Finding regularity: describing and analysing circuits that are not quite regular

Mary Sheeran

Chalmers University of Technology
ms@cs.chalmers.se

Abstract. We demonstrate some simple but powerful methods that ease the problem of describing and generating circuits that exhibit a degree of regularity, but are not as beautifully regular as the text-book examples. Our motivating example is not a circuit, but a piece of C code that is widely used in graphics applications. It is a sequence of compare-and-swap operations that computes the median of 25 inputs. We use the example to illustrate a set of circuit design methods that aid in the writing of sophisticated circuit generators.

1 Introduction

In arithmetic and digital signal processing, many algorithms are well understood, and result in efficient regular circuits. The functional approach to hardware design has proved particularly well-suited to the development of such circuits [3, 10]. Here, we continue to explore this theme; this paper is not about verification, but about design methods – a valid, if under-represented, topic of the Charme conference. We emphasise the *description* of circuits, as we feel that ease of describing the intended circuit is a key to design productivity. The methods presented here go beyond what can be done in VHDL or C, through the use of higher order functions and polymorphism, which are features of many functional programming languages. The examples shown use Lava, a hardware design system implemented as an embedded domain specific language in the functional programming language Haskell [2].

Batcher's classic odd even merge sorting algorithm illustrates the power and elegance of the combinator-based approach to describing complex networks:

```
oemerge :: Int -> ([a] -> [a]) -> [a] -> [a]
oemerge 1 s2 = s2
oemerge n s2 = ilv (oemerge (n-1) s2)  ->-  odds s2

oesort :: Int -> ([a] -> [a]) -> [a] -> [a]
oesort 0 s2  = id                          -- the identity function
oesort n s2  = two (oesort (n-1) s2)  ->-  oemerge n s2
```

Here, `ilv`, for *interleave*, is a combinator that applies the given function to the odd and even elements of a list of inputs, to produce the odd and even elements of the output list. So, the function `ilv reverse` applied to the list `[1..8]` gives

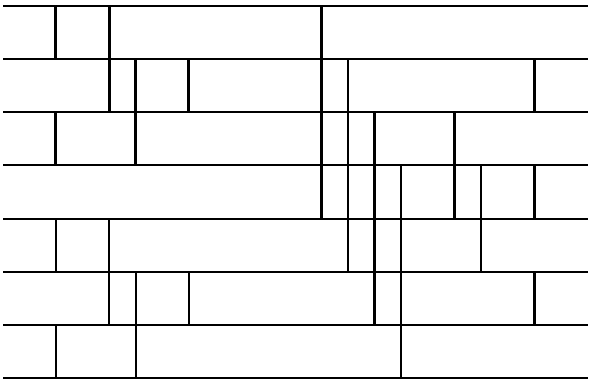

Fig. 1. `two (oesort 2 s2) ->- oemerge 3 s2`

`[7,8,5,6,3,4,1,2]`. `reverse` is a Haskell function whose type is `[a] -> [a]`. It takes a list of elements of any type `a` to a list of elements of the same type. It is a *polymorphic* function and works at many types. Similarly, `ilv` has type `([a] -> [b]) -> [a] -> [b]`. It takes a function from list of `a` to list of `b` and returns a function of the same type. In functional programming parlance, it is a *higher order function*; it takes a function and returns a function. We use polymorphic higher order functions like `ilv` to capture circuit interconnection patterns. A second such function is `two`, which applies a function to the first n elements and to the second n elements of a $2n$-length input list, so that, for instance, `two reverse [1..8]` is `[4,3,2,1,8,7,6,5]`. Serial composition is written `->-`, and `odds s2` applies `s2` to pairs of adjacent elements of the input, but starting with the second element rather than the first. The function `oesort` is parameterised both on an integer and on a two-input, two-output sorter component, `s2`. The integer and the `s2` parameter determine the size and type of the resulting network. For instance, `oesort 3 intSort2` is a circuit that sorts lists of integers of length 2^3, built from a component that sorts a 2-list of integers, `intSort2`.

```
intSort2 :: [Signal Int] -> [Signal Int]
intSort2 [x,y] = [imin (x,y), imax(x,y)]
```

To illustrate the combinators, `oesort 3 s2` is shown in figure 1. Values flow through the network from left to right, and the vertical lines are 2-sorters. The first (or leftmost) value of the input list is input along the top wire.

The `oesort` pattern can be instantiated with many different comparator components, depending on the context in which the sorter is to be placed. The same description can be used to give bit-parallel and bit-serial implementations, simply by plugging in new comparator components. The object of study is the connection pattern from which both combinational and sequential sorters can be built. To perform verification, we plug in a 2-sorter on bits (`bitSort2`) and, using the 0-1 principle [7], verify functional correctness by generating and checking a propositional formula that states that a fixed-size circuit obeys the required sorting property. The 0-1 principle states that if a network with n input lines

sorts all 2^n sequences of 0s and 1s into nondecreasing order, it will sort any arbitrary sequence of n numbers into nondecreasing order. We have studied the design and analysis of sorting networks in a previous paper [3], and we use the same verification methods in this paper.

