Communicating Process Architectures 2014
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2014

N Queens on an FPGA: Mathematics, Programming, or Both?

Jan KUPER [1] and Rinse WESTER

Computer Architecture for Embedded Systems,
University of Twente,
Enschede, The Netherlands.

{J.Kuper , R.Wester} @utwente.nl

Abstract. This paper presents a design methodology for deriving an FPGA implementation directly from a mathematical specification, thus avoiding the switch in semantic perspective as is present in widely applied methods which include an imperative implementation as an intermediate step.

The first step in the method presented in this paper is to transform a mathematical specification into a Haskell program. The next step is to make repetition structures explicit by higher order functions, and after that rewrite the specification in the form of a Mealy Machine. Finally, adaptations have to be made in order to comply to the fixed nature of hardware. The result is then given to CλaSH, a compiler which generates synthesizable VHDL from the resulting Haskell code. An advantage of the approach presented here is that in all phases of the process the design can be directly simulated by executing the defining code in a standard Haskell environment.

To illustrate the design process, the N queens problem is chosen as a running example.

Introduction

Common practice in the design process of mapping an application onto an FPGA often includes various perspectives. Usually, a problem is given in a mathematical form, and is simulated in, for example, MATLAB. Then a first implementation is produced in a sequential, imperative language for which C or C++ are popular candidates. As a next step, this sequential code has to be transformed into VHDL or Verilog, in order to configure the FPGA. This last step may be supported by intermediate frameworks such as SystemC or other high-level synthesis tools (see, for example, [1] for an overview of several approaches).

In our view, the problem with such an approach is that the semantics of the various intermediate formulations differ substantially. First of all, translating a mathematical specification into C is non-trivial and introduces a sequentialization that is not present in the mathematical specification itself. Then, the step from C to VHDL causes the problem of discovering parallelism and concurrency in the C-code, which again is a non-trivial step. Consequently, such a design process is time consuming and error prone, no matter the high-level synthesis tools that are developed over the years for (partially) synthesizing code that is imperative in nature.

In this paper we present a methodology for mapping an application onto an FPGA, starting from a mathematical specification of the application, but without first transforming the

[1]Corresponding Author: Jan Kuper, Computer Architecture for Embedded Systems, University of Twente, Enschede, The Netherlands. E-mail: J.Kuper@utwente.nl.

mathematical specification into an implementation in an imperative language such as C. Instead, we chose Haskell as programming language for various reasons. First, it is close to mathematics which leads to the feature that translating the mathematical specification into Haskell is straightforward. Second, we use the CλaSH compiler [2], which translates a slight modification of the Haskell code into VHDL. Consequently, the design process remains in the same semantic domain, and does not switch perspective as is the case when moving from mathematics to C, and then from C to hardware. In fact, we feel that a mathematical specification in a specific format is closer to the structure of hardware than C. Thus, programming an FPGA coincides, to a large extent, to a mathematical derivation of a specification in a specific format.

There are more compilers which translate functional specifications of hardware into VHDL. These include Lava [3] and Bluespec [1]. However, most of these approaches are based on an embedded language within Haskell, whereas CλaSH uses Haskell *itself*. The advantage is that at every level in the design process, simulation and testing can be done by executing the corresponding specification in a standard Haskell environment.

As a running example to describe our design methodology we chose the well-known 8 queens problem, or, more generally, the N queens problem: how to place N queens on an $N \times N$ chessboard such that they do not threaten each other? We will focus on a method to find *all* solutions to this problem. This is a well known problem, studied from different prespectives, and there are implementations in many different programming languages [4]. The N queens problem includes both regular and non-regular algorithmic aspects, which makes it an interesting case study for the design methodology presented in this paper. Besides, the complexity is high, for example, the largest value of N for which all solutions are currently known is N=26: $2.23 \cdot 10^{16}$ solutions (to be more precise: 22,317,699,616,364,044 [5,6]), whereas the number of partial solutions investigated is a multitude of this. For this result, some 25 FPGA's of different types were used, requiring 10 months of computation.

