Reactive Sorting Networks

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

Abstract

Sorting is a central problem in computer science and one of the key components of many applications. To the best of our knowledge, no reactive programming implementation of sorting algorithms has ever been presented.

In this paper we present reactive implementations of so-called sorting networks. Sorting networks are networks of comparators that are wired up in a particular order. Data enters a sorting network along various input wires and leaves the sorting network on the same number of output wires that carry the data in sorted order.

This paper shows how sorting networks can be expressed elegantly in a reactive programming language by aligning the visual representation of a sorting network with the canonical DAG representation of reactive programs. We use our own experimental language called Haai to do so. With a limited number of built-in higher-order reactive programs, we are able to express sorting networks for bubble sort, insertion sort, bitonic sort, pairwise sort and odd-even merge sort.

CCS Concepts: • **Software and its engineering → Data flow languages**.

Keywords: Reactive Programming, Higher-Order Programming, Reactor Composition, Sorting Networks

ACM Reference Format:
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 '20), November 16, 2020, Virtual, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3427763.3428316

1 Introduction

Implementing sorting algorithms forms an essential programming exercise in any computer science curriculum and is a key part of many applications. Implementing the standard textbook set of sorting algorithms is also used to serve as a litmus test to study the expressiveness of new programming languages and programming paradigms. To the best of our knowledge, this exercise is currently lacking from the literature on reactive programming.

This paper presents an implementation of a number of elementary sorting algorithms in an experimental reactive programming language called Haai [22]. One of the key inspirations of our research is that the canonical representation of reactive programs, using directed acyclic graphs (DAGs) as dependency graphs, naturally maps onto the visual representation of so-called *sorting networks*. Although sorting networks are nowadays primarily used by the parallel programming community [24, 25], we found it interesting to look at their applications in reactive programs.

Conceptually, a sorting network is an acyclic network of comparators that receives a fixed number of incoming values as input and which produces a sorted permutation of these values. The simplest approach to implement a sorting network is to write a program that iterates in a particular way over an array of values to be sorted, such that, at each computational step, the values in the array represent the state of the sorting network. We propose a different approach to implement sorting networks. By implementing sorting networks in a reactive programming language, the textual representation (i.e. the implementation in code) directly corresponds to the visual representation (i.e. the dependency graph) of a sorting network.

In this paper, we introduce *reactive sorting networks*, which are sorting networks that sort data in a reactive manner. We have identified several shortcomings that embedded reactive programming languages (which are reactive programming languages that extend a non-reactive language with reactive functionality) have when they are being used to implement these sorting networks. Instead of mixing two language semantics to construct a sorting network, which makes the program more difficult to reason about (among other issues), we implement our reactive sorting networks in a purely reactive programming language (one that is not embedded in another programming language). In short, our approach for implementing reactive sorting networks in this language is to *generate* networks of the desired type (e.g., a bubble sorting network) and size (i.e. the number of elements to sort) by a reactive program itself. This is achieved by using a set of built-in *higher-order reactive operators* that are used to

REBLS '20, November 16, 2020, Virtual, USA

ACM ISBN 978-1-4503-8188-8/20/11…$15.00
https://doi.org/10.1145/3427763.3428316

combine the different subcomponents of the different types of sorting networks. Because the understanding of these programs relies on understanding the DAG they produce as output, we introduce a new type of diagram, called *Jacquard Diagrams*, that helps to understand how such sorting networks (among other types of networks) can be constructed as a reactive program. Understanding these diagrams allows one to visualise how the higher-order program generates the required sorting network.

The structure of this paper is as follows. In Section 2 we introduce reactive sorting networks in more detail and describe the difficulties that we encounter when trying to generate them in an embedded reactive programming language. Section 3 introduces Haai, a prototypical reactive programming language that is used in the rest of this paper. Section 4 presents the built-in higher-order operators that generate new reactive programs out of existing ones, and their corresponding Jacquard Diagrams. We use these operators in Section 5 where we describe our implementations of a number of sorting networks: bubble sorting networks, bitonic sorting networks and pairwise sorting networks.

2 Reactive Sorting

A sorting network can be seen as an abstract device that is built around the concept of "wires" that carry individual values [17, Section 5.3.4]. There is only one kind of fundamental operation, namely a "compare-and-exchange" operation that compares the values carried by two wires and exchanges them if they are not in the correct order. A sorting network is a carefully crafted linkage of compare-and-exchange operations that will ensure that the values carried by the wires are sorted when they reach the output of the network.

Consider a reactive dashboard application that perpetually displays 6 incoming values produced by 6 temperature sensors in sorted order. A simple way to achieve this is to build a sorting network with 6 inputs that will incrementally sort the input values as they arrive. There are many ways how such a sorting network can be constructed. Some networks are constructed with the goal to minimise the number of compare-and-exchange operations used [3, 8], whilst others are constructed with the goal to implement the working of conventional sorting algorithms such as bubble sort and insertion sort. For this paper, we only consider the latter approach.

A simple network (based on bubble sort [17, Section 5.3.4]) for sorting 6 incoming wires ($\texttt{in}_\texttt{i}$) resulting in 6 outgoing wires ($\texttt{out}_\texttt{i}$) is depicted in Figure 1. A connection between two wires corresponds to a compare-and-exchange operation that compares the values on two wires, and then puts them in the right order. The network performs a bubble sort because the first vertical "slice" of the network consists of a series of comparators that percolate the largest value sitting on all wires towards the end of the network (i.e. the topmost

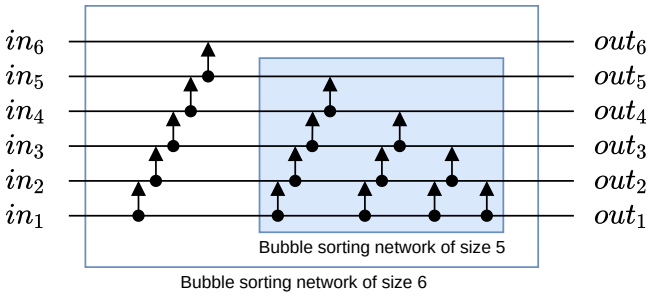


Figure 1. Bubble sorting network of size 6. Data flows left to right and the arrows show the direction of how values are sorted.

wire, $\texttt{out}_6$). The values that remain on wires $\texttt{in}_1$ to $\texttt{in}_5$ are subsequently sent through a smaller bubble sorting network, which has exactly the same structure. In other words, an existing bubble sorting network of size n can be extended to create an n+1 bubble sorting network by prefixing it with a stack of n comparator modules that are chained together, such that the largest value is "bubbled" to the topmost wire, and the other wires are then subsequently sorted by the smaller sorting network of size n. Figure 1 indicates this inductive structure of the bubble sorting network by means of nested rectangles.

The introduction of this paper already mentioned the correspondence between sorting networks and reactive programs: both consist of a DAG of operations that are "wired up" and that together perform some prescribed functionality. After the "wiring up" phase, the values start flowing through the network. In this paper, we investigate the idea of "reactive sorting networks" which are sorting networks that "reactively" sort incoming data as it arrives in the program, without needing to execute each compare-and-exchange operator every time a single input changes. By implementing a reactive sorting network in a reactive programming language, we get this reactivity for free.

Listing 1 contains an implementation of a REScala [27] program where the `bubbleSN` function is used to create reactive sorting networks. This Scala function has 1 argument `in` which is an array of REScala signals (i.e. "live carriers of data") that carry `Int` values, representing the values of the input wires. The body of the function imperatively constructs a bubble sorting network such that the output signals are stored in the same array that was given as input.