The problem that we want to address here is the fact that not all circuits are beautiful. They don't all have a number of inputs that is a power of two, and they don't all have such an obvious recursive structure. For example, how would we describe any 7-sorter that contains the minimal number of comparators (which is known to be 16 [7])? More generally, how do we describe circuits that are somewhat regular?

Via a running example, a median circuit, we present a series of ideas for how to make more sophisticated circuit descriptions, using polymorphism and higher order functions. *Shadow values* and *clever components* are aids to writing circuit generators. *Non Standard Interpretation* is an old idea that we (and others) have used before. Here, we use ordinary polymorphism and components of different types, and do not rely on Haskell's type classes (although type classes are used extensively in the Lava implementation). Finally, we needed to extend our range of combinators in order to explore a variety of solutions to the median problem. We have deliberately not used the more esoteric parts of Haskell, in the hope of making the ideas usable in other contexts.

The median example was inspired not by a circuit, but by a piece of C-code, due to Paeth, which appears in Graphics Gems I, a book of classic graphics algorithms [11]. It is a sequence of 99 compare-swap operations that arranges an array of 25 inputs so that the median element is in the middle position, and all smaller elements are at lower indices (and hence all larger are at larger indices). I first came across a transliteration of this code in reference [5], where it is claimed (informally and without justification) that this function cannot be performed in fewer than 99 comparison-swaps without further information about the input. The application area of such programs (and circuits) is median filtering of digital images, in which n by n windows of the image have their middle pixel replaced by the median pixel, thus removing white noise. A 5 by 5 kernel (as it is called) is often used, so the algorithm is of practical interest. A common approach is to actually sort the 25 pixels, using Batcher's odd even merge sort, but in a more general variant that allows the division of the input into two parts of unequal length. That would take 138 comparators.

2 Shadow Values I

The user of Lava describes circuits by writing circuit *generators*. For example, in the `oesort` example above, the recursive description is instantiated at a particular size, and with a particular type of comparator, in order to produce a circuit. When we simulate, say, an 8-sorter on integers, what happens is that in the background a representation of the concrete circuit is created, and the `simulate` function walks over that representation:

```
simulate (oesort 3 intSort2) [3,2,1,6,5,4,0,7]
[0,1,2,3,4,5,6,7]
```

Here, the values that flow through the circuit are of type `Signal Int` and are *circuit level* values (even though they look like integers). The component `intSort2` sorts two such circuit level integers. However, the `3` that is a parameter to `oesort` is an ordinary Haskell integer. This is an important distinction, at least intuitively, as the Lava user must be able to tell what is a circuit description and what is a more general Haskell function. There are circuit level values (with `Signal` types), and there are ordinary Haskell values that are used in the generation of circuits. Once we have got to a concrete circuit in the internal netlist representation, all the ordinary Haskell values have disappeared. But in writing the Haskell code that is to be used to generate such a netlist, we can make use of ordinary Haskell values, and can make decisions about how the circuit should look, based upon them. A common pattern is to pair a Haskell value with each circuit level value. The *shadow values* can control the shape of the resulting circuit.

The simplest form of shadow value is just a boolean that indicates whether the corresponding wire should have any components attached to it. The Haskell function `tomarked f` applies `f` only to those inputs that are paired with `True`. It simply passes through those inputs that are paired with `False`.

```
Main> tomarked (map (*2)) [(1,True),(3,False),(5,True)]
[(2,True),(3,False),(10,True)]
```

Here, only the first and third values are doubled. We can use this idea when generating circuits. If `f` is a connection pattern that places instances of the component `s` in a particular way on n inputs, to give n outputs, we might want to get a circuit with $n - i$ inputs by deleting the top i wires and all components attached to them. The resulting circuit will take $n - i$ inputs. We pair each of those $n - i$ real inputs with `True`, and then add i dummy inputs paired with `False`. Then, we can apply `f (tomarked s)` to the resulting marked list, secure in the knowledge that the dummy wires will never be touched. Then, we can drop the dummy wires, and all the marks, to produce $n - i$ circuit level outputs. This is what the function `cutTop i` does. Similarly, `cutTopBottom i j` cuts `i` wires at the top and `j` on the bottom. Note that a component that is an argument to `tomarked` must be flexible, in that it may be required to deal with a number of arguments that is smaller than usual, because of the presence of inputs marked with `False`. In our sorting example, this means that we need a component that is not just a 2-sorter, but that can also deal with one or even zero inputs. The function `smallSort` takes a two-input sorter and makes it flexible in this way. We will have reason to extend this function later.

```
smallSort s2 []    = []
smallSort s2 [a]   = [a]
smallSort s2 [a,b] = s2 [a,b]
```

