Splitting Recursion Schemes into Reversible and Classical Interacting Threads

Publisher:

Springer International Publishing

Published version:

DOI:10.1007/978-3-030-79837-6_12

Splitting recursion schemes into reversible and classical interacting threads

Armando B. Matos[1], Luca Paolini[2][0000−0002−4126−0170], and Luca Roversi[2][0000−0002−1871−6109]

[1] Universidade do Porto, Departamento de Ciência de Computadores – Portugal
armandobcm@yahoo.com
[2] Università degli Studi di Torino, Dipartimento di Informatica – Italy
{luca.paolini,luca.roversi}@unito.it

Abstract. Given a simple recursive function, we show how to extract from it a reversible and an classical iterative part. Those parts can synchronously cooperate under a Producer/Consumer pattern in order to implement the original recursive function. The reversible producer is meant to run on reversible hardware. We also discuss how to extend the extraction to a more general compilation scheme.

1 Introduction

Our goal is to compile a class of recursive functions in a way that parts of the object code produced can leverage the promised green foot-print of truly reversible hardware. This work illustrates preliminary steps towards that goal. We focus on a basic class of recursive functions in order to demonstrate its feasibility.

Contributions. Let `recF[p,b,h]` be a recursive function defined in some programming formalism, where `p` is a *predecessor* function, `h` a *step* function, and `b` a *base* function. We show how to compile `recF[p,b,h]` into `itFCls[b,h]` and `itFRev[p,pInv]` such that:

$$\texttt{recF[p,b,h]} \simeq \texttt{itFCls[b,h]} \parallel \texttt{itFRev[p,pInv]} \ , \qquad (1)$$

where: (i) "$\simeq$" stands for "*equivalent to*"; (ii) `itFCls[b,h]` is a classical `for`-loop that, starting from a value produced by `b`, iteratively applies `h`; (iii) `itFRev[p, pInv]` is a reversible code with two `for`-loops in it one iterating `p`, the other its inverse `pInv`; (iv) "$\parallel$" is interpreted as an *interaction* between `itFCls[b,h]` and `itFRev[p,pInv]`, according to a Producer/Consumer pattern, where `itFRev[p, pInv]` produces the values that `itFCls[b,h]` consumes to implement the initially given recursion `recF[p,b,h]`. In principle, `itFRev[p,pInv]` can drive a real reversible hardware to exploit its low energy consumption features.

In this work we limit the compilation scheme (1) to use: (i) a predecessor `p` such that the value `p(x)-x` is any *constant* $\Delta_{\texttt{p}}$ equal to, or smaller than, `-1`; (ii) recursion functions `recF[p,b,h]` whose *condition* identifying the base case is `x<=0` instead than the more standard `x==0`; this means that more than one base *non positive* value for `recF[p,b,h]` exists in the interval $[\Delta_{\mathtt{p}} + 1, 0]$. This slight generalization will require a careful management of the reversible behavior of `itFRev[p,pInv]` and its interaction with `itFCls[b,h]` in order to reconstruct `recF[p,b,h]`.

Contents. Section 2 sets the stage to develop the main ideas about (1), restricting `recF[p,b,h]` to a recursive function that identifies its base case by means of the standard condition `x==0`; this ease the description of how `itFRev[p,pInv]` and `itFCls[b,h]` interact. Section 3 extends (1) to deal with `recF[p,b,h]` having `x<=0`, and not `x==0`, to identify its base case(s); this impacts on how `itFRev[p,pInv]` must work. In both cases, the programming syntax we use can be interpreted into the reversible languages SRL [3,4] and RPP [5,6,4], up to minor syntactic details. Section 4 addresses future work.

```
Fix recF(x)                          {
    if   (c(x)) { b(x);          }
    else        { h(x,recF(p(x))); } }
```

