Deduction - CADE-25, Springer International Publishing, Cham, 2015, pp. 378–388.

[11] G. Maletto, L. Roversi, Certifying Algorithms and Relevant Properties of Reversible Primitive Permutations with Lean, in: Claudio Antares Mezzina and Krzysztof Podlaski (Ed.), Reversible Computation - 14th International Conference, RC 2022, Urbino, Italy, July 5-6, 2022, Proceedings, Vol. 13354 of Lecture Notes in Computer Science, Springer, 2022, pp. 111–127. doi:10.1007/978-3-031-09005-9_8.

[12] G. Cantor, Ein beitrag zur mannigfaltigkeitslehre, Journal für die reine und angewandte Mathematik 84 (1878).

[13] M. P. Szudzik, The Rosenberg-Strong Pairing Function, CoRR abs/1706.04129 (2017). arXiv:1706.04129.

[14] L. Paolini, M. Piccolo, L. Roversi, A certified study of a reversible programming language, in: T. Uustalu (Ed.), TYPES 2015 postproceedings, Vol. 69 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2017.

[15] A. Asperti, C. Sacerdoti Coen, E. Tassi, S. Zacchiroli, User interaction with the matita proof assistant, Journal of Automated Reasoning 39 (2007) 109–139. doi:https://doi.org/10.1007/s10817-007-9070-5.

[16] J. Jacopini, P. Mentrasti, Generation of invertible functions, Theor. Comput. Sci. 66 (3) (1989) 289–297. doi:10.1016/0304-3975(89)90155-2.

[17] G. Maletto, A Formal Verification of Reversible Primitive Permutations, BSc Thesis, Dipartimento di Matematica – Torino, October 2021. https://github.com/GiacomoMaletto/RPP/tree/main/Tesi.

[18] G. Maletto, RPP in LEAN, https://github.com/GiacomoMaletto/RPP/tree/main/Lean.

[19] M. Carneiro, Formalizing computability theory via partial recursive functions, in: 10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA, 2019, pp. 12:1–12:17. doi:10.4230/LIPIcs.ITP.2019.12.

[20] M. Carneiro, computability.primrec, https://leanprover-community.github.io/mathlib_docs/computability/primrec.html.

[21] L. d. Moura, S. Ullrich, The Lean 4 Theorem Prover and Programming Language, in: A. Platzer, G. Sutcliffe (Eds.), Automated Deduction – CADE 28, Springer International Publishing, Cham, 2021, pp. 625–635.