For example, we can make a 7-sorter from an 8-input odd even merge sorter by using `cutTop 1` and `(oesort 3)`. The resulting network is shown in figure 2. It is derived from the network shown in figure 1 by omitting the top wire, and the three comparators connected to it. In this instance, the resulting netlist has only 7 inputs and 7 outputs, and it no longer looks very regular. All

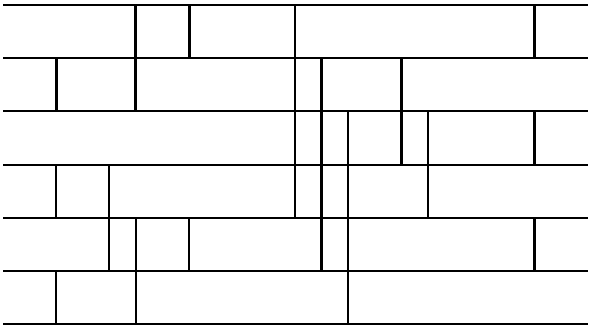

Fig. 2. `cutTop 1 (oesort 3) (smallSort s2)`

history of how that netlist was generated using shadow booleans is forgotten at this stage. The reader might argue that one could just use padded inputs and leave the pruning of unnecessary gates and wires to the lower level design tools. However, we find this approach more convenient and less error prone. We have found that padding makes for unreadable circuit descriptions, and can lead to the introduction of bugs. Also, we often make designs in which we first develop abstract circuits (say with integers whose representation has not yet been chosen flowing on wires). We want to be able to prune these circuits at an early stage in the design, before we are ready to produce input to lower level design tools.

Formal verification using a SAT-solver is done in the usual way [3]. (Satzoo is a SAT-solver developed by Een here at Chalmers [6]. The function `satzoo` creates a file in DIMACS format that is passed to the solver, the output of which is then passed back to the Haskell interpreter.)

```
sortCheck n cct =
  satzoo (prop_doesSortsize (cct (smallSort bitSort2)) n)

Main> sortCheck 7 (cutTop 1 (oesort 3))
Satzoo: ... (t=0.0) Valid.
```

Because we consider only restricted forms of networks, we choose not to prove that the networks permute their inputs. Such proofs, if required, can also be done using a SAT-solver in Lava.

3 Non-standard Interpretation

We have already seen how to verify sorting networks by using a 2-sorter on bits and the 0-1 principle. This is an example of non-standard interpretation, in which we replace the circuit components with others that are intended to gather information about the circuit. We then simulate the circuit with the new components, and suitable initialising inputs, to perform the required analysis.

To count the number of comparators in a circuit, we replace each comparator by a component that adds one to its left hand input and passes its right hand input through unchanged. Then, at the end, we sum all of the numbers appearing on the output. (This simple method works as long as all of the information-carrying wires eventually reach the output, but that is the case for all of our networks.) We simulate the resulting circuit on a list of zeros. Note that `csize2` is most definitely a circuit level component, whose inputs and outputs are lists of integer signals. It is included as a first step towards the use of such functions during circuit generation, rather than, as here, during simulation. A more general `count` function would be a recursive function over the internal data type representing circuits.

```
csize2 :: [Signal Int] -> [Signal Int]
csize2 [i,j] = [plus(1,i),j]

count n cct
  =  simulate(cct (smallSort csize2) ->- sum) (replicate n 0)

Main> count 7 (cutTop (oesort 3))
16
```

The 7-sorter has as few comparators as possible. Circuit depth is just as easy to calculate. Again, integers flow on the wires, and the depth of the output of a comparator is one more than the integer maximum of the inputs. The 7-sorter has optimal depth (which is 6)[7].

```
cdepth2 :: [Signal Int] -> [Signal Int]
cdepth2 [x,y] = [m,m]
  where m = plus(1,imax(x,y))

depth n cct
  =  simulate(cct (smallSort cdepth2) ->- imaximum) (replicate n 0)
```

Cutting 2 wires on the top of an 8-sorter also gives a size-optimal circuit with 12 comparators. We don't do so well when the number of inputs to the sorter is just above a power of two, rather than just below. The smallest known 9-sorters have 25 comparators, but cutting 7 wires from a 16-input odd even merge sort gives a 28-comparator sorter.

Our next step is to generalise the combinators `ilv` and `two` to be multi-way rather than two-way. This leads us to a generalisation of odd even merge sort, and also broadens the range of sorters and other networks that can be described easily.