As indicated above, our emphasis is not on improving the efficiency of existing mappings of the N queen problem on an FPGA, but to use the N queen problem as a running example for the presentations of a methodology for systematic FPGA design, starting from a mathematical specification and avoiding switches in the semantical domain. Earlier presentations of the methodology, focusing on transformation rules for higher order functions for a systematic trade-off between usage of space and time on an FPGA, can be found in [7,8].

In the remainder of this paper we will first give an overview of the design methodology (Section 1), after which we introduce the N queens problem and give a mathematical specification of it (Section 2). The mathematical definition is then transformed into a sequence of Haskell definitions (Section 3) which in turn lead to the formulation in terms of a Mealy Machine from which the actual hardware is generated using the CλaSH-compiler (Section 4). We conclude the paper with some results (Section 5) and some conclusions (Section 6).

1. Overview of the Design Methodology

In this section we give a short global overview of the steps to be performed in the derivation of an implementation on an FPGA starting from a mathematical specification. We will discuss transformations of the mathematical expressions at various stages of the design process. To introduce the role of these transformations, we give a short example: how to calculate a polynomial expression on an FPGA.

Let the polynomial function be given by

$$f_0(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$$

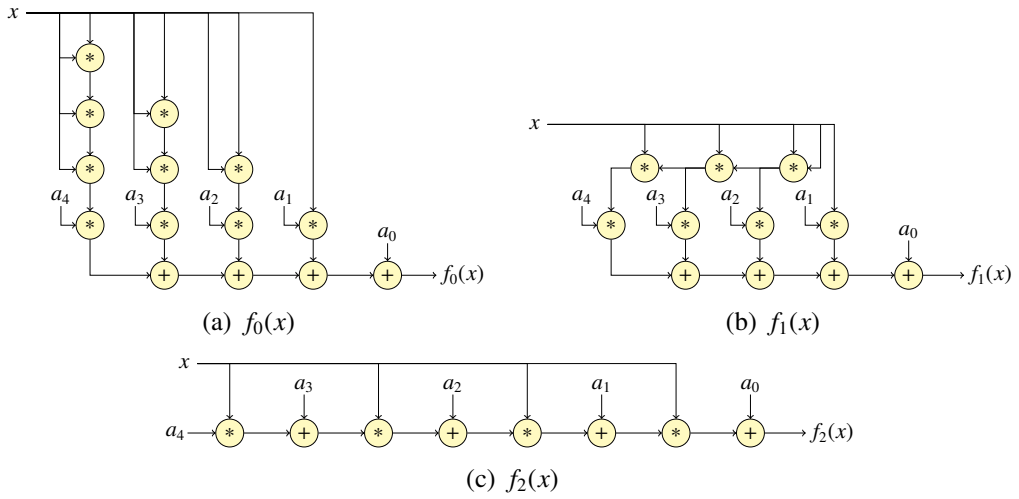


Figure 1. Architectures for a polynomial function.

In Figure 1(a) this polynomial is graphically represented: x^i is represented by a sequence of multiplications with x, starting from $x{\cdot}x$, and the result is multiplied with a_i. Then, the results for every i are added.

Figure 1(a) may also be read as an *architecture* on an FPGA which calculates the given polynomial. This architecture reflects the *structure* of the mathematical expression for the polynomial closely, in particular if we read exponentiation as repeated multiplication.

However, this architecture is rather inefficient since the outcomes of all terms x^i are calculated separately. Instead, we may calculate the corresponding values $x_i = x^i$ incrementally:

$$x_1 = x,\ x_2 = x{\cdot}x_1,\ x_3 = x{\cdot}x_2,\ x_4 = x{\cdot}x_3$$

and then define an improved version of the polynomial function:

$$f_1(x) = a_4x_4 + a_3x_3 + a_2x_2 + a_1x_1 + a_0$$

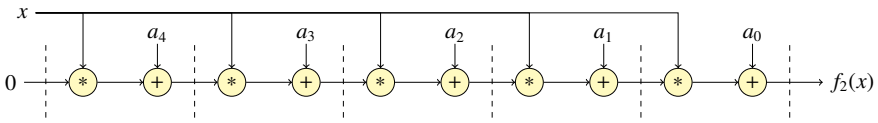
The function f_1 also specifies an architecture, given in Figure 1(b). This architecture is more efficient than the previous one, in the sense that fewer multipliers are required. We can still do better, by writing the polynomial as follows:

$$f_2(x) = (((a_4{\cdot}x + a_3){\cdot}x + a_2){\cdot}x + a_1){\cdot}x + a_0$$

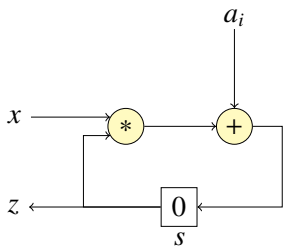
This function specifies the architecture in Figure 1(c).