Fig. 1. The recursive function `recF`.

```
 /*** Assumption: the inital value of x is 3 */
 x = p(x)       // ==2
 x = p(x)       // ==1
 x = p(x)       // ==0
 y = b(x)       // ==b(p(p(p(3))))
 y = h(x,y)     // ==h(p(p(p(3))),b(p(p(p(3)))))
 x = pInv(x)    // ==pInv(p(p(p(3))))==p(p(3))
 y = h(x,y)     // ==h(p(p(3)),h(p(p(p(3))),b(p(p(p(3))))))
 x = pInv(x)    // ==pInv(p(p(3)))==p(3)
 y = h(x,y)     // ==h(p(3),h(p(p(3))
                //         ,h(p(p(p(3))),b(p(p(p(3)))))))
 x = pInv(x)    // ==pInv(p(3))==3
 y = h(x,y)     // ==h(3,h(p(3),h(p(p(3))
                //      ,h(p(p(p(3))),b(p(p(p(3))))))))
```

Fig. 2. Iterative unfolding `recF(3)`: the bottom-up part.

2 The driving idea

Let `recF[p,b,h]` in (1) have a structure as in **Fig. 1** where `b(x)` is the *base* function, `h(x,y)` the *step* function, `p(x)` the *predecessor* `x-1`, and `c(x)` the *condition* `x==0` to identify a unique base case.

Fig. 2 details out `h(3,h(p(3),h(p(p(3)), h(p(p(p(3))),b(p(p(p(3))))))))`, unfolding of `recF(3)`. Every comment asserts a property of the values that `x` or `y` stores. Lines 2–4 unfold an iteration that computes `p(p(p(3)))`, which eventually sets the value of `x` to `0`. Line 5 starts the construction of the final value of `recF(3)` by applying the base case of `recF`, i.e. `b(x)`. By definition, let `pInv` denote the inverse of `p`, i.e. `pInv(p(z))==p(pInv(z))==z`, for any `z`. Clearly, in our running example, the function `pInv(x)` is `x+1`. Lines 6–13 alternate `h(x,y)`, whose result `y`, step by step, gets closer to the final value `recF(3)`, and `pInv(x)`, which produces a new value for `x`.

```
1  s = 0, e = 0, g = 0, w = 0
2  w = w + x;
3  for (i = 0; i<=w; i++)      {
4    if      (x> 0) { g++; }
5    else if (x==0) { e++; }
6    else           { s++; }
7    x = p(x);               }
8  
9  for (i = 0; i<=w; i++)                 {
10   x = pInv(x);
11   if      (x> 0) { g--; y = h(x,y); }
12   else if (x==0) { e--; y = b(x);   }
13   else           { s--;             } }
14 w = w - x;
```

Fig. 3. Iterative `itF` equivalent to `recF`.

Let us call `itF` the code in **Fig. 3**. It implements `recF` by means of finite iterations only. Continuing with our running example, if we run `itF` here above starting with `x==3`, then `x==0` holds at line 8, just after the first `for`-loop; after the second `for`-loop `y==recF(3)` holds at line 14.

The code of `itF` has two parts. Through lines 2–7 the variable `g` counts how many times `x` remains positive, the variable `e` how many it stays equal to `0`, and the variable `s` how many it becomes negative. In this running example we notice that `x` never becomes negative, for the iteration at lines 3–7 is driven by the value of `x` which, initially, we can assume non negative, and which `p(x)` decreases of a single unity. We shall clarify the role of `s` later. Lines 9–13 undo what lines 2–7 do by executing `pInv(x)`, `g--`, `e--`, `s--`, i.e. the inverses, in reversed order, of `p(x)`, `g++`, `e++`, `s++`. So the correct values of `x` are available at lines 12, and

11, ready to be used as arguments of `b(x)` and `h(x,y)` to update `y` as in **Fig. 3**, according to the results we obtain by the recursive calls to `recF`.

```
1  s = 0, e = 0, g = 0, w = 0
2  w = w + x;
3  for (i=0; i<=w; i++)      {
4    if      (x> 0) { g++; } //number of times x is ‘g’reater than 0
5    else if (x==0) { e++; } //number of times x is ‘e’qual to 0
6    else           { s++; } //number of times x is ‘s’maller than 0
7    x = p(x);               }
8
9  for (i=0; i<=w; i++)                                          {
10   x = pInv(x);
11   if      (x> 0) { g--; /* Value of x for h availabe here */ }
12   else if (x==0) { e--; /* Value of x for b availabe here */ }
13   else           { s--;                                     } }
14 w = w - x;
```

Fig. 4. Reversible side of `itF`.