The inner loop on Line 3 of Listing 1 is responsible for generating a stack of comparator modules between a decreasing number of wires. For example, the first stack of comparators connects the wires $\texttt{in}_1$ until $\texttt{in}_6$. The next stack of comparators connects the wires $\texttt{in}_1$ until $\texttt{in}_5$. And so on. The outer loop on Line 2 repeats this process $n-1$ times (where n is the number of input signals) to generate the $n-1$ vertical stacks of comparators that make up the network.

```
1 def bubbleSN(in: Array[Signal[Int]]): Unit =
2   for (sortedLn <- in.indices.reverse.tail)
3     for (ln <- 0 until sortedLn) {
4       val a = in(ln); val b = in(ln+1)
5       in(ln) = Signal { a() min b() }
6       in(ln+1) = Signal { a() max b() }
7     }
```

Listing 1. Generating a bubble sorting network in REScala.

Problem Statement. Using the code in Listing 1, we have identified two shortcomings that embedded reactive programming languages have to implement reactive sorting networks.

First, the programmer has to deal with **two distinct language semantics**. The `bubbleSN` function itself is just a regular Scala function that uses Scala loops to iteratively construct and instantiate a sorting network. The other programming language is the REScala language, which is the language responsible to construct the dependency graph of the reactive component of the program. In this program, the applications of the "`Signal {}`" constructors extend the dependency graph during the execution of the `bubbleSN` function. Remark that the code *inside* a "`Signal {}`" constructor is also Scala code that expresses the behaviour of a signal. The code inside is said to be *lifted* such that it is executed each time one of the signals that it depends on changes. The same expression inside or outside a "`Signal {}`" block has, therefore, entirely different semantics (either the active semantics as defined by Scala, or the reactive semantics as defined by REScala), and programmers using REScala continuously need to switch between both types of language semantics. In addition, inadvertently interleaving reactive code (i.e. REScala code) with active code (i.e. Scala code) can result in undesirable run-time behaviour [7].

Second, the **code does not correspond to the description of the actual algorithm**. Indeed, the Scala function from Listing 1 is just a function that imperatively constructs the dependency graph. The only way to reconstruct the bubble sorting algorithm from the code is to visualise the dependency graph that is being constructed during the execution of the `bubbleSN` function, and then try to recognise the bubble sorting network from this visualisation. This issue occurs as in REScala signal composition is the only way how new signals can be created. Thus a dependency graph can only be constructed from living signals: there is no built-in abstraction that helps to reason in terms of (the parts of) the dependency graph itself. One can argue that this is not that big of a problem for a bubble sorting network. However, if we take a more complicated (but more efficient) sorting network (e.g., a bitonic sorting network), this issue becomes more prominent. We have included an implementation of a REscala program that constructs bitonic sorting networks in [21, Section A] (which contains supplementary material that belongs to this paper). We invite the reader to compare the complexity of this implementation with our implementation of a bitonic sorting network shown here in Listing 9. Given some exposure to the semantics of the language used, the structure of the code in Listing 9 directly corresponds to the visual representation of the bitonic network (cf. Figure 4), whereas the code in [21, Section A] does not.

In the rest of this paper we will present our solution to these problems in a reactive programming language called Haai. In Section 5 we present a reactive implementation of five famous sorting networks, implemented in Haai, such that:

- **All code is expressed in the same programming language**, namely Haai. The key feature of Haai that enables this is that reactive programs can be generated by other, so-called higher-order, reactive programs (called reactors). Using a set of well-defined composition operators that can be used on these reactors, it is then possible to compose and reuse parts of the dependency graph. In such a language the composition of parts of the dependency graph becomes part of the semantics of the reactive program itself.
- **The structure of the Haai code directly corresponds to the algorithm**, i.e. to the visual representation of the sorting network. This is a consequence of how we have chosen to tackle the first problem: reactors are used as the building blocks to create sorting networks, and reactors can create new reactors. Thus, when the sorting network has an inductive structure, that structure directly corresponds to the recursive use of reactors.

We first present a brief overview of Haai.

3 Haai

Haai[1] is an experimental reactive programming language that we use to research composition techniques for reactive programs. In this section we present a brief overview of Haai's syntax and semantics.

3.1 Reactors, Deployments and Signals

In Haai, the dependency graph of a reactive program is built out of reusable components called *reactors*. A reactor is an abstraction for a dependency graph that can be reused in the definition of other reactors. A reactor consists of zero or more source nodes, any number of internal nodes, and at least one sink node. Listing 2 contains a definition of a reactor that sorts the values it receives on its two source nodes. The name of the reactor is `cae` (compare-and-exchange) and the two source nodes are named `a` and `b`. The body of the reactor defines the internal nodes of the reactor, and in Listing 2 two named signals are defined using the `def` syntax (which will be explained in more detail shortly). For example, the `smallest` signal is the result of deploying the `min` reactor on the two source signals `a` and `b`. Finally, the `out` syntax is used

[1] A prototype implementation of Haai can be found online: https://soft.vub.ac.be/~boeyen/haai/.

```
1 (defr (cae a b)
2   (def smallest (min a b))
3   (def largest (max a b))
4   (out smallest largest))
```

Listing 2. Reactor definition of a comparator reactor that compares the values passed via its source nodes.

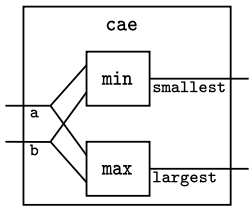


Figure 2. Visualisation of the cae reactor. The source nodes are shown on the left, and the sink nodes on the right. The connections inside the rectangle show how the internal nodes are connected.

to denote the sink nodes of a reactor. cae has two sink nodes: namely the smallest and largest signals. When the out syntax is absent in a reactor's definition, the last expression describes the sole sink signal. A visualisation that shows the dependency graph of cae is contained in Figure 2.

A reactor is an abstraction for a dependency graph that describes the *behaviour* of (a part of) the reactive program. On its own, a reactor does not produce any values. It needs to be instantiated first. Every reactor can be instantiated multiple times in the same reactive program. We will call these instances *deployments*. A deployment is responsible for storing all run-time information that is pertinent to one specific instance of the dependency graph (such as the latest values for each signal contained therein).

A reactor is deployed by making a copy of its dependency graph and replacing its source nodes with value-carrying signals. Similar to signals or behaviours in other reactive languages [5, 10], signals make up a reactive program's dependency graph and they can carry a value that changes over time. In code, reactors are deployed by a deployment expression, which has the same syntax as a procedure application in Scheme [16]. For example, the expression (min a b) from Listing 2 deploys the min reactor on the a and b signals. The sink node of this deployment contains a new signal whose value is equal to the smallest value of its two operand signals. When a reactor has more than one sink node (like the cae reactor), the def syntax can be used to bind all the sink signals in the deploying environment. The cae reactor can thus be deployed as follows:

```
(def (low-temp high-temp)
  (cae temp-a temp-b))
```

Similar to the define-values syntax in Racket [12, Section 4.5.3], def is used to bind multiple sink signals, produced by a single reactor deployment, at once. When there is only one identifier the parenthesis surrounding the identifier can be omitted.

```
1 (defr (min a b)
2   (if (< a b) a b))
```

Listing 3. Definition of a reactor that returns the smallest value as carried by its two source signals.