4 Generalised combinators

Recall that `two f` applies `f` to each half of a list. Its generalisation, `parI i f` applies `f` to each ith part of the list, so that, for instance, `parI 5 f` applies `f` to each fifth of the list. The function `concat` flattens a list of lists back into a list. The general version of `ilv` instead chops the list into i-length sublists and transposes, to give i sublists, before applying `map f` and then returning the list to its original order.

```
parI i f  =  chopinto i ->- map f ->- concat

ilvI i f  =  chop i ->- transpose ->- map f ->- transpose ->- concat
```

Armed with these new combinators, we can generalise `oesort`, provided we can figure out what `odds` should become. Well, `odds s2` sorts an almost-sorted list. It is able to sort the list by comparing only adjacent elements, and it compares only those elements that have not already been compared. For $i = 3$, it turns out that the new pattern, which we will call `fmerge i`, should compare elements a distance two apart, and then adjacent elements, while refraining from comparing elements whose relation is already known. In general, `fmerge i` should first compare elements a distance $i-1$ apart, then $i-2$ and so on, down to 1. The function `dist i k ss` applies `ss` to elements a distance `k` apart, but avoids comparing elements in each `i`-length sublist.

```
fmerge i ss = compose [dist i k ss | k <- reverse [1..(i-1)]]

oemergeI i 1 ss = ss
oemergeI i n ss = ilvI i (oemergeI i (n-1) ss) ->- fmerge i ss

oesortI i 0 ss = id
oesortI i n ss = parI i (oesortI i (n-1) ss) ->- oemergeI i n ss
```

Think of the second parameter to `oesortI` as the number of *dimensions*. The instance `oesort i j` sorts a list of length `i` to the power of `j`. The `i` parameter, the size of each dimension, must be odd, although 2 works as a special case (and gives Batcher's odd even merge sort shown earlier). For larger even-length dimensions, some extra comparators are needed, but we will not pursue this topic here.

Now, if we are to use this general sorting algorithm for i greater than 2, we must be able to make sorting components (for use as the `ss` parameter) for more than two inputs. To do this, we extend the function `smallSort` that was introduced earlier. The 3-sorter is made from three comparators, and is completely standard. The 4- and 5-sorters are made from `oesort` (and are optimal in both size and depth). Larger sized sorters are easily included in a similar way, and it may then make sense to change the style of the definition to a case analysis on the length of the input.

```
sort3l s2 [x,y,z] = [a,b,c]
  where
   [x1,y1] =  s2 [x,y]
   [y2,c]  =  s2 [y1,z]
   [a,b]   =  s2 [x1,y2]

smallSort s2 []           =  []
smallSort s2 [a]          =  [a]
smallSort s2 [a,b]        =  s2 [a,b]
smallSort s2 [a,b,c]      =  sort3l s2 [a,b,c]
smallSort s2 [a,b,c,d]    =  oesort 2 s2 [a,b,c,d]
smallSort s2 [a,b,c,d,e]  =  cutTop 3 (oesort 3) (smallSort s2) [a,b,c,d,e]
```

If we restrict `oesortI` to two dimensions, we get the sorting algorithm proposed by Kolte et al [8] from Motorola. In that case, the rows and columns of the $i \times i$ grid are first sorted, and then the call of `fmerge i` sorts all the diagonal lines, starting with the main diagonals. What we add here is both a much more streamlined verification process and the generalisation to more than two dimensions. The paper by Kolte et al proposes an elaborate scheme for testing the proposed sorting network, but the use of a SAT-solver and the 0-1 principle is a much easier option. On the other hand, the Motorola paper develops software for a complete median filter that gives impressive performance on a particular architecture. It would be very interesting to develop an efficient median filter on an FPGA and compare its performance with more standard implementations. That is future work.

Using 3 dimensions, for example, we can quickly analyse a 27-sorter (made from 3- and 2-sorters) to find that it has depth 20 and size 154. This is one comparator smaller (though considerably deeper) than the general two-way odd even merge. We will make use of `oesortI 3 3` later, when constructing the 25-median circuit.

Further discussion of the algorithm `oesortI` is beyond the scope of this paper. We believe that `fmerge` could be improved for larger dimension sizes, and Van Voorhis' work shows how to deal with even-length dimensions [12]. Independent of the example, we are pleased with the simplicity of the generalised combinators. They give the user access to a broader range of connection patterns, without the need to learn many new combinators.

Now, we return to the 25-median problem. To solve it, we need to use more complicated shadow values than those that we have seen so far. We aim to keep only those parts of a sorter that contribute to arranging the outputs of the median circuit into an order that satisfies the specification.

5 Shadow Values II

We saw in section 3 that we can gather information about an instantiated circuit by simulating it using specially designed circuit level components like `csize2`. Here, we use similar ideas, but in the world of shadow values. Shadow values have so far been unchanging Boolean values. Now, we make them more dynamic and more complicated.