Now, let us focus on the main difference between **Fig. 4** and **Fig. 3**.

Both `x=b(x)` and `y=h(x,y)` at lines 12, and 11 of **Fig. 3** are missing from lines 12, and 11 of **Fig. 4**. Dropping them let **Fig. 4** be the *reversible side* of `itF`; calling `b(x)` and `h(x,y)` in it generates`y`, which is the result we need, so preventing the possibility to reset the value of every variable dealt with in **Fig. 4** to their initial value. This is why we also need a *classical side* of `itF` that generates `y` in collaboration with the *reversible side* in order to implement the initial `recF` correctly.

```
/*** Assumption. The value of the input x is available here */
/* Inject the current x at line 2 of itFRev to let it start */
iterations = /* Probe line 9 of itFRev to get the
                number of iterations to execute   */
y = b(/* Probe line 14 of itFRev to get the argument */);
for (i = 0; i<iterations; i++)                 {
  y = h(/* Probe line 12 itFRev to get
           the first argument of h     */ , y); }
```

Fig. 5. Classical side of `itF`: the consumer `itFCls`.

The previous observations lead to **Fig. 5** which defines the *classical side* `itFCls` of `recF`, and to **Fig. 6** which defines the *reversible side* `itFCRev` of `recF`.

```
1  s = 0, e = 0, g = 0, w = 0;
2  x = /* Inject here the value of x at line 2 of itFCls */
3  w = w + x;
4  for (i = 0; i<=w; i++)      {
5    if      (x> 0) { g++; }
6    else if (x==0) { e++; }
7    else           { s++; }
8    x = p(x);                 }
9  /* itFCls probes here g which has the number of iterations */
10 for (i = 0; i<=w; i++)                                  {
11   x = pInv(x);
12   if      (x> 0) { g--; /* itFCls probes here the
13                            first argument value of h */ }
14   else if (x==0) { e--; /* itFCls probes here the
15                            argument value of b       */ }
16   else           { s--;                                 } }
17 w = w - x;
```

Fig. 6. Reversible side of `itF` updated to be the producer `itFRev` of the values that the consumer `itFCls` needs.

So, here below we can illustrate how `itFCls` and `itFRev` synchronously interact, `itFRev` producing values, `itFCls` consuming them as arguments of `b(x)` and `h(x,y)`.

Line 2 of `itFCls` is the starting point of the synchronous interaction between `itFCls` and `itFRev`; its comment:

```
/* Inject the current x at line 2 of itFRev to let it start */
```

describes what, in a fully implemented version of `itFCls`, we expect in that line of code. The comment says that `itFCls` injects (sends, puts) its input value `x` to line 2 of the *reversible side* `itFRev` (cf. **Fig. 6**). Once `itFRev` obtains that value at line 2, as outlined by:

```
/* Inject here the value of x from line 2 of itFCls */
```

its `for`-loop at lines 4–8 executes.

After line 2, `itFCls` stops at line 3. It waits for `itFRev` to produce the number of times that `itFCls` has to iterate line 7. Accordingly to:

```
/* Probe line 9 of itFRev to get the number of iterations to execute */
```

`itFRev` makes that value available in its variable `g` at line 9:

```
/* itFCls probes here g which has the number of iterations */ .
```

Once gotten the value in `iterations`, `itFCls` proceeds to line 5 and stops, waiting for `itFRev` to produce the argument of `b` which is eventually available for probing at line 14 of `itFRev`.

Once the argument becomes available `b` is applied, and `itFCls` enters its `for` -loop, stopping at line 7 at every iteration. The reason is that `itFCls` waits for line 12 in `itFRev` to produce the value of the first argument of `h(x,y)`. This interleaved dialog between line 7 of `itFCls` and line 12 of `itFRev` lasts `iterations` times.

```
Fix recG(x)                       {
  if (x<=0) { b(x);             }
  else      { h(x,recG(p(x))); } }
```

Fig. 7. The generic structure of `recG`.

