Cantor Pairing in a reversible programming language

Francesco Rossini
francesco.rossini@edu.unito.it
Dipartimento di Informatica – Torino
Torino, Italy

ABSTRACT

Yarel is a core reversible programming language that implements a class of permutations, defined recursively, which are primitive recursive complete. The current development of Yarel is quite preliminary and almost every library to make it usable is missing. We here present the encoding of Cantor pairing in Yarel. The encoding improves the original one, given in the work that introduces the class of permutations which Yarel relies on.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**.

KEYWORDS

Reversible computation, Cantor pairing.

ACM Reference Format:
Francesco Rossini. 2019. Cantor Pairing in a reversible programming language. In *Proceedings of ACM Conference (Programming'19).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Various models catch the meaning of reversible computation. One of them is the class of Reversible Primitive Permutations (RPP), introduced in [2, 3], and simplified in [4]. Their aim is to show how to extend the formal pattern under the design of Primitive Recursive Functions (PRF) in order to capture computational reversible behaviors. Every permutation in RPP has $\mathbb{Z}^k$ as domain and co-domain, for some $k \in \mathbb{N}$. RPP contains total functions and is primitive recursive complete, i.e. every primitive recursive function f can be compiled to an equivalent $f^\bullet$ in RPP [4]. The translation $(\_)^\bullet$: PRF → RPP relies on proving that RPP represents Cantor Pairing, a pair of isomorphisms between $\mathbb{Z}^2$ and $\mathbb{Z}^2$ that stack elements of $\mathbb{N}$. So, RPP is at least as expressive as PRF.

Yarel. Yarel is a functional language which implements RPP, Therefore it inherits its properties, such as Primitive Recursive Completeness. Concerning the syntax, the topmost grammatical construct of Yarel are the modules, every one with a name. Within them, you can import other modules, declare functions - specifying type and arity - and define them. Figure 1 is an example. Currently, the only type is `int`. The notation `3 int` in the declaration states

```
1  module CantorPairing                {
2    dcl Sum : 3 int
3    def Sum := /*x,y,0*/ /3 1 2/;
4               /*0,x,y*/ it[inc]|id;
5               /*x,x,y*/ /3 2 1/;
6               /*y,x,x*/ it[/*y,x*/    it[inc];
7                            /*y+x,x*/ id|dec];
8                            /*y+x,x-1*/ ];
9               /*y+∑_{i=0}^{x} i,  0,x*/ /3 1 2/
10              /*x,  y+∑_{i=0}^{x} i,0*/
11   dcl CP : 3 int /* three args */
12   def CP := /*x,y,0*/  /2 1 3/;
13             /*y,x,0*/  it[inc]|id;
14             /*y+x,x,0*/ Sum;
15             /*y+x,x+∑_{i=0}^{x+y} i,0*/ }
```

Figure 1: The function `CP` in Yarel.

that the function has arity 3 and all its arguments are integers. Functions in Yarel are the least class that we can build from identity `id`, increment `inc`, decrement `dec`, negation `neg` and finite permutations `/`$i_1 \ldots i_n$`/`, by means of serial composition f`;`g, parallel composition$f$`|`g, iteration `it[`f`]`, selection `if[`f`,`g`,`h`]` or inverse `inv[`f`]`, given some already defined functions. Comments `/*...*/` help to show the flow of the values in Yarel functions, which is *point-free*, i.e. a language of combinators with no explicit reference to variables. For example, `/*`$x, y, 0$`*/it[inc]|id;/*`$x+y, y, 0$`*/` at line 4, the iteration `it` of `inc` maps $(x, y, 0)$ to $(x + y, y, 0)$ because `it[inc]` is a function of arity 2 *iterating the increment on the first argument as many times as the absolute value of the second argument.* Clearly it sums its two arguments. Always at line 4, the 2-arity function `it[inc]` in parallel with `id` preserves the value 0 of the third argument from the input to the output.

Cantor Pairing. As already mentioned, Cantor pairing serves to show that Yarel is PRF-complete. We here present how to encode Cantor pairing as a reversible primitive permutation in Yarel whose arity is strictly smaller than the arity of the original Cantor pairing in [4]. We recall that a Cantor pairing is a function $\mathbb{N}^2 \to \mathbb{N}$ defined as $cp(x, y) = x + \sum_{i=0}^{x+y} i$. We can turn this definition into the set of reversible functions of Figure 1 where `CP` represents $cp(x, y)$, but with a further arguments, which is a 0-valued ancilla. `CP` first sums x and y and yields $x + y$. Then, it calls `Sum` which sums all the natural numbers from 0 to the value of its first argument, which amounts to $x + y$. This yields $\sum_{i=0}^{x+y} i$. Its last step is to add $\sum_{i=0}^{x+y} i$ to the second argument. This is neatly simpler than in [4] where, instead, `CP` relies on computing triangular numbers.