The idea is to use shadow values to record information about the circuit so far, allowing decisions to be made about how the rest of the circuit should look. For the median example, what we want to do is to figure out for each "wire" in the circuit whether or not it is still in the running to be the median, and so needs to be processed further. And we want to do this figuring out at circuit generation time. This is not straightforward, and requires some insights into the mathematics of sorting. We cannot go into the details here, but the reader is referred to the work of Van Voorhis to see the kinds of arguments that are required [12]. Our approach is to rewrite our sorter so that the first steps are to sort the different dimensions of the input. So, for example, a two-dimensional

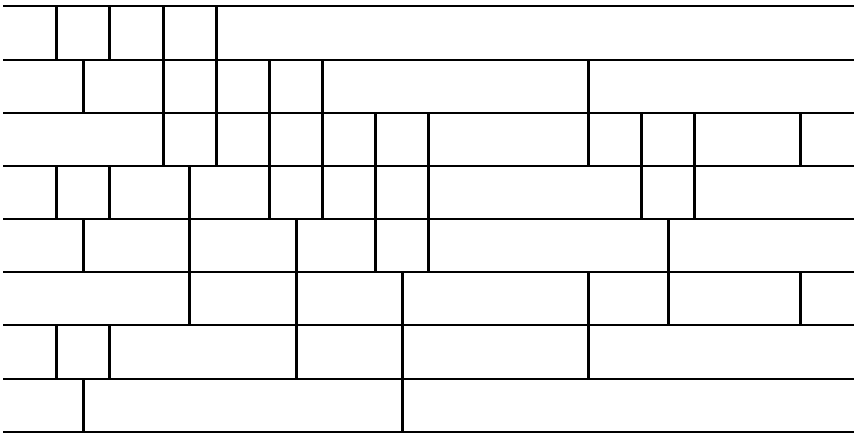

Fig. 3. `bflyI 3 2 (smallSort s2)` `->-` `fmerge 3 (smallSort s2)`

sorter will start by sorting the rows and columns, and a three-dimensional sorter will sort along each of the three axes. This pattern is called a butterfly network. It is straightforward to rewrite `oesortI` into a butterfly network of sorters followed by the rest, which we call `bafterI`. `boesortI 3 2` is shown in Figure 5. It is essentially the same as the optimal 25-comparator 9-sorter due to Floyd [7].

```
bflyI i 0 f = id
bflyI i n f = parI i (bflyI i (n-1) f) ->- (iter (n-1) (ilvI i) f)

boemergeI i 1 ss = id
boemergeI i n ss = ilvI i (boemergeI i (n-1) ss) ->- fmerge i ss

bafterI i 1 ss = id
bafterI i n ss = parI i (bafterI i (n-1) ss) ->- boemergeI i n ss

boesortI i n ss = bflyI i n ss ->- bafterI i n ss
```

The reason why we do this is that the sortedness of the different dimensions, which is the result of the initial butterfly network, remains unaffected throughout the rest of the network. Also, inside the butterfly, sorting each new dimension leaves the previously sorted dimensions still sorted. So, after the butterfly, it is easy to figure out, for a given wire, how many other wires are greater than or smaller than it. We give each wire an address that records what happened in the butterfly. So, for example, the address [2,1,2] is given to a wire that has "passed through" the top, bottom, and top of three 2-way comparators. After the butterfly, this wire is greater than or equal to the following set of wires: [[1,1,1],[1,1,2],[2,1,1],[2,1,2]]. Similarly, in the case of 27 inputs, the address [3,1,2] is less than or equal to the addresses [[3,3,3],[3,3,2],[3,2,3],[3,2,2],[3,1,3],[3,1,2]], after a butterfly of 3-sorters. Such calculations have been implemented in the functions `under` and `overI`. To calculate the list of addresses greater than a given one, one needs to know the size of the dimensions.

Now, inside `bafterI`, on each shadow wire, we keep lists of the addresses that are `over` and `under` it. The shadow component for the 2-sorter manipulates and updates these lists, which represent sets of addresses, and so do not contain duplicates. The standard function `nub` removes duplicates from a list.

```
combs2 :: [([Address],[Address])] -> [([Address],[Address])]
combs2 [(l1,g1),(l2,g2)] = [(nub (l1++l2),g1), (l2,nub(g1++g2))]
```

So, the wire that "passes through" the lower part of the comparator gets a new (`over`, `under`) pair containing the union of the two input `over` lists, but only the lower `under` list. For the upper wire, the situation is dual. Then, the *lengths* of these lists give good information about the status of a wire, and its relation to the remaining wires. On the input to the circuit, we provide information about the target for each wire. In our case, we place a single (shadow) integer on each wire, and the wire should be taken out of the running (in the same way as with the simple shadow Booleans that we saw earlier) once it is known to be either greater than or less than that number of other wires. The target remains unchanged, while the address lists grow longer as one moves through the network. (One could choose to use two integers for the target, which could be different for the `over` and `under` lists, but that is not necessary in the median examples shown here.) The new shadow component is `combine id combs2` where

```
combine f g [(a,x),(b,y)] = [(fa,gx),(fb,gy)]
  where
    [fa,fb] = f [a,b]
    [gx,gy] = g [x,y]
```

Each wire has a shadow value of type `(Int,([Address],[Address]))`, that is a pair of an integer and a pair of lists of addresses.