Haai follows the "reactors all the way down" philosophy [22] and thus the language does not contain procedures that can be applied. Even the expression (sqrt 4) corresponds to a reactor deployment (namely the deployment of the sqrt reactor on the signal carrying the number 4). Due to various implementation-level reasons that will be discussed later, a distinction is made between time-varying signals (which are signals whose value changes over time, such as time), and constant signals (whose value does not change after the signal's creation, such as 4). Reactors cannot create time-varying signals ex-nihilo, since the value of a time-varying signal is always expressed in terms of another time-varying signal. We therefore assume that the reactive runtime contains a number of *primitive signals* whose values changes naturally over time (outside of the control of the reactive program). The time signal is the only primitive signal available to Haai programs. Other primitive signals can be defined by the programmer, but the means to do so are not part of this paper.

3.2 Conditional Signals

Conditional signals are signals whose value depends on a certain run-time condition, which usually manifest themselves as signals created using the if syntax. Just like in Scheme, its syntax is defined as follows: (if ⟨condition⟩ ⟨consequent⟩ ⟨alternative⟩). When an expression that contains an if is deployed, a conditional signal is created whose value is equal to the value carried by either the ⟨consequent⟩ or ⟨alternative⟩ signal, depending on the truth value carried by the ⟨condition⟩ signal. For example, the definition of the min reactor (as used by cae) is given in Listing 3. When that reactor is deployed, its sink signal is a conditional signal that carries the value of a if that value is smaller than the value carried by the signal b (and vice versa).

Although the following note is not a crucial part to understand the main narrative of the paper, it is interesting to note that the if syntactic form is not the only construct that gives rise to the existence of conditional signals. Ordinary deployment expressions are also able to instantiate conditional signals when the expression in operator position of a deployment expression (i.e. the first sub-expression of a deployment expression) is a time-varying signal. These conditional signals will carry the value as carried by the sink signals of the deployment of the reactor that is being carried by the signal in operator position. We will call these deployment expressions *dynamic deployments* since the deployment

used by these conditional signals changes dynamically at run-time.

3.3 The Deployment Phase and Run-Time Phase

Central to the design of the language is the fact that a Haai program is run in two separate phases. The *deployment phase* is responsible for constructing the complete dependency graph, and the *propagation phase* will use this dependency graph to percolate changes of the primitive signals. Every time a primitive signal carries a different value, a propagation turn is executed by the reactive runtime.

Deployment Phase. A reactive program's dependency graph is constructed by loading all signal and reactor definitions of a Haai program in the global environment. Initially, this global environment only contains bindings for all the built-in primitive reactors and signals.

The *complete* dependency graph is constructed by evaluating the deployment expressions of each top-level signal definition. This is trivial for simple deployment expressions that only deploy a single reactor, but dynamic deployment expressions (deployment expressions where the value in operator position is a time-varying signal, see Section 3.2) are more complex. For dynamic deployments it is, in general, impossible to construct the complete dependency graph during the deployment phase as each time the signal in operator position carries a new reactor value, that reactor needs to be deployed. However, while a complete technical overview of our implementation of Haai is outside the scope of this paper, it suffices to know that during the deployment phase the program is analysed to determine the set of possible reactor values that can be carried by a signal (which is at worst an over-approximation). Using this analysis for all signals used in the operator position of a deployment expression, it is possible, in most cases, to deploy all possible reactors during the deployment phase, such that during a propagation turn the reactive runtime only has to toggle between these deployments. Only when the deployment-time analysis cannot determine a finite set of deployments to pre-allocate, which occurs in complex situations using recursion (see Section 4.3), the deployment phase fails.

Propagation Phase. The propagation phase is responsible for propagating values through the reactive program's dependency graph. Whenever a primitive signal carries a new value, a *propagation turn* is executed. Our approach to propagate changes is inspired by FrTime's [5] propagation algorithm as it ensures that only the signals affected by a change are updated, and ensures the absence of glitches. One minor difference is that in our implementation, there is no need to create new deployments during the propagation phase, as the complete dependency graph is already constructed in the first phase. Instead, signals in the dependency graph are disabled (and later re-enabled) when the value they carry is not being used during a given turn.

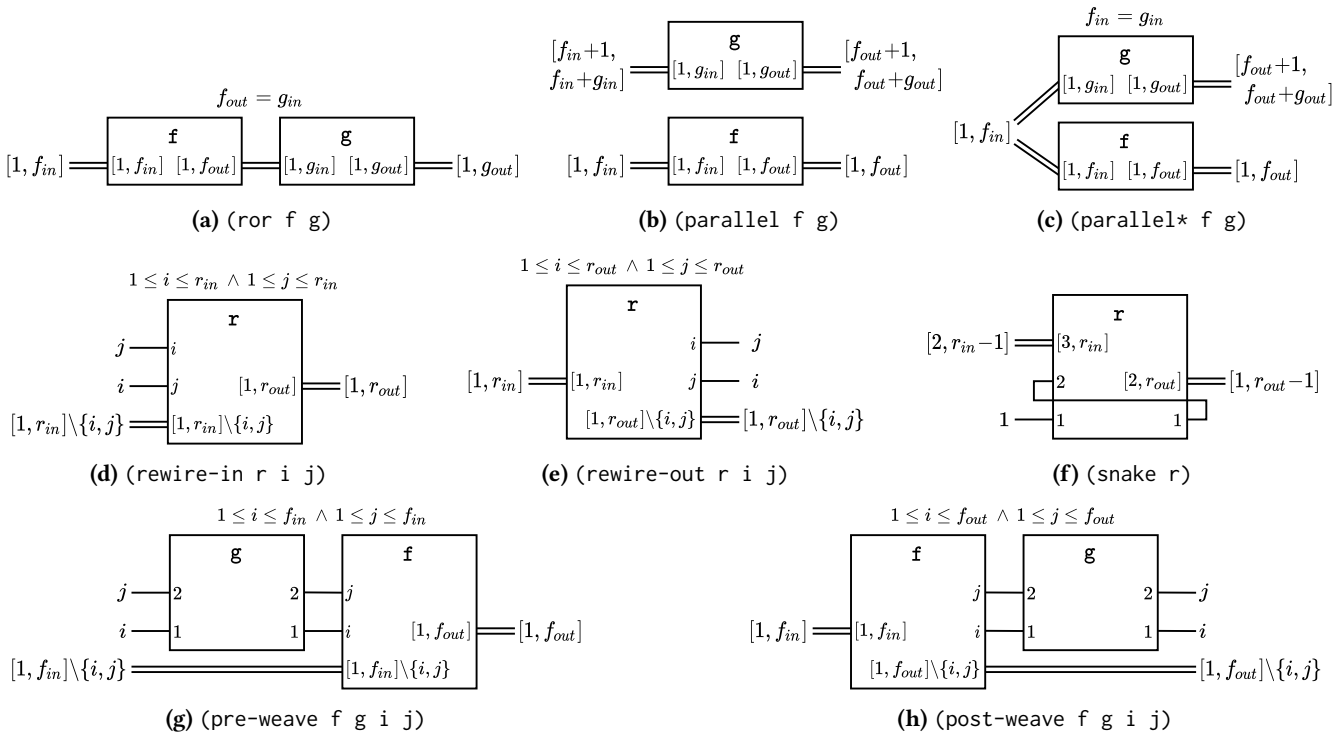
As the deployment phase can be executed independently from the propagation phase, the propagation algorithm can be replaced with another, more performant, algorithm. As sorting networks are often used on parallel processing hardware [24] (or even on GPUs [26]) it can be beneficial to replace our single-threaded propagation algorithm with a parallel propagation algorithm. This can not only speed up the execution time of a single propagation turn [6, 15], but can also allow for multiple turns to be executed in parallel [9, 20]. This can substantially increase the performance of Haai programs, especially since no run-time (re)deployments have to take place since all deployments were allocated in the deployment phase. In addition, when all primitive operations in the dependency graph have a worst-case time complexity of $O(1)$, then each propagation turn also has a worst-case time complexity of $O(1)$ (i.e., strongly reactive [7]) as the number of (primitive) operations is finite (as determined during the deployment phase). In other words, the worst-case time complexity of a single propagation turn is always independent of the data carried by the primitive signals.

