Reactive Sorting Networks (Supplementary Material)

Bjarno Oeyen
bjarno.oeyen@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Sam Van den Vonder
sam.van.den.vonder@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Wolfgang De Meuter
wolfgang.de.meuter@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

1 Introduction

The following two pages serve as supplementary material for the paper titled "Reactive Sorting Networks" [3].

A Bitonic Sorting Network in REScala

Listing 11 contains an implementation of a REScala [4] program that constructs bitonic sorting networks. Both sequences of signals (with type `Seq[Signal[T]]`) and arrays of signals (with type `Array[Signal[T]]`) are used, as some kinds of rewiring mechanisms are easier to express with (immutable) lists (e.g., splitting a list in two halves) and others with (mutable) arrays (e.g., adding new comparators to an existing network).

```
type CMP[T] = (Signal[T], Signal[T]) =>
  (Signal[T], Signal[T])

def flip[T](cmp: CMP[T]): CMP[T] =
  (line1: Signal[T], line2: Signal[T]) =>
    cmp(line1, line2).swap

def bitonic[T](arr: Seq[Signal[T]],
               cmp: CMP[T]): Seq[Signal[T]] = {
  arr match {
    case List(a, b) =>
      val (out1, out2) = cmp(a, b)
      List(out1, out2)
    case _ =>
      val n = arr.length
      val bottom = bitonic(arr.take(n / 2), flip(cmp))
      val top = bitonic(arr.drop(n / 2), cmp)
      merge(bottom ++ top, cmp)
  }
}

def merge[T](arr: Seq[Signal[T]],
             cmp: CMP[T]): Seq[Signal[T]] = {
  val woven = weave(arr, cmp)
  if (woven.length == 2) {
    woven
  } else {
    val (a, b) = woven.splitAt(woven.length / 2)
    merge(a, cmp) ++ merge(b, cmp)
  }
}

def weave[T](arr: Seq[Signal[T]],
             cmp: CMP[T]): Seq[Signal[T]] = {
  val n = arr.length
  val total: Array[Signal[T]] = arr.toArray
  for (i <- 0 until n / 2) {
    val (newA, newB) = cmp(total(i), total(n / 2 + i))
    total(i) = newA; total(n / 2 + i) = newB
  }
  total.toSeq
}
```

Listing 11. A REScala implementation of Bitonic Sort.

B Insertion Sorting Network

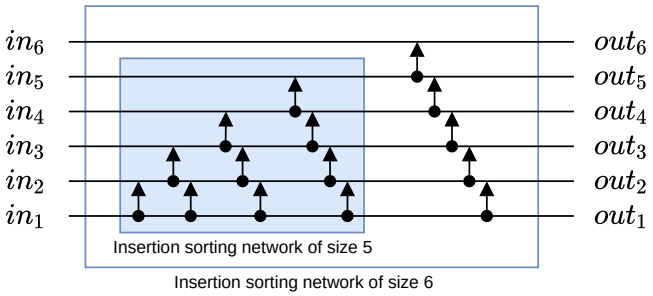

Figure 5. Insertion sorting network of size 6. Although the network is constructed differently, the sorting network itself is equivalent to a bubble sorting network of the same size. This property holds for all sizes.

```
(defr (insert-one n cmp)
  (if (= n 2)
      cmp
      (let
        (def p (parallel (insert-one (- n 1) cmp) cmp))
        (snake-on p n (- n 1)))))

(defr (insertion n cmp)
  (def stack (insert-one n cmp))
  (if (= n 2)
      stack
      (let
        (def small-network (insertion (- n 1) cmp))
        (ror (parallel small-network identity)
             stack))))
```

Listing 12. Implementation of `insertion`: which generates an insertion sorting network reactor of a given size.

Listing 12 contains an implementation of an insertion sorting network [2, Section 5.3.4] written in Haai [3]. The sorting networks generated by `insertion` are identical, at least structurally with respect to the positions of the comparators in the network, to those generated by `bubble` in Listing 7 of the main paper, as shown in Figure 5.

C Batcher's Odd-Even Mergesort

```
(defr (batcher n cmp)
  (if (= n 2)
      cmp
      (let
        (def batcher-half (batcher (/ n 2) cmp))
        (ror (parallel batcher-half batcher-half)
             (merge n cmp)))))

```

```
(defr (merge n cmp)
  (ror (weave n cmp)
       (merge2 1 (/ n 2) n cmp)))

(defr (merge2 i j n cmp)
  (def k (/ (- n (* i j)) 2))
  (def identities (parallel-n identity k))
  (def parallel-weaves (parallel-n (weave j cmp) i))
  (def r (parallel identities
                   (parallel parallel-weaves
                             identities)))
  (if (= k 1)
      r
      (ror r (merge2 (+ (* 2 i) 1) (/ j 2) n cmp))))

(defr (weave n cmp)
  (defr (loop r i)
    (if (> i k)
        r
        (loop (post-weave r cmp i (+ i k)) (+ i 1))))
  (def k (/ n 2))
  (loop (parallel-n identity n) 1))
```

Listing 13. Implementation of `batcher`, which constructs a Batcher's odd-even sorting network of a given size.

Listing 13 contains an implementation of a Haai program that generates sorting networks inspired by Batcher's Odd-Even Mergesort [1]. Just like bitonic sorting networks, each (sub)network of size n contains two recursively-generated subnetworks (of size $n/2$) that each sort one half of the inputs of the larger network. However, unlike bitonic sorting networks, the sorting order is not reversed by either of these smaller sorting networks. As a consequence, a different merging strategy is used for this type of sorting networks.

Also note that the definition of `weave` differs from the one used in the implementation of a bitonic sorter from the main paper, where the `do` notation was used instead of a recursive deployment of `loop`. Although the definition is different, both reactors produce, given the same inputs, equivalent reactors.

References

[1] Kenneth E. Batcher. 1968. Sorting Networks and Their Applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968 (AFIPS Conference Proceedings, Vol. 32)*. Thomson Book Company, Washington D.C., 307–314. https://doi.org/10.1145/1468075.1468121

[2] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume III: Sorting and Searching, 2nd Edition.* Addison-Wesley, Reading, MA, USA.

[3] Bjarno Oeyen, Sam Van den Vonder, and Wolfgang De Meuter. 2020. Reactive Sorting Networks. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS@SPLASH 2020* (Virtual, USA) *(REBLS@SPLASH 2020)*. ACM, New York, NY, USA. https://doi.org/10.1145/3427763.3428316

[4] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging between Object-Oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity* (Lugano, Switzerland, April 22-26) *(MODULARITY '14)*, Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld (Eds.). ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2577080.2577083