A wire is certain *not* to be the median if the number of distinct addresses that are either smaller than or greater than it is large enough. The target is set to $1 + \lfloor n/2 \rfloor$, where n is the number of inputs to the median circuit. Just after the butterfly, the address lists are all singletons containing the address of the wire to which they are attached. The function `placeTargetAddressI` introduces the required initial shadow values.

To be able to make use of these shadow values, we must generalise `tomarked`. The function `onPredicate p f` causes `f` to be applied only to those inputs for which the predicate `p` is true of the shadow value.

Recall that the version of `oesortI` with the butterfly in the first columns was

```
boesortI i n ss = bflyI i n ss ->- bafterI i n ss
```

Following this definition, we define

```
medI i j ss = bflyI i j ss ->-
              placeTargetAddressI i j ->-
              bafterI i j (onPredicate ok (smallSort comp)) ->-
              unmark
  where
    comp = combine ss (combine id combs2)
    ok   = not  . (notmedianI i)
```

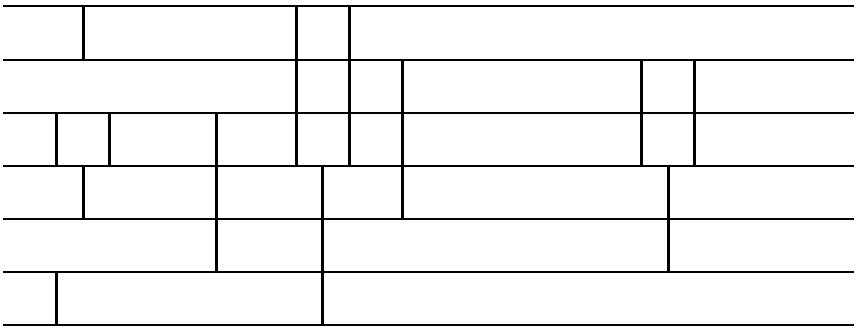

Fig. 4. `cutTopBottom 1 1 (medI 3 2) (smallSort s2)`

We leave the butterfly alone, but transform `bafterI` so that it performs the calculations described above when deciding whether or not to include a comparator. The result is promising:

```
Main> medCheck 27 (medI 3 3)
Satzoo: ... (t=0.3) Valid.

Main> count 27 (medI 3 3)
114

Main> count 27 (boesortI 3 3)
154
```

We have a circuit that correctly places the median input in the middle output, and all of the smaller values to the left of it in the output list. This property is checked by the observer `medCheck`, whose key function is `reallyMedian`, which checks that a given value is larger than all of the elements of a given list, and smaller than all of the elements of another. Logical implication (written `==>`) is the ordering on bits, and `andl` is a multi-input *and* gate.

```
reallyMedian a smaller bigger =
  andl ([s ==> a | s <- smaller] ++ [a ==> b | b <- bigger])
```

Again, we use the 0-1 principle, which applies also in the context of median networks; for a proof of this, see [9]. (It should be noted that the $O(log\ n)$ depth selection networks developed in reference [9] are far from being practical.)

We have saved 40 comparators in making a 27-median circuit from a 27-sorter. And the step to a 25-median circuit is now an easy one. We simply cut off the top and bottom wires, and attached comparators. Note that for making smaller median circuits from larger ones, it is necessary to crop the network symmetrically. To illustrate the step from a sorter to a median circuit, Figure 4 shows a 7-median circuit made from the 9-sorter shown in figure 3.

The 7-median circuit is optimal, but, sadly, that for 25 inputs has 102 comparators. And making the 25-median circuit directly (out of $2, 3, 4$ and 5-sorters), using `medI 5 2` takes 112 comparators, although 12 of them could be pruned

from the butterfly, which has so far been left untouched. Since we have pored over the Paeth code and discovered that it starts with a butterfly of 3-sorters that is missing its top and bottom wires, we choose to make a final change to the 102-comparator network. We use one last idea, *clever components* that adapt themselves to the context in which they find themselves in the final circuit.