4 Higher-Order Weaving Reactors

In the previous section we have shown how new reactors can be defined using the `defr` syntactic form. However, `defr` is not the only way in which reactors can be created by a program. Reactors can also be created as a result of using a *weaving operator*. A weaving operator takes as input one or more reactors, and produces a new reactor whose dependency graph is the result of combining the reactors given as input in a particular way.

Weaving operators will be crucial to express sorting networks. The idea is that the canonical visual representation of the networks (which relies on combining certain "boxes and lines" together) will be mimicked by applying the right weaving operators on existing reactors to form the desired sorting network.

A complete overview of all the built-in weaving operators present in Haai will be shown later in this section. We first exemplify their usage by means of an example. Listing 4 shows an alternative implementation of the `cae` reactor from Listing 2. In its definition, the `parallel*` weaving operator is used to combine the already existing `min` and `max` reactors similar to how they were combined earlier. In other words, conceptually, both definitions result in the same dependency graph being created at deployment-time. Deployments of the reactors created by `parallel*` will deploy both constituent reactors on all of the source signals of the composite reactor deployment. The sinks of the composite reactor deployment are the result of appending the sinks of the first constituent deployment to the sinks of the second constituent deployment. Thus, deployments of this alternative definition of `cae` are semantically equivalent to its definition in Listing 2.



(a) (ror f g) **(b)** (parallel f g) **(c)** (parallel* f g)

(d) (rewire-in r i j) **(e)** (rewire-out r i j) **(f)** (snake r)

(g) (pre-weave f g i j) **(h)** (post-weave f g i j)

Figure 3. Jacquard diagrams of the composition and rewiring operators.

```
(def cae
  (parallel* min max))
```

Listing 4. Alternative definition of cae using the point-free parallel* weaving operator.

4.1 Jacquard Diagrams

Haai has a total of 8 built-in weaving operators: ror, parallel, parallel*, snake, rewire-in, rewire-out, pre-weave and post-weave.

From our experience, understanding the semantics of these operators just from reading a textual description alone can be cumbersome. Something that we found invaluable to understand the behaviour of the weaving operators is to draw a diagram that shows how the sources and sinks of the original reactors are rewired and connected by a weaving operator. We call these diagrams *Jacquard diagrams*. They are named after Joseph Marie Jacquard, the inventor of the Jacquard machine. Which is a machine that uses punched cards to control the inner workings of a loom [11]. Similar to how punched cards described the weaving patterns produced by a loom, a Jacquard diagram describes the various ways in which a weaving operator "weaves" the sources and sinks of its argument reactors.

The Jacquard diagrams of all built-in weaving operators are shown in Figure 3. The source nodes of the reactor that is produced by a weaving operator are shown on the left, and the right side shows its sink nodes. A box represents the DAG of one of the argument reactors. Its source and sink nodes are connected respectively to the left and right side of the box. The notation r_{in} is used to denote the total number of source nodes (inputs) of reactor r, and r_{out} denotes the number of its sink nodes (outputs). Signal nodes are 1-indexed. Thus the last source node of the reactor r is annotated as the $r_{in}{}^{\text{th}}$ source node. A single line represents a connection from one signal node to another. Both ends of the line should be annotated with a single index, which denotes which signal nodes are connected. For simplicity, no arrows are shown on the connections as data always flows from a value producer (sources of the composite reactor, or sinks of a constituent reactor) to a value consumer (sources of a constituent reactor or the sinks of the composite reactor). A double line represents the connections between multiple signal nodes. Both ends of the line should be annotated with an interval of indices.

For example, Figure 3c shows the Jacquard diagram of parallel*, a weaving operator that we used earlier in Listing 4. The reactor produced by parallel* takes a copy of its two constituent reactors f and g and it connects all the source nodes of the composite reactor to all the source nodes of each constituent reactor. The reactor produced by parallel* has, in total, f_{out}+g_{out} sink nodes: f's sink nodes are connected to the the first f_{out} of the composite reactor, and g's sink nodes are connected to the last g_{out} sink nodes.

4.2 The Operators

We briefly explain each higher-order weaving operator.

`ror`. The `ror` weaving operator (reactor after reactor) concatenates two reactors `f` and `g` in sequence, such that all the sink signals of `f` are connected to the sources of `g` (in the same order). As a consequence, the number of source nodes `g` must be equal to the number of sink nodes of `f`.

`parallel`. The `parallel` weaving operator places two reactors (`f` and `g`) in parallel. The first f_{in} source signals of the composite reactor are connected to `f`, and the last g_{in} source signals are connected to `g`. The first f_{out} source signals of the composite reactor are connected to `f`'s sinks, and the last g_{out} source signals to `g`'s sinks.

`parallel*`. Like `parallel`, `parallel*` places two reactors in parallel, but the sources of the composite reactor are connected to both the sources of `f` and `g` in the same order. The sink nodes of the composite reactor are constructed in the same way as in `parallel`.

`rewire-in` *(resp.* `rewire-out`*).* This weaving operator takes one reactor as input, and two indices (`i` and `j`). It will produce a new reactor where the source nodes (resp. sink nodes) on positions `i` and `j` are swapped.

`snake`. The `snake` weaving operator takes a single reactor as input and produces a new reactor whose first sink node is connected to the second source node. Thus, the resulting reactor has one source and one sink less compared to the original reactor. As long as the first sink node does not already depend (either directly or indirectly) on the second source, the resulting dependency graph will not contain a cycle.

`pre-weave` *(resp.* `post-weave`*).* This higher-order weaving operator takes two reactors `f` and `g` as input, and will produce a new reactor where the reactor `g` is placed before (resp. after) the reactor `f` on two specific source (resp. sink) nodes.

These higher-order weaving operators are implemented as primitive reactors themselves. This means that, to use them, they have to be deployed on a number of source signals. Weaving reactors are therefore able to work with time-varying signals. For example, when `ror` is deployed on two reactor signals, it produces a new composite reactor every time one of the operand signals carries a different reactor.

In contrast to reactor signals that can be time-varying, the indices of `rewire-in`, `rewire-out`, `pre-weave` and `post-weave` are required to be constant in the current implementation of Haai. This is needed to simplify the deployment-time analysis that determines which signals can carry which reactor values at run-time (which is needed for dynamic deployments).

```
1 (defr (factorial n)
2   (if (= n 0)
3       1
4       (* n (factorial (- n 1)))))
```

Listing 5. An implementation of a `factorial` reactor.

```
1 (defr (parallel-n r n)
2   (if (= n 1)
3       r
4       (parallel r (parallel-n r (- n 1)))))
5
6 (defr (move-in r i j)
7   (if (= i j)
8       r
9       (let
10        (def swap-idx (if (< i j) (+ i 1) (- i 1)))
11        (def moved (move-in r swap-idx j))
12        (rewire-in moved i swap-idx))))
13
14 (defr (snake-on r out-idx in-idx)
15   (snake (move-in (move-out r out-idx 1) 2 in-idx)))
```

Listing 6. New weaving reactors can be constructed using the existing ones. These reactors will be used in Section 5 to construct reactive sorting networks.