3 From recursion to iteration

We now generalize what we have seen in Section 2. Inside (1) we use `recG` of **Fig. 7** instead than `recF` of **Fig. 1**. This requires to generalize **Fig. 6**.

From the introduction we recall that, given a *predecessor* `p(x)`, we define $\Delta_{\mathtt{p}}$ = `p(x)-x`, which is a negative value. In this section $\Delta_{\mathtt{p}}$ can be any *constant* k `<=-1`, not only k `==-1`; this requires to consider the slightly more general *condition* `x<=0` in `recG`. For example, let `p(x)` be `x-2`. The computation of `recG(3)` is `h(3,h(p(3),h(p(p(3)),b(p(p(3))))))` which looks for the least n of iterated applications of `p(x)` such that `p(...p(3)...)<=0`; in our case we have `2==`n`<3`.

Fig. 8 introduces `itG` which generalizes `itF` in **Fig. 3**.

The scheme `itG` iteratively implements any recursive function whose structure can be brought back to the one of `recG`. We remark that line 1 in **Fig. 8** initializes ancillae `s`, `e`, `g`, and `w`, like **Fig. 3** initializes the namesake variables of `itF`, but line 2 of `itG` has new ancillae `z`, `predDivX`, and `predNotDivX`.

We also assume an initial *non negative* value for `x`. The reason is twofold. Firstly, it keeps our discussion as simple as possible, with no need to use the absolute value of `x` to set the upper limit of every index `i` in the `for`-loops that occur in the code. Second, negative values of `x` would widen our discussion about what a classical recursive function on negative values is and about what its reversible equivalent iteration has to be; we see this as a very interesting subject connected to [1], which is much more oriented than us to optimization issues of recursively defined functions.

We start observing that line 3 of `itG` sets `w` to the initial value of `x`; the reason is that every `for`-loop, but the one at lines 10–12, has to last `x+1` iterations, and `x` changes in the course of the computation; so, `w` stores the initial value of `x` and stays constant from line 4 through line 21. In fact it can change at lines 22–33. We will see why, but `w` is eventually reset to its initial value `0` at line 36.

```
s = 0, e = 0, g = 0, w = 0;
z = 0, predDivX = 0, predNotDivX = 1;
w = w + x; /* x is assumed to be the input */
for (i = 0; i <= w; i++)   {
  if      (x >  0) { g++; }
  else if (x == 0) { e++; }
  else             { s++; }
  x = p(x);                    }

for (i = 0; i < e; i++)                  {
  predDivX = predDivX + predNotDivX;
  predNotDivX = predDivX - predNotDivX; }

for (j = 0; j < predDivX; j++)            {
  for (i = 0; i <= w; i++)               {
    x = pInv(x);
    if      (x >  0) { g--; y = h(x,y); }
    else if (x == 0) { e--; y = b(x);   }
    else             { s--;             }}}

for (j = 0; j < predNotDivX; j++)                          {
  w++;
  for (i = 0; i <= w; i++)                                {
    x = pInv(x);
    if      (x >  0) { g--;
                       x = p(x);
                       if      (z <  0) {                }
                       else if (z == 0) { y = b(x); z++; }
                       else             { y = h(x,y);    }
                       x = pInv(x);                       }
    else if (x == 0) { e--;             }
    else             { s--;             }}
  w--;                                                     }
for (i = 0; i < predNotDivX; i++)  {
  z--;                             }
w = w - x;
/* y carries the output */
```

Fig. 8. The iterative function `itG`.

With the here above assumptions, given a non negative `x`, and in analogy to `itF`, the `for`-loop at lines 4–8 of `itG` iterates the application of `p(x)` as many times as `w+1`, i.e. the initial value of `x` plus `1`. So, the value of `x` at line 9 is equal to `w+(w+1)*`Δ_p which cannot be positive. In particular, all the values that `x` assumes in the `for`-loop at lines 4–8 belong to the following interval:

$$I(\mathtt{w}) \triangleq [\mathtt{w+(w+1)*}\Delta_p, \mathtt{w+w*}\Delta_p, \ldots, \mathtt{w+}\Delta_p, \mathtt{w}] \tag{2}$$

from the least to the greatest; the counters `g`, `e`, `s` say how many elements of $I(\mathtt{x})$ are greater, equal or smaller than `0`, respectively. Depending on `0` to belong to $I(\mathtt{x})$ determines the behavior of the reminder part of `itG`, i.e. lines 10–36.