6 Clever components

When a component is applied to inputs that have shadow values, then the definition of the component can decide what is to happen by looking at those shadow values. We have seen this several times. More interestingly, we can, in the definition, look to see what a particular arrangement of the basic components does to those shadow values. This is done simply by applying the proposed circuit to the inputs (which are mixed concrete and shadow values) and then looking only at the *resulting shadow values*. Then, the decision about what circuit to actually apply to the inputs can depend on those computed shadow values. This is a kind of "try it and see" approach, used during circuit generation.

To make the idea more concrete, let us return to the median circuit. Consider the case of a flexible 3-sorter, and our predicate (`notmedianI`) that indicates when an output is now done. If the 3-sorter is applied to 3 inputs (none of which is done), then it might be the case that two of the outputs become done. In that case, we don't need to know the order between the two done outputs, and we might as well use a 2-comparator *min* or *max* circuit of three inputs, as appropriate. So, think of a component that applies circuit A to the inputs, has a look at the shadow part of the result, and decides whether or not to be circuit B or to remain as circuit A, when producing the actual outputs of the component.

The definition of the clever version of `smallSort` starts off looking very much like that of `smallSort`, but with the addition of the predicate to the parameters of the function. When `smallSortV p ca` is applied to three inputs, `[in1,in2,in3]`, it computes `sort3l ca [in1,in2,in3]`, and names the resulting shadow values `a1`, `a2` and `a3`. By applying the predicate to those values, it can decide which of the `max3l`, `min3l` or `sort3l` patterns to actually use. Note that this is not just about calculating the cone of influence of the wires that are not done. Removing the comparator closest to the output of a 3-sorter can give either the maximum or the minimum circuit, but not both.

```
smallSortV p ca [in1,in2,in3]
   = if      (p a1) && (p a2) then max3l  ca [in1,in2,in3]
     else if (p a2) && (p a3) then min3l  ca [in1,in2,in3]
     else                          sort3l ca [in1,in2,in3]
       where
         [(_,a1),(_,a2),(_,a3)] = sort3l ca [in1,in2,in3]
```

If needed, `smallSortV` should be extended to longer inputs in a similar manner.

Our final median circuit generator, `medVI`, is identical to the previous `medI`, except that `smallSort` is replaced by `smallSortV (notmedianI i)`. And this does the trick! `cutTopBottom 1 1 (medVI 3 3)` has 98 comparators. We can use the same descriptions to generate median circuits of other sizes.

The modification of the sorter is most definitely a hack, though a rather effective one. Using another sorter, hand-crafted for the purpose, we have, in fact, been able to get the number of comparators down to 96, but we are unable to generalise that sorter to other sizes, and so chose not to present it here. The circuit development shown here allowed us to exemplify shadow values and clever components. However, for a circuit, as distinct from a C program, one should really aim for small depth rather than small number of comparators, so we have many more median circuits to explore. It would have been more pleasing to develop a recursive median circuit meeting our specification from scratch. But even when we have designed specific median circuits, we have found them difficult to express. They have an annoying lack of regularity, in that they tend to have fewer comparators in each phase as one approaches the outputs, but not according to any simple pattern. This is what led us to use clever components.

There are other ways to make median circuits, for example by looking at the inputs bit by bit [4], and indeed they may well be better. We have restricted our attention to comparator-based networks so far.

7 Related Work

Ideas similar to shadow values and clever components were used in the generation of the FM9001 netlist, as part of a large microprocessor verification effort [1]. Circuit generators (for example for the ALU) not only used precursors of both shadow values and clever components, but were also verified to meet their specifications for all sizes. This was done by a deep embedding of the `DUAL-EVAL` netlist language produced by the generators within the Boyer-Moore logic. The proofs required that the interpretation of the resulting netlists did indeed work correctly for all possible sets of inputs. This work, which is so closely in line with our aims, is barely mentioned in the published paper, which concentrates on the overall verification goal. Indeed, it is barely mentioned (in the English text) even in the very long technical report on the verification effort, so it will be necessary to delve into the code. I did not know about this work when writing the first version of this paper. My disappointment at discovering that my ideas are not as new as I thought has been overshadowed by the realisation that my tentative ideas for mixing clever components and verification have already been shown to work.

In the current version of Lava, we perform formal verification only of fixed size circuit instances. A first step towards making use of the FM9001 generator work would be to generate `DUAL-EVAL` code and to perform proofs about that code. However, we have long considered a move to using first order provers and inductive proofs of recursive circuits. Our emphasis on the use of higher order functions gives our circuit descriptions (and our use of shadow variables and clever components) a different style from those in the FM9001 work. The next step will be to find good ways to combine the best of both approaches.