4.3 Towards Richer Weaving Technology

Sorting networks are usually defined inductively. For example, a bubble sorting network of size n+1 can be created by extending a bubble sorting network of size n with an additional stack of comparators (see Figure 1). To enable such combinations of reactors, Haai features a restricted form of recursion.

To explain the basic idea, let us consider the "Hello, World!" of recursion: the factorial. Consider the definition of the `factorial` reactor in Listing 5. This reactor recursively deploys itself on Line 4. Deployments of the `factorial` reactor can only occur when the depth of the recursion (i.e. the number of recursive deployments of `factorial`) is known at deployment-time. For example, there is no problem when the `factorial` reactor is deployed by the deployment expression `(factorial 5)`. However, when the recursion depth cannot be determined at deployment-time (e.g., in the program `(factorial time)`), the deployment phase fails.

Using recursion, it is possible to construct more complex weaving operators. For example, by combining recursive deployments with `parallel`, a reactor called `parallel-n` can be implemented. When given a reactor `r`, and a number `n`, `parallel-n` constructs a new reactor where `n` copies of reactor `r` are placed in parallel. Its implementation is shown in Listing 6, together with a number of weaving operators that will be used in the implementations of the sorting networks in Section 5. We briefly summarise the other reactors as well. `move-in` (resp. `move-out`[2]) is used to move a specific source

[2]The implementation of `move-out` has been omitted from Listing 6 since it is similar in structure to `move-in`.

(resp. sink) node to a new location, by recursively swapping the source (resp. sink) node with one of its neighbours until it has been placed on the correct location. `snake-on` then uses `move-in` (and `move-out`) to connect a specific sink node to a specific source node (given two indices) by using the `snake` primitive weaving operator.

In the rest of the paper we use the current set of 8 built-in weaving operators, together with the limited form of recursion, to build sorting networks. We make no claims about whether or not this combination of weaving operators and limited recursion suffices to make any reactor that one can imagine or draw. In other words, we do not answer the question as to whether or not every kind of weaving semantics (given the number of source and sink nodes a description of how the signal nodes are connected) can be implemented using the current state of the language. More theoretically, the question does give rise to the notion of *reactive completeness*: whether or not any reactor can be built given a particular set of weaving operators and a particular composition technique.

5 Reactive Sorting with Reactors

In this section we explain how reactors in Haai can *generate* sorting networks, which are also reactors. The sorting networks themselves are generated by higher-order reactors that have two source nodes: one that determines the size of the sorting network, and another one that determines the order in which data has to be sorted via a two-wire comparator reactor (`cae` from Listing 2 is an example of such a reactor).

5.1 Bubble Sorting Network

Consider the code in Listing 7 that generates a bubble sorting network as previously introduced in Section 2. It contains two reactor definitions: `bubble-one` which produces a stack of comparators such that the largest value bubbles to the topmost wire (which corresponds to the inner loop in Listing 1) and `bubble` which uses `bubble-one` to generate the complete bubble sorting network (which corresponds to the outer loop in Listing 1).

`bubble-one`. The `bubble-one` reactor uses recursion to construct a stack of comparators of a given size n. In other words, its output is a new reactor with n sources and n sinks where the largest value provided on the sources is "bubbled" to the topmost sink. For the base case of the recursion, we rely on the fact that when n=2, the sorting network is equivalent to a single two-wire comparator (Lines 2–3). When n>2, the `bubble-one` reactor is recursively deployed on $n-1$ to construct a smaller stack of comparators (which has $n-1$ sources and $n-1$ sinks). This smaller stack is then extended with another two-wire comparator, connecting the second sink of the comparator (carrying the largest value of the first two inputs) to the first source of the recursively-generated stack of comparators using `snake-on` from Listing 6.

```
1 (defr (bubble-one n cmp)
2   (if (= n 2)
3       cmp
4       (let
5         (def p (parallel cmp (bubble-one (- n 1) cmp)))
6         (snake-on p 2 3))))
7
8 (defr (bubble n cmp)
9   (def stack (bubble-one n cmp))
10  (if (= n 2)
11      stack
12      (let
13        (def small-network (bubble (- n 1) cmp))
14        (ror stack
15             (parallel small-network
16                       identity)))))
```

Listing 7. Implementation of `bubble`: which generates a bubble sorting network reactor of a given size.

Note that in the definition of `bubble-one`, the `let` syntax is used, which is a syntactic construct used for defining signals in a new (deployment-time) environment. Unlike the `let` syntax in Scheme [16], the `let` syntax in Haai does not have a list of binding expressions. The `def` syntax can be used in the body of the `let` to define new signals as used in the definition of `bubble-one`.

`bubble`. The `bubble` reactor generates the sorting network of a given size n. First, a stack of comparators is constructed using `bubble-one` which bubbles the largest value to the topmost sink (Line 9). When n=2, the desired bubble sorting network corresponds to this stack (which contains, as explained in the description of `bubble-one`, a single comparator). When n>2, a smaller bubble sorting network is generated first (Line 13) using a recursive deployment of `bubble`. This smaller sorting network is then combined with the `identity` reactor using the `parallel` weaving operator (Line 15). This adds an additional source and sink node to the smaller sorting network as the smaller sorting network only has $n-1$ source nodes and $n-1$ sink nodes. This combined reactor is then combined with the stack of comparators using the `ror` weaving operator (Line 14).

To use the sorting networks generated by `bubble`, it has to be deployed on the correct number of source signals. The code in Listing 8 shows how `bubble` from Listing 7 can be used. On Line 1, a bubble sorting network of size 6 is generated using the `cae` reactor as the two-wire comparator. Lines 2–3 deploy this reactor on 6 source signals (`i1` to `i6`) and the sink signals produced by the deployment of the sorting network are stored in the variables `o1` to `o6`. These signals will carry a sorted permutation of the values carried by the source signals.

5.2 Bitonic Sorting Network

The code in Listing 9 contains a Haai program that generates bitonic sorting networks [1] of size n (where n is required to be a power of two). The principle of divide and conquer is

```
(def the-sorting-network (bubble 6 cae))
(def (o1 o2 o3 o4 o5 o6)
  (the-sorting-network i1 i2 i3 i4 i5 i6))
```

Listing 8. Code example that first generates the bubble sorting network using `bubble` from Listing 7, and then deploys the network on 6 source signals.

used to construct a bitonic sorting network. The n input signals are split in two equal halves and each half is connected to a recursively-created sorting network. However, one half is sorted in the reversed order. This creates, what we call, a *bitonic signal sequence*: a sequence of signals which carry, during every turn (of the reactive program), a bitonic sequence. Then the bitonic signal sequence itself is sorted by a specialised network that merges one bitonic signal sequence in two smaller bitonic signal sequences while shifting the larger values upwards, and the smaller values downwards. By recursively applying this operation on each bitonic signal sequence until n=2; the complete sorting network is constructed.