We remark that the equivalence of f_1, f_2, and f_2 can be proven. Since the correspondence between the function definitions and the architectures is direct, this also means that the equivalence of the corresponding architectures is guaranteed. For now we abstract away from multi-cycle multipliers and adders, so we assume that all variants of the polynomial function shown so far are executed in a single clock cycle, that is, in all three cases the calculation of the polynomial is fully parallel.

In general, mathematical specifications will be more complex than the case of the polynomial above, and not every formula will correspond to an architecture directly. For example, a formula may contain expressions from continuous mathematics such as differentials or integrals, or there may be set theoretical objects included, or a function may be defined recursively. In case a formula does not correspond to architecture directly, there are still some possibilities to solve this. For continuous mathematics a discretization of the formula may be needed, whereas for a recursive function it may be helpful to make the pattern of recursion explicit by means of higher order functions. We will see examples of that below.



(a) $f_2(x)$ with time slots



(b) $f_t(x)$

Figure 2. Polynomial architecture over time.

Another situation in which a function may not be directly realizable as an architecture, is when the function is too complex, and the architecture becomes too big to fit on an FPGA. If the architecture has a regular structure, there is the possibility to take a specific part of the architecture and repeat that over time. That is to say, the architecture is sequentialized. For example, the architecture that realizes the function f_2 above may be too big. One may "cut through" the picture in Figure 1(c) by vertical lines as indicated in Figure 2(a), select one part, and repeat that over time. At places where such a cutting vertical line crosses an arrow, a memory element has to be introduced to remember the value on that arrow to the next clock cycle. That leads to the architecture given in Figure 2(b), defined as follows:

$$f_t\,(s, (x, a)) = (s{\cdot}x{+}a,\, s)$$

That is to say, the argument s indicates the *state* of the architecture, that is, the value contained in the memory cells. The argument (x, a) is the input, consisting of two values, where it is intended that x is the same input each clock cycle, whereas a will have the values a_4, a_3, etc.

In order to get a somewhat more regular pattern, Figure 2(a) is extended to the left with respect to Figure 1(c), exploiting the equality $a_4 = 0 * x + a_4$. The effect is that the starting value in the memory cell can be 0, as shown in Figure 2(b).

The result of the function f_t also consists of two components: $s{\cdot}x{+}a$ is the new content of the memory cell, and s is the output, called z in Figure 2(b). Note, that now every clock cycle a new value for z is produced. The definition of the function f_t is along the lines of a Mealy Machine, which will be discussed in Section 4.1.

Below we will generalize the process introduced above, describing several transformation steps which can be applied in a rather systematic manner.

1.1. Towards a general design methodology

If an application is defined by an expression in continuous mathematics, as for example with signal processing, or in computational physics, then first of all the expression has to be discretized in order to make it computable. The example in this paper, the N queens problem, is formulated in discrete mathematics already, so for that problem this is not an issue. Also, based on a discretized mathematical specification, observations on complexity can be made, and transformations of the specification can be applied in order to arrive at an expression which is more optimal from a computational perspective.

Often, The chosen mathematical expression can be translated in a word-for-word fashion into a first representation in Haskell. The resulting Haskell definition often contains program-

ming abstractions like list comprehension and recursion. In order to arrive at hardware, the constructions have to be removed and replaced by higher order functions.

The previous steps are strongly mathematical in nature, and the semantics of both the mathematical formulations and the Haskell formulations are close to each other. Thus, the equivalence of the various formulations can be shown. In contrast to the mathematical formulation, the Haskell definitions are executable, and provide a both a specification and a simulation of the functional behaviour of the architecture under design.