We distinguish two cases in order to illustrate them.

First case. Let `w%`Δ_p `== 0`, i.e. the integer value Δ_p divides with no reminder the initial value of `x` that we find in `w`. So, $\mathtt{0} \in I(\mathtt{x})$, which implies the following relations hold at line 9:

$$\mathtt{e == 1} \qquad \mathtt{g == -}\frac{\mathtt{w}}{\Delta_p} \qquad \mathtt{s == (w+1)-g-e}\ . \tag{3}$$

```
if      (e <  0) {                                      }
else if (e == 0) { predDivX = predDivX+predNotDivX;
                   predNotDivX = predDivX - predNotDivX; }
else             {                                      }
```

Fig. 9. A possible replacement of lines 10–12 in **Fig. 8**.

Lines 10–12 execute exactly once, swapping `predDivX` and `predNotDivX`. As a remark, we could have well used the `if`-selection in **Fig. 9** (a construct of RPP) in place of the `for`-loop at lines 10–12, but we opt for a more compact code.

Swapping `predDivX` and `predNotDivX` sets `predDivX==1` and `predNotDivX==0`, computationally exploiting that Δ_p divides `w` with no reminder: the `for`-loop body at lines 15–19 becomes accessible, while lines 22–33, with `for`-loops among them, do not. Lines 15–19 are identical to lines 10–16 of `itF` in **Fig. 4** which we already know to correctly apply `b(x)` and `h(x,y)` in order to simulate the recursive function we start from.

As a second case. Let `w%`Δ_p `!= 0`, i.e. the integer value Δ_p divides the initial value of `x` that `w` stores, *but with some reminder.* So, $\mathtt{0} \notin I(\mathtt{x})$, which imply:

$$\mathtt{e == 0} \qquad \mathtt{g == -}\left\lfloor\frac{\mathtt{w}}{\Delta_p}\right\rfloor \qquad \mathtt{s == (w+1)-g-e} \tag{4}$$

hold at line 9. Lines 11–12 cannot execute, leaving `predDivX` and `predNotDivX` as they are: lines 22–33 become accessible and the `for`-loop at lines 15–19 does not. Line 22 increments `w` to balance the information loss that the rounding of `g` in (4) introduces; line 33 recovers the value of `w` when the outer `for`-loop starts.
The `if`-selection at lines 25–32 identifies when to apply `b(x)`, which must be followed by the required applications of `h(x,y)`. We know that $0 \notin I(x)$, so `x==0` can never hold. Clearly, `s--` is executed until `x>0`. But the *first* time `x>0` holds true we must compute `b(p(x))`, because the *base* function `b(x)` *must be used the last time* `x` assumes a negative value, *not the first time* it gets positive; lines 26–30 implement our needs. Whenever `x>0` is true, the value of `x` is one step ahead the required one: we get one step back with line 26 and, if it is the first time we step back, i.e. `z==0` holds, then we must execute line 28. If not, i.e. `z!=0`, we must apply the *step* function at line 29. Line 30, restores the right value of `x`. Finally, the `for`-loop at line 34 sets `z` to its initial value.

At this point, in order to obtain the fully reversible version of **Fig. 8** we must think of replacing the calls to `h(x,y)` and `b(x)` at lines in 28, and 29 by means of actions that probe the value of `x`, in analogy to **Fig. 6**, lines 12 and 14. The full details are in [7] which we look as a playground with Java classes that implement **Fig. 8** and **Fig. 5** as synchronous and parallel threads, acting as a producer and a consumer.

4 Future work

We have shown that we can decompose every classical recursive function, based on a *predecessor* that decreases every of its input by a constant value, into reversible and classical components that cooperate to implement the original recursive functions under a Producer/Consumer pattern (see (1)).