The Haai code is best appreciated when read with the visualisation of the sorting network in mind. A visualisation of a bitonic sorting network of size 8 is shown in Figure 4. Each box represents one part of the network, and the colours differentiate between the different types of boxes. We will now explain how Listing 9 implements a reactor that generates these bitonic sorting networks for arbitrary sizes.

`bitonic`. The `bitonic` reactor generates the complete bitonic sorting network. When n=2, the result is equal to the two-wire comparator. When n>2 the bitonic sorting network is constructed by first constructing two recursively-defined sorting networks (of size $n/2$) using `bitonic` to create two sorted sequences of signals. One of these networks sorts the wires in the opposite order (by modifying the comparator reactor using `flip`; see below). Both recursive sorting networks are then combined using `parallel` to construct a reactor that has n inputs and n outputs in total. The complete bitonic sorting network is then created by adding a merging network (generated by `merge`) to this network using `ror`.

`flip`. The `flip` reactor swaps the two sink nodes of a given two wire-comparator.

`weave`. The `weave` reactor constructs the red boxes of Figure 4. Each red box of the same size has exactly the same structure, but depending on the location in the sorting network, the sinks of the two-wire comparators might be swapped. A weave of size n consists of $n/2$ comparators which are placed on every i^{th} and $i{+}n/2^{\text{th}}$ wire for every $i \in [1, n/2]$. This reactor is constructed using iteration. First, an empty n-wire reactor is constructed by placing n copies of the `identity` reactor in parallel. Then, using the `do` syntax (which has same syntactic form as in Scheme [16]),

```
(defr (bitonic n cmp)
  (if (= n 2)
      cmp
      (let
        (def pmc (flip cmp))
        (ror (parallel (bitonic (/ n 2) pmc)
                       (bitonic (/ n 2) cmp))
             (merge n cmp)))))

(defr (flip cmp)
  (rewire-out cmp 1 2))

(defr (weave n cmp)
  (def r-init (parallel-n identity n))
  (def k (/ n 2))
  (do ((r r-init (post-weave r cmp i (+ i k)))
       (i 1 (+ i 1)))
      ((> i k) r)))

(defr (merge n cmp)
  (def weavings (weave n cmp))
  (if (= n 2)
      weavings
      (let
        (def post-merge (merge (/ n 2) cmp))
        (ror weavings
             (parallel post-merge post-merge)))))
```

Listing 9. Implementation of `bitonic`: which generates a bitonic sorting network of a given size.

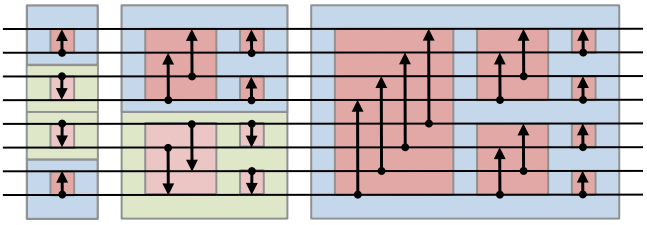


Figure 4. Bitonic sorting network of size 8. Adapted from https://commons.wikimedia.org/wiki/File:BitonicSort1.svg (original image provided under CC0 1.0 Universal Public Domain Dedication).

new comparators are iteratively appended to the network. The `do` syntactic form is implemented as syntactic sugar for recursion, and thus the same restrictions that apply on recursion also apply on iteration (i.e. the number of iterative steps cannot depend on a time-varying signal, see Section 4.3).

`merge`. The `merge` reactor uses `weave` to split an incoming bitonic signal sequence into two bitonic signal sequences which are then sorted using two copies of a recursively-generated merging network of size $n/2$. By repeating this until n=2, the original bitonic sequence is sorted (the reasoning for this is explained in [1]).

5.3 Pairwise Sorting Network

An implementation of a reactor that constructs pairwise sorting networks [23] is shown in Listing 10. The pairwise

sorting network of size n (just as for bitonic sorting networks, n is required to be a power of two) is constructed in three phases. First, all consecutive pairs of input wires are internally sorted (via a network constructed by placing $n/2$ comparators in parallel). Then, all odd wires and even wires are recursively sorted by a recursively-created sorting network. This results in a sequence of sorted pairs, which is then merged by another network of comparators. As before, we explain each reactor in Listing 10 in more detail.

odd-even-sort. The `odd-even-sort` reactor constructs a reactor where all odd wires (and all even wires) are sorted by a recursive sorting network of size $n/2$. The recursive sorting network is constructed first, and then two copies are placed in parallel. Then, the sources and sinks of this network are rewired such that all odd wires are connected to the first copy, and all even wires connected to the second copy. The sources are rewired first. The order in which `loop-in` rewires the source nodes does not matter, as long as all the odd sources are connected to one network (e.g., the bottom network), and all the even sources are connected to the other network. Thus, only the odd signal nodes that are connected to the even network need to be swapped with the even nodes that are connected to the odd network. This takes $n/4$ steps as half of the sources nodes are already connected to the correct network, and each `rewire-in` swaps two source nodes to their correct position.

The loop to connect the sink nodes is slightly more complex, as the order in which the sink nodes are rewired does matter for the remainder of the algorithm. The sink nodes are rewired from left-to-right, starting by moving the first sink node of the second network (connected to the even source nodes) to the second position. The next step of the iteration moves the next sink signal of the even network to the correct position. At every step of the iteration, one sink node of the first (i.e. odd) network is skipped, thus they do not need to be rewired explicitly. The `loop-out` reactor takes $n/2$ steps to execute in total.

Note the use of `n` in the body of `loop-in` and `loop-out` which shows that reactors are lexicographically scoped. The free variable `n` (in both `move-in` and `move-out`) has its binding location in the `odd-even-sort` reactor.

ssp. The `ssp` ("sort sorted pairs") reactor iteratively adds two-wire comparators to the sorting network according to the algorithm described in [23, Section 3]. Just like `odd-even-sort`, inline reactor definitions are used in the body of `ssp`.

pairwise. The `pairwise` reactor constructs the complete pairwise sorting network. Unless when n=2 (which is once again equal to the two-wire comparator), $n/2$ copies of the given comparator are first placed in parallel (the first phase of the algorithm), then the even wires and the odd wires are independently sorted by the network generated by `odd-even-sort` (the second phase of the algorithm), and then these sorted pairs are sorted using `ssp` (the third phase of the algorithm). All three subnetworks are placed after each other (i.e. in sequence) using `ror`.

```
(defr (odd-even-sort n cmp)
  (defr (loop-in r i j)
    (if (> i (/ n 2))
        r
        (loop-in (rewire-in r i j) (+ i 2) (+ j 2))))
  (defr (loop-out r i j)
    (if (> i n)
        r
        (loop-out (move-out r i j) (+ i 1) (+ j 2))))
  (def half-network (pairwise (/ n 2) cmp))
  (def r (parallel half-network half-network))
  (loop-out (loop-in r 2 (+ (/ n 2) 1))
            (+ (/ n 2) 1)
            2))

(defr (ssp n cmp)
  (def start (parallel-n identity n))
  (def k (/ n 2))
  (defr (outerloop r m)
    (defr (innerloop r i)
      (if (> i (- k (/ m 2)))
          r
          (innerloop
            (post-weave r
                        cmp
                        (* i 2)
                        (- (* (+ i (/ m 2)) 2) 1))
            (+ i 1))))
    (if (= m 1)
        r
        (outerloop (innerloop r 1) (/ m 2))))
  (outerloop start k))

(defr (pairwise n cmp)
  (if (= n 2)
      cmp
      (ror (ror (parallel-n cmp (/ n 2))
                (odd-even-sort n cmp))
           (ssp n cmp))))
```

Listing 10. Implementation of `pairwise`: which generates a pairwise sorting network of a given size.