8 Conclusion

We have presented a collection of methods that together allow us to describe and analyse circuits that are not quite regular. We distinguish *circuit generation time* from *circuit analysis time*, and there is a clear analogy with compile time and run time, and with static and dynamic semantics in VHDL. The aim of circuit generation is to produce a representation (in terms of a suitable recursive data type) of a complete fixed-size circuit, something very close to a netlist. Circuit analysis is what happens when we turn this representation into various other notations, in order to scrutinise it further, often with the help of external tools such as SAT-solvers and model checkers. Simulation is one such analysis.

During circuit generation, we use the power of Haskell to control the process of generating the required netlist. Special values called *shadow values* are associated with the circuit level values, and can be used to control the generation process. They can be static, like the shadow Booleans that we use when omitting unwanted parts of networks, or dynamic, like the address lists that we used to track progress towards a target in the median circuit example. The shadow values can also encode information about the circuit that feeds a component, allowing the component itself to decide what circuit would best be introduced into the network at that point. These *clever components* are likely to have many applications. For example, the "try it and see" could extend to calling external tools like, say, automatic place and route tools with possible circuits that might be included in the final design, and then picking the one that gives the best result according to some criterion such as timing, testability or power consumption. We have not yet incorporated the notion of layout into this work but that will be the next step. At Chalmers, we are developing a language that captures wires and layout explicitly, but uses Lava functions for describing circuit function. It can be seen as a generalisation of layout combinators [3], and we have had to move from 2-dimensional tiles to 3-dimensional blocks. We aim to be able to capture the ways in which regular circuits become irregular during the design process, for example when they are designed to fit under a particular interconnect fabric. Our intention is to combine the design methods illustrated here with circuit analyses that capture wire length and related non-functional properties. Thus, it is the problem of how to do interconnect-aware design that is the main motivation for this research.

However, there is a second motivation, the need to push formal verification earlier in the design process. We had speculated that clever components would allow sub-parts of circuits to be verified during circuit generation, but had not yet performed any experiments in this area. The FM9001 work shows, very convincingly, that these ideas enable both hierarchical proofs and the generation of circuits that are built for verifiability. That the FM9001 proof can simply be rerun for any size is an extremely important property of the verification effort. The use of verified circuit generators in the FM9001 work goes beyond what we had envisaged. We feel spurred on to investigate ways to support inductive proofs of recursive circuit generators based on Lava combinators, while still aiming for as much proof automation as possible.

Finally, we would like to investigate whether or not our methods, and in particular clever components, could be applied to the description and analysis of reconfigurable circuits.

Acknowledgements

Thanks to Satnam Singh, who suggested the median example. This research is funded by the Swedish funding agency Vetenskapsrådet. Thanks to the anonymous reviewers for their thoughtful reports. It was especially useful to learn of related work in the generation and verification of the FM9001 microprocessor.

References

1. B. C. Brock and W. A. Hunt, Jr.: The `DUAL-EVAL` Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor, *Formal Methods in System Design*, Vol. 11, Kluwer Academic Publishers, 1997.
2. K. Claessen and M. Sheeran: A Tutorial on Lava: A Hardware Description and Verification System, April 2000. http://www.cs.chalmers.se/∼koen/Lava/tutorial.ps
3. K. Claessen, M. Sheeran and S. Singh: The Design and Verification of a Sorter Core, Proc. Int. Conf. on Correct Hardware Design and Verification Methods, Margaria and Melham (eds.), Springer LNCS 2144, 2001.
4. P.E. Danielsson: Getting the median faster. *Computer Graphics and Image Processing*, Vol. 17, No. 1, 1981.
5. N. Devillard: Fast median search: an ANSI C implementation. http://ndevilla.free.fr/median/median/index.html
6. N. Een: Satzoo home page. http://www.cs.chalmers.se/∼een/Satzoo/
7. D. E. Knuth: *The Art of Computer Programming, Volume 3, Sorting and Searching, Second Edition*, Addison Wesley, 1998.
8. P. Kolte, R. Smith and W. Su: A Fast Median Filter Using AltiVec. In Proc. Int. Conf. on Computer Design, IEEE Press, 1999.
9. S. Jimbo and A. Maruoka: A Method of Constructing Selection Networks with O(log n) Depth SIAM Journal on Computing Volume 25, Number 4, 1996.
10. John O'Donnell: From transistors to computer architecture: teaching functional circuit specification in Hydra, in Proc. FPLE'95: Symposium on Functional Programming Languages in Education, Hartel and Plasmeijer (eds.), Springer LNCS 1022, 1995.
11. Alan W. Paeth: Median finding on a 3×3 grid. in *Graphics Gems I*, Glassner (ed.), Academic Press Professional, Inc., 1990.
12. D. C. Van Voorhis: A Generalization of the Divide-Sort-Merge Strategy for Sorting Networks. Technical Report STAN-CS-71-237, Stanford University, 1971.