Firstly, we plan to extend (1) to recursive functions `recF` based on predecessors `p` not limited to a constant $\Delta_{\mathtt{p}}$ not greater than `-1`. A predecessor `p` should be at least such that:

1. $\Delta_{\mathtt{p}}$ is not necessarily a constant. For example, $\Delta_{\mathtt{p}}$ `== -3` on even arguments, and `-2` on odd ones can be useful;
2. the predecessor can be an integer division `x/k`, for some given `k>0`, like in a dichotomic search, which has `k==2`.

Secondly, we aim at generalizing (1) to a compiler $[\![\cdot]\!]$:

$$
\begin{aligned}
[\![\mathtt{p}]\!] &= \mathtt{some\ implementation\ code} \\
[\![\mathtt{pInv}]\!] &= \mathtt{!}[\![\mathtt{p}]\!]\mathtt{,\ i.e.\ implementation\ that\ inverts\ }[\![\mathtt{p}]\!] \\
[\![\mathtt{recF[p,b,h]}]\!] &= \mathtt{itFCls[}[\![\mathtt{b}]\!]\mathtt{,}[\![\mathtt{h}]\!]\mathtt{]} \parallel \mathtt{itFRev[}[\![\mathtt{p}]\!]\mathtt{,}[\![\mathtt{pInv}]\!]\mathtt{]}\ .
\end{aligned}
\tag{5}
$$

The domain of $[\![\cdot]\!]$ should be a class R of recursive functions built by means of standard composition schemes, starting from a class of predecessors `p1`, `p2`, ... each of which must have the corresponding inverse function `p1Inv`, `p2Inv`, ...

In these lines we want to explore interpretations of || more liberal than the essentially obvious synchronous Producer/Consumer that we implement in [7]. We shall very likely take advantage of parallel discrete events simulators as described in [8,9] in order to get rid of any explicit synchronization between the pairs of reversible-producer/classical-consumer that (5) would recursively generate when applied to an element in R.

We also plan to follow a more abstract line of research. The compilation scheme (5) recalls Girard's decomposition $A \to B \simeq\, !A \multimap B$ of a classical computation into a linear one that can erase/duplicate computational resources. Decomposing `recF[p,b,h]` in terms of `itFCls[b,h]` and `itFRev[p,pInv]` suggests that the relation between reversible and classical computations can be formalized by a linear isomorphism $A^n \multimapboth B^n$ between tensor products A^n, and B^n of A, and B, in analogy to [2]. Then we can think of recovering classical computations by some functor, say γ, whose purpose is, at least, to forget, or to inject replicas, of parts of A^n, and B^n in a way that $(\gamma A^n \to \gamma A^n) \uplus (\gamma A^n \leftarrow \gamma A^n)$ can be their type. The type says that we move from a reversible computation to a classical one by choosing which is input and which is output, so recovering the freedom to manage computational resources as we are used to when writing classical programs.

References

1. E. A. Boiten. Improving recursive functions by inverting the order of evaluation. *Science of Computer Programming*, 18(2):139 – 179, 1992.
2. R. P. James and A. Sabry. Information effects. In J. Field and M. Hicks, editors, *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. ACM, 2012.
3. A. B. Matos. Linear programs in a simple reversible language. *Theor. Comput. Sci.*, 290(3):2063–2074, 2003.
4. A. B. Matos, L. Paolini, and L. Roversi. On the expressivity of total reversible programming languages. In I. Lanese and M. Rawski, editors, *Reversible Computation*, pages 128–143, Cham, 2020. Springer International Publishing.
5. 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.
6. L. Paolini, M. Piccolo, and L. Roversi. A class of recursive permutations which is primitive recursive complete. *Theor. Comput. Sci.*, 813:218–233, 2020.
7. L. Roversi, A. Matos, and L. Paolini. Eclipse java project rev2iterrev. https://github.com/LucaRoversi/Rec2IterRev.
8. M. Schordan, T. Oppelstrup, D. R. Jefferson, and P. D. B. Jr. Generation of reversible C++ code for optimistic parallel discrete event simulation. *New Gener. Comput.*, 36(3):257–280, 2018.
9. M. Schordan, T. Oppelstrup, M. K. Thomsen, and R. Glück. *Reversible Languages and Incremental State Saving in Optimistic Parallel Discrete Event Simulation*, pages 187–207. Springer International Publishing, Cham, 2020.