5.4 Additional Sorting Networks

We have implemented two other reactive sorting networks of the traditional canon of sorting networks. Due to space constraints, we were not able to include their implementations in this paper. Their implementations can be found in the supplementary material of this paper [21, Sections B–C]. We briefly summarise both networks:

Insertion sorting network. In contrast to one's intuition, an insertion sorting network [17, Section 5.3.4] is identical to the bubble sorting network of the same size. Its implementation highlights some of the intricacies of the weaving operators.

Batcher's odd-even mergesort sorting network. This sorting network is similar to bitonic sorting networks (in

size and complexity) [1] but a different pattern is used to merge two sorted subsequences. In its implementation an alternative definition of `weave` (as also used in Listing 9) is included which does not use the `do` syntactic form.

6 Discussion

6.1 Evaluation

In Section 2 we outlined two problems that are present in today's reactive programming languages to implement sorting networks: (1) that programmers using a reactive programming language often need to deal with two distinct programming language semantics (which can lead to undesirable run-time behaviour), and (2) that the code of sorting networks built in these languages does not directly correspond to the description of the sorting network. We revisit these two problems and describe how they are not present in programs written in Haai.

1. As explained in Section 3, Haai is a manifestation of the "reactors all the way down" philosophy. The only "unit of code" in Haai is that of a reactor. There are no functions or procedures that can be applied. Several primitive reactors are present in the language which create new time-varying signals out of existing ones. In essence, the Haai programming language itself is just a wiring language where reactors are combined to create new reactors. The precise behaviour of a Haai program depends, mainly, on the functionality provided by these built-in reactors. We have shown that given a particular set of weaving operators (implemented as built-in reactors), it is possible to construct sorting networks in Haai, without the need for an active (sub)language.
2. In our opinion, weaving operators are an intuitive approach to generate (reactive) sorting networks. Starting from the Jacquard Diagrams of the weaving operators, it is straightforward to visualise how a reactive program can construct a sorting network using the weaving operators. Compared to signal composition in an embedded reactive programming language, where an active programming language is used to manipulate signals to create an ad-hoc instantiation of a sorting network, our approach is centered on the structure of the dependency graph itself. Indeed, at no point in the execution of Listing 1 is an actual sorting network (as a value) constructed: it is during the execution of the function that, as a side effect of being executed, a dependency graph is being constructed that sorts incoming data. The reactive runtime itself is in fact oblivious to the ordering of these signals. Exactly this information is captured by reactors. This makes it, in our opinion, more intuitive to reason about code written in Haai as the correspondence between Haai and the dependency graph (and thus, the structure of the sorting network) is more immediate.

While our solution in Haai using reactors solves both problems from Section 2, embedded reactive programming languages can also benefit from having the notion of a reactor as an abstraction for reusable (parts of the) dependency graph(s). Though, our approach does show that, in a pure reactive programming language like Haai, reactors themselves are powerful enough to build sorting networks (by using the weaving operators).

6.2 Incremental Reactive Sorting

Reactive sorting networks have one major advantage compared to a traditional implementation in an active programming language: incremental updates. Whenever one of the source signals of a reactive sorting network changes, only that single change has to be propagated through the dependency graph. Comparators that are not affected by the change do not need to be re-evaluated. The computational complexity of these incremental updates depends on the structure of the sorting network in question.

Remember from Sections 4.2 and 4.3 that weaving operators and recursive deployments cannot make use of time-varying signals. Thus, in contrast to the fact that our sorting networks are able to incrementally react to changes of the source signals, they are, actually, not reactive on the number of the signals to sort. This means that once a sorting network has been constructed, its size cannot change at a later point during the program's execution. A fully reactive sorting network also needs to be able to shrink and grow, depending on the number of actual source signals to sort.

If the size of the sorting network would be able to change at run-time, our language would be, in a way, similar to the Rete algorithm [13], which is a pattern matching algorithm where not only the facts (the data) can change dynamically (i.e. incrementally), but the rules (the program) can too. One seemingly innocent modification, which would make it possible to adjust the size of the network at propagation-time, is to allow the depth of the recursive deployments to be unrestricted during the deployment phase, along with removing the restrictions on some of our weaving operators. However, by doing so, (recursive) deployments should be allowed during the propagation phase, which means that the dependency graph can grow dynamically (i.e. while the program is running). To accommodate this, Haai programs would no longer be strongly reactive (as discussed in Section 3.3).

6.3 Beyond Sorting Networks

Our focus in this paper was to use reactor composition to express the structures of sorting networks. While sorting networks have given rise to the design of a particular set of eight weaving operators, we are actively interested in extending this set of weaving operators. We have identified two types of "networks", each interesting in their own way, that can give rise to new kinds of weaving operators whose behaviour might not yet be supported due to the current limitations on recursion, or the absence of a particular weaving operation.

Software-Defined Networking. Software-defined networking is a software-based approach for managing network infrastructure. An interesting research avenue is how Haai can be used to implement various types of network topologies, and how reactors can be used to alter the behaviour of a network infrastructure which can have entirely different structures compared to sorting networks. Interestingly, reactive programming has already been used in the field of Software Defined Networking [14, 29].

Neural Networks. Neural networks [19] are also an interesting type of network that may be suitable to be expressed as reactors. Similarly to sorting networks, a neural network is constructed given a number of parameters that determine the structure of the neural network (e.g., the number of hidden layers, the number of nodes per layer…). However, while a sorting network can only be used to sort incoming data, the same neural network should be able to be used both to learn and to make predictions. Thus, an interesting research avenue is to investigate how reactors can also generate neural networks.

By analysing different kinds of networks, we aim to establish a definitive set of weaving operators that make it possible to express more types of networks in our programming language. This should aid us in our quest towards reactive completeness.

7 Related Work

In earlier sections, we have already compared our approach to REScala [27]. The same issues that occur in REScala, as described in Section 2, are also present in other embedded reactive programming languages like Frtime [5] and Elm [6]. However, other embedded programming languages offer other alternatives to signal composition. For example, in Yampa [15] (an FRP language embedded in Haskell), signal functions are proposed as an alternative to signal composition[3]. Signal functions have similar characteristics to Haai's reactors, and just like in Haai, signals in Yampa are not first-class values. However, signal functions are not as powerful to reactors. First, signal functions are not as easy to compose as reactors. While a number of operators exist to compose signal functions, some of the weaving operators that we used in our implementation of the sorting networks are, to the best of our knowledge, not trivial to implement for signal functions, either due to restrictions of the type system, or due to the fact that signal functions themselves are treated as a black box (which would disallow internal rewirings like performed by `snake`). In addition, we point out that the arrow syntax is quite limited as it requires every intermediate signal to be named, an issue that is not present in Haai.

In the Emfrp [28] language, reactive programs are written as modules that are similar to reactors in Haai. However, unlike reactors, modules in Emfrp are not first-class and modules cannot create new modules the same way as how reactors can be created by a weaving operator in Haai. In addition, Emfrp has no support for recursion, which we deem as a crucial property for making (reactive) sorting networks.