The next steps are different in nature and add space and time to the specification. That is to say, information on the question where and when the execution will take place is introduced. To that end the Haskell specifications will be reformulated in the form of a Mealy Machine whose functional body is specified by the Haskell definitions so far. This makes space and time explicit in the execution of the Haskell specification.

The reformulation as a Mealy Machine can be done in different ways, in particular, the higher order functions present in the Haskell definitions may be executed over space or over time, Thus, design decision have to be taken concerning the question which higher order functions should be executed over space, and which over time. That is to say, which will be executed in parallel, and which sequentially. In this paper we will restrict ourselves to choose for each higher other function to fully execute it over space, or fully over time. However, there are mixed patterns possible, examples of that can be found in [8,9], where systematic methods are discussed to partition the execution of higher order functions over space and time. Traditional techniques such as pipelining and retiming are also possible and can be expressed as variations in the Mealy machine specifications.

The last step in the design methodology discussed here, consists of some finalizing transformations to massage the resulting Haskell code into a suitable input for CλaSH, such that an FPGA configuration can be derived from it. For example, this step includes choices for bit widths of numbers, lengths of lists, etcetera, which are necessary since hardware is fixed. Since these choices are expressed in the types of expressions, the type system of Haskell will check the choices, and CλaSH can use the information for the generation of hardware.

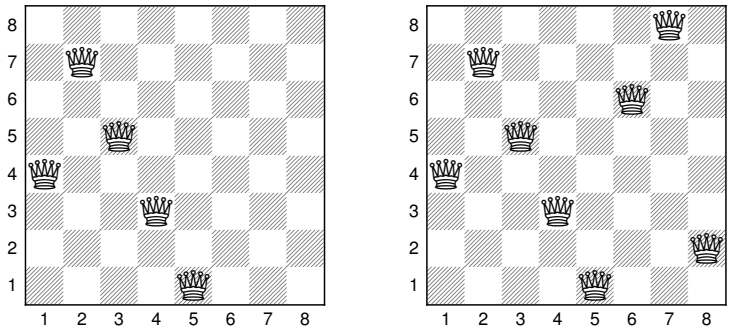
Remark on the notation. A major intention of this paper is that design steps are provably correct, assuming that the initial specification is given in a mathematical format. That means that the framework in which the design steps are performed should be amenable for formal reasoning and mathematical proof. At the same time we strive for testability at all stages of the design process, that is, at all stages the preliminary and intermediate specifications of the final architecture should be executable so that a designer can check whether they realize the intended behaviour.

As mentioned above, we choose Haskell as the design framework, since in principle it meets these requirements. However, to make Haskell's support for a mathematical way of reading even stronger, we will use symbols with a more mathematical flavour than (combinations of) the symbols available at a common keyboard. It is our intention that this notation stimulates the reader to read the specifications in the following sections rather as *definitions* of structures, than as computer *programs*. Nevertheless, the specifications are executable, and at the relevant places we will indicate what keyboard combinations should be chosen for the mathematical symbols.

Finally, we remark that the standard Haskell compiler GHC does recognize Unicode, so many mathematical symbols can be used directly in executable Haskell code.

2. Mathematical Definition of the N Queens Problem

We assume that in a (possibly partial and possibly incorrect) configuration of queens on a chessboard every column contains (at most) one queen, where the columns containing a



(a) $\langle 4, 7, 5, 3, 1 \rangle$ (b) $\langle 4, 7, 5, 3, 1, 6, 8, 2 \rangle$

Figure 3. Queen configurations.

queen are consecutive from the left. Hence, a configuration of queens is completely specified by a sequence of positions within the columns, denoted as $\mathbf{q} = \langle q_1, \ldots, q_n \rangle$ (see Figure 3). The empty sequence is denoted as $\langle\,\rangle$, and the set of all possible configurations of length N is denoted as $\mathbf{Q}_N$.