But what is the inverse of Cantor pairing? As recalled, for example, in [4], it is $cu(z) = \langle z - \sum_{i=0}^{k-1} i, (k-1) - (z - \sum_{i=0}^{k-1} i)\rangle$ from $\mathbb{N}$ to $\mathbb{N}^2$, where k is the least value such that $k \leq z$ and $z < \sum_{i=0}^{k} i$. An

Programming'19, April 2019, Genova, ITALY

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
1  public class seqComp implements RPP
                  {
2    public seqComp() { } // Constructor
3    // l(eft-hand side) of the sequential
       composition
4    private RPP l = new RPP()
                  {
5      private RPP f = new inc();  // an
       instance of inc
6      private final int a = f.getA();
7      public int[] b(int[] x) { return this.f.b
       (x); }
8      public int getA() { return this.a; }
                  };
9    // r(ight-hand side) of the sequential
       composition
10   private RPP r = new RPP()
                  {
11     private RPP f = new dec();  // an
       instance of dec
12     private final int a = f.getA();
13     public int[] b(int[] x) { return this.f.b
       (x); }
14     public int getA() { return this.a; }
                  };
15   private final int a = l.getA(); // Same
       arity of l or r
16   public int[] b(int[] x)          { // Seq.
       composition
17     return this.r.b(this.l.b(x)); }
18   public int getA() { return this.a; }
                  }
```

Figure 2: The compilation of `scExample` in Java.

implementation `CU` of $cu(z)$ in Yarel exists whose arity is lower than the one of the namesake function in [4]. For lack of space we cannot report its full details. What really matters is that our `CU` is *equivalent* to `inv[CP]`, i.e. the inverse we can get for free in Yarel by means of the constructor `inv` which, applied to any Yarel-function f, yields its inverse. This means that we can indifferently use `CU` or its more operational `inv[CP]` in Figure 1 to unfold any given natural z to its corresponding pair (x, y) such that $z = cp(x, y)$. This is an effective and non trivial example showing that the inverse of a Yarel function really computes the inverse of the corresponding abstract function.

Yarel-IDE. The practical uses of this language goes behind our immagination, but an initial one is to translate it into a library of java functions and their inverse. Yarel-IDE is an integrated development environment, distributed as an Eclipse plug-in, that we generate by means of XText, a further Eclipse plug-in for developing domain specific languages. XText naturally leads to compile a domain specific language implemented with it into Java classes. Yarel is not an exception and its operational semantics drives the compilation. As an example, Figure 2 is the object code of compiling a function `seqComp`, defined as `inc;dec`. The class has name `seqComp` and implements a suitable interface `RPP`. Two private fields `l` and `r` contain the compilation of the left and of the right-hand side of the sequential composition. I.e., `l` is an anonymous class that contains an instance of the compilation `inc()` of `inc` whose arity is in `a` and whose behaviour is in `int[] b(int[] x)`. Analogous comments hold on `r`. Given `l` and `r`, we let the arity of the sequential composition coincide with `l.getA()` while the behavior is the sequential composition of `l.b` and `r.b` in lines 15 and 16. Every function of Yarel is compiled under analogous patterns.

Remarkably, compiling some given f in Yarel to Java let the compilation of f and of its inverse f^{-1} available in Java as methods that we can freely use with the proviso of never dropping any of the arguments that assure the reversibility. We see this as pursuing the vision of [1], focused on formalizing classes of classical functions with lossless inverses. Moreover, whatever we write in Yarel becomes compliant with the object oriented conceptual tools that Java supplies without any *ad-hoc* extension of Yarel with object oriented features.

2 CONCLUSIONS

Being Cantor pairing a bijection means that every element of $\mathbb{N}^2$ maps uniquely to $\mathbb{N}$ and vice versa. In an ideal programming reversible setting, we should need only (the representation of) z to return its corresponding pair (x, y).

This way we could take advantage of an isomorphism between $\mathbb{N}$ and $\mathbb{N}^k$ to compress data; in fact, the pair of arguments (x, y) would be compressed into the single argument z — the Cantor pairing — without any loss of information. Since z preserves the initial amount of information, it is sufficient to go back to initial (x, y) by means of Cantor unpairing.

We are currently working on implementing a Cantor pairing whose outputs are all 0-valued but one, which, of course, has to carry the encoding z with it.

In this direction, we plan to reformulate Yarel in order to drop the constraint which forces to program arity-respecting functions, i.e. with type $\mathbb{Z}^k \to \mathbb{Z}^k$.

REFERENCES

[1] David A. Huffman. 1959. Canonical forms for information-lossless finite-state logical machines. *IRE Trans. Information Theory* 5, 5 (1959), 41–59. https://doi.org/10.1109/TIT.1959.1057537

[2] Luca Paolini, Mauro Piccolo, and Luca Roversi. 2016. A Class of Reversible Primitive Recursive Functions. *Electronic Notes in Theoretical Computer Science* 322, 18605 (2016), 227–242. https://doi.org/10.1016/j.entcs.2016.03.016

[3] Luca Paolini, Mauro Piccolo, and Luca Roversi. July 2018. On a Class of Reversible Primitive Recursive Functions and Its Turing-Complete Extensions. *New Generation Computing* 36, 3 (July 2018), 233–256. https://doi.org/10.1007/s00354-018-0039-1

[4] Luca Paolini, Mauro Piccolo, and Luca Roversi. June 2016. A class of Recursive Permutations which is Primitive Recursive complete. , 27 pages. Under consideration for publication. Available at http://www.di.unito.it/~paolini/papers/2017rpp.pdf.