Lava [2] is a high-level programming language used to describe the circuits on FPGA (Field Programmable Gate Arrays) chips. Lava is embedded in Haskell, and has also been used to implement sorting networks [4]. Similar to this paper, Lava has a number of built-in operators that are used to create new circuits by composing existing circuits. However, besides composing the behaviours of these circuits, Lava also has to compose the circuits themselves (using a grid system). Some of Lava's operators are specially crafted to optimise the placings of the on-chip circuits. As Lava is not a reactive language, comparing its reactive features would be unfair. However, one noteworthy difference that we would like to point out is that our weaving operators can also be used on time-varying reactor signals, something that we deem as an unlikely feature in languages for FPGAs.

8 Conclusion

In this paper, we have introduced the notion of reactive sorting networks, which are reactive programs that incrementally sort data arriving from time-varying data sources. We have implemented five reactive sorting networks in a language called Haai. A crucial property of Haai is that Haai programs are run in two phases: a deployment phase and a propagation phase. The deployment phase constructs the dependency graph of the reactive program, and the propagation phase propagates updates of primitive signals using this dependency graph. Reactors are first-class values, a property that we used to provide a number of primitive operators (called weaving operators) that can create new reactors by weaving the source and sink nodes of existing reactors in a particular way. We used these weaving operators in our implementations of the sorting network-generating reactors to let reactive programs themselves generate reactive sorting networks. To aid in understanding the behaviour of these weaving operators, we visualised their inner workings via Jacquard Diagrams.

In short, our approach shows that reactors are a useful abstraction mechanism for constructing (reusable parts of) sorting networks, or even dependency graphs in general. And, assuming that the right set of weaving operators is available as reactors themselves, a reactive programming language actually has no need for an active (sub)language to construct the dependency graphs themselves.

Acknowledgments

Bjarno Oeyen and Sam Van den Vonder are funded by the Research Foundation - Flanders (FWO) under grant numbers 1S93820N1 and 1S95318N, respectively.

[3] The original motivation for signal functions was to use them as a means to avoid space and time leaks [18].

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] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, Matthias Felleisen, Paul Hudak, and Christian Queinnec (Eds.). ACM, 174–184. https://doi.org/10.1145/289423.289440

[3] Daniel Bundala, Michael Codish, Luís Cruz-Filipe, Peter Schneider-Kamp, and Jakub Závodný. 2017. Optimal-depth sorting networks. *J. Comput. Syst. Sci.* 84 (2017), 185–204. https://doi.org/10.1016/j.jcss.2016.09.004

[4] Koen Claessen, Mary Sheeran, and Satnam Singh. 2003. Using Lava to design and verify recursive and periodic sorters. *International Journal on Software Tools for Technology Transfer* 4, 3 (2003), 349–358. https://doi.org/10.1007/s10009-002-0089-y

[5] Gregory H. Cooper and Shriram Krishnamurthi. 2006. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Proceedings of the 15th European Conference on Programming Languages and Systems, ESOP'06* (Vienna, Austria, March 27-28, 2006) *(Lecture Notes in Computer Science, Vol. 3924)*, Peter Sestoft (Ed.). Springer-Verlag, Berlin, Heidelberg, 294–308. https://doi.org/10.1007/11693024_20

[6] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. *SIGPLAN Not.* 48, 6 (June 2013), 411–422. https://doi.org/10.1145/2499370.2462161

[7] Sam Van den Vonder, Thierry Renaux, Bjarno Oeyen, Joeri De Koster, and Wolfgang De Meuter. 2020. Tackling the Awkward Squad for Reactive Programming: The Actor Reactor Model. In *34th European Conference on Object-Oriented Programming, ECOOP 2020 (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. Accepted/In press.

[8] Bert Dobbelaere. 2020. Smallest and fastest sorting networks for a given number of inputs. http://web.archive.org/web/20200715102935/http://users.telenet.be/bertdobbelaere/SorterHunter/sorting_networks.html. Accessed: 2020-07-15.

[9] Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. 2018. Thread-safe reactive programming. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 107:1–107:30. https://doi.org/10.1145/3276477

[10] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman (Eds.). ACM, 263–273. https://doi.org/10.1145/258948.258973

[11] James Essinger. 2004. *Jacquard's web: how a hand-loom led to the birth of the information age.* Oxford University Press.

[12] Matthew Flatt and Robert Bruce Findler. 2014. The Racket Guide.

[13] Charles Forgy. 1982. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artif. Intell.* 19, 1 (1982), 17–37. https://doi.org/10.1016/0004-3702(82)90020-0

[14] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: a network programming language. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 279–291. https://doi.org/10.1145/2034773.2034812

[15] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2002. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures (Lecture Notes in Computer Science, Vol. 2638)*, Johan Jeuring and Simon L. Peyton Jones (Eds.). Springer, 159–187. https://doi.org/10.1007/978-3-540-44833-4_6

[16] Richard Kelsey, William D. Clinger, and Jonathan Rees. 1998. Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices* 33, 9 (1998), 26–76. https://doi.org/10.1145/290229.290234

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

[18] Hai Liu and Paul Hudak. 2007. Plugging a Space Leak with an Arrow. *Electron. Notes Theor. Comput. Sci.* 193 (2007), 29–45. https://doi.org/10.1016/j.entcs.2007.10.006

[19] Tom M. Mitchell. 1997. *Machine learning, International Edition.* McGraw-Hill. https://www.worldcat.org/oclc/61321007

[20] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. 2019. Distributed Reactive Programming for Reactive Distributed Systems. *Art Sci. Eng. Program.* 3, 3 (2019), 5. https://doi.org/10.22152/programming-journal.org/2019/3/5

[21] Bjarno Oeyen, Sam Van den Vonder, and Wolfgang De Meuter. 2020. Reactive Sorting Networks (Supplementary Material). https://doi.org/10.5281/zenodo.4139829

[22] Bjarno Oeyen, Humberto Rodríguez-Avila, Sam Van den Vonder, and Wolfgang De Meuter. 2018. Composable higher-order reactors as the basis for a live reactive programming environment. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, REBLS@SPLASH 2018* (Boston, MA, USA) *(REBLS@SPLASH 2018)*, Guido Salvaneschi, Wolfgang De Meuter, Patrick Eugster, Lukasz Ziarek, and Francisco Sant'Anna (Eds.). ACM, New York, NY, USA, 51–60. https://doi.org/10.1145/3281278.3281284

[23] Ian Parberry. 1992. The Pairwise Sorting Network. *Parallel Process. Lett.* 2 (1992), 205–211. https://doi.org/10.1142/S0129626492000337

[24] Behrooz Parhami. 2002. *Introduction to parallel processing : algorithms and architectures.* Kluwer Academic, New York.

[25] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. 2012. A Novel Sorting Algorithm for Many-core Architectures Based on Adaptive Bitonic Sort. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012.* IEEE Computer Society, 227–237. https://doi.org/10.1109/IPDPS.2012.30

[26] Matt Pharr and Randima Fernando. 2005. *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation.* Addison-Wesley Professional.

[27] 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

[28] Kensuke Sawada and Takuo Watanabe. 2016. Emfrp: a functional reactive programming language for small-scale embedded systems. In *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*, Lidia Fuentes, Don S. Batory, and Krzysztof Czarnecki (Eds.). ACM, 36–44. https://doi.org/10.1145/2892664.2892670

[29] Andreas Voellmy, Ashish Agarwal, and Paul Hudak. 2010. *Nettle: Functional reactive programming for openflow networks.* Technical Report.