Assuming that there is only one queen in every column, then two queens in vertical positions p and q are *safe* (denoted as S) with respect to each other, if they are not in the same row, and the horizontal distance is not the same as the vertical distance:

$$S\,(p, q, d) \;\Leftrightarrow\; p{\neq}q \,\wedge\, |q{-}p|{\neq}d$$

Using the relation S, the initial definition of the set Q_N° of all correct configurations on an $N \times N$ chessboard is as follows:

$$Q_N^\circ = \{\, \mathbf{q} \mid \mathbf{q} \in \mathbf{Q}_N, \forall i{\neq}j\colon S\,(q_i, q_j, |\,j{-}i\,|)\} \tag{1}$$

Following this definition, the calculation of the set Q_N° requires checking N^N possible sequences $\mathbf{q}$. A well known, more efficient, definition is as follows: let $\mathbf{q}$ be a partial configuration of length n, and let p be a position in column $n{+}1$ (that is, immediately to the right of $\mathbf{q}$). Then a queen is *safe* (S') on position p with respect to configuration $\mathbf{q}$, if:

$$S'\,(\mathbf{q}, p) \;\Leftrightarrow\; \forall i{\in}\{1, \ldots, n\}\colon S\,(q_i, p, n{+}1{-}i)$$

Now let Q_n denote the set of all correct sequences up to and including column n (where numbering on a chessboard starts with 1), and let $\mathbf{q};p$ denote the concatenation of p to the right of sequence $\mathbf{q}$. Using S' we define the set Q_n recursively as follows:

$$\begin{aligned} Q_0 &= \{\,\langle\,\rangle\,\} \\ Q_n &= \{\,\mathbf{q};p \;\mid\; \mathbf{q} \in Q_{n-1},\quad p \in \{1, \ldots, N\},\quad S'\,(\mathbf{q}, p)\,\} \end{aligned} \tag{2}$$

Hence Q_N denotes the set of all acceptable configurations of N queens on a board of size $N \times N$. Without going into details, we mention that definitions (1) and (2) are equivalent, and the calculation of the set Q_N requires substantially fewer steps than the calculation of Q_N°.

For N=5, the generation of the sequences according to definition (2) is shown in Figure 4. The set Q_n consists of the paths of length n, starting from the root of the tree. Thus, Q_0 only contains the empty path starting at the root, whereas Q_5 contains 10 correct configurations, indicated by the 10 paths from the root to the leafs of the tree. Note that shorter paths are dead ends in the generation process and cannot be safely extended further.

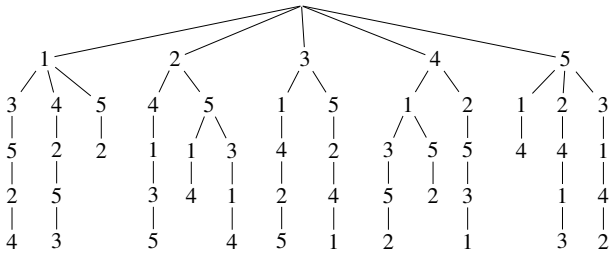


Figure 4. Generation tree for N=5.

3. Haskell

In this section we will discuss the implementation of the N queens problem in Haskell[1], starting from the definitions in section 2.

3.1. Implementations of the Safety Functions: safe, safeAll

We start with the implementation *safe* of the predicate S in Listing 1. The parameters p,

Listing 1 *safe*.

```
safe p q d = p ≠ q ∧ abs (p−q) ≠ d
```

q, d have the same meaning as before, though they are not written between brackets and separated by commas, but they are mentioned one after another and are separated by spaces, thus facilitating *partial application* (see below).

The formulation of the definition of *safe* is very close to the definition of S; the main difference being that *safe* is a *Boolean valued function*, whereas S is a *predicate*. However, that is a standard difference between a programming language and mathematics, making *safe* executable by a machine, where S is not.

Next, in Listing 2 the predicate S' is implemented as the Boolean valued function *safeAll* (writing *qs* instead of **q**). In this definition we exploit partial application: the function *safe* is

Listing 2 *safeAll*.

```
safeAll qs p = foldl (∧) True $ zipWith (safe p) qs ds
  where
    n  = length qs
    ds = [n,  n−1 .. 1]
```

applied only to p, meaning that *safe p* is a *function* which still expects its two other arguments q and d. Hence, *safe p* is a *binary* function. The parameter *qs* contains a sequence of q-values, and *ds* is the sequence of corresponding d-values, the distances of the q-values to p, defined in the **where**-clause. The higher order function *zipWith* "zips" the sequences *qs* and *ds* with a function. For example:

$$zipWith\ (+)\ [1,2,3]\ [1,2,3] \quad \Rightarrow \quad [2,4,6]$$

[1]The symbols "≡", "≢" and "∧" are smoothed forms of the Haskell operations "==", "/=" and "&&", respectively.

In the definition of *safeAll*, the zipping is done with the function *safe p*, leading to the sequence of Boolean expressions *safe p q d* for all corresponding values q and d.

Intuitively, the \$-sign may be read as a "pipeline" operation, the total expression on the right hand side is given as input to the expression on the left hand side. This might also be expressed by means of brackets around the expression on the right hand side. Thus, this sequence of Boolean values resulting from the *zipWith* function is given to the higher order function *foldl* which applies the operation $\wedge$ to all Booleans one after another, starting with value *True*. This leads to *True* when *all* Booleans in the sequence calculated before are *True*, and to *False* otherwise, thus implementing the $\forall$-quantifier in the definition of S'. Hence, the overall effect of the function *safeAll* is that it checks whether queen position p can be safely added to the configuration *qs*.

3.2. Word-for-word Translation: $queens_1$

A first implementation of the function $\mathcal{Q}_n$ as defined in (2), the function $queens_1$ is defined as a word-for-word translation[2] of definition (2) in Listing 3.

Listing 3 $queens_1$.

$$queens_1\ 0 = [[\,]]$$
$$queens_1\ n = [qs <: p \mid qs \leftarrow queens_1\ (n-1),\ \ p \leftarrow [1\,..\,N],\ \ safeAll\ qs\ p]$$

Note that in the mathematical definition $\mathcal{Q}_n$ is a *set*, whereas in Haskell $queens_1$ is a *function* which takes n as an argument, and $queens_1\ n$ is the set corresponding to $\mathcal{Q}_n$.

Note also, that the set $\mathcal{Q}_n$ is implemented as a *list*, indicated by "[...]" instead of "{...}", which introduces an order in the elements. The notation "$\leftarrow$" iterates through the elements in a list in the order in which they occur, whereas the order is irrelevant for the corresponding notation "$\in$" for sets. Finally, the operation <: glues the element p to the end of the list *qs*.

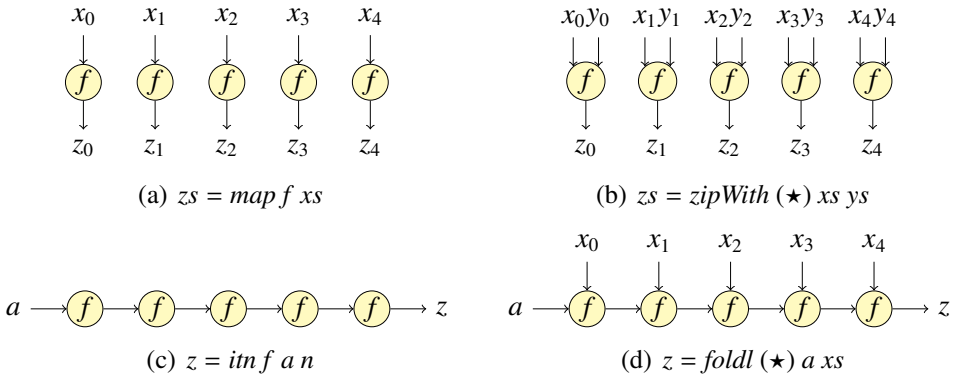
The definition of $queens_1$ is given in the form of a *list comprehension*. Evaluating such an expression involves *backtracking*, which is a standard pattern for problems such as the N queens problem. Though list comprehension is the direct, and more or less word-for-word translation of definition (1), CλaSH cannot directly map such expressions to an FPGA, so we have to apply a further rewriting step. For that we will use *higher order functions*, that is, functions which have a *function* as argument or result.

3.3. Intermezzo: Higher Order Functions

Within a sequential programming environment, higher order functions express *patterns of repetition*, whereas in a parallel environment such as an FPGA they express *architectural structures*. Some standard higher order functions we need here are *map*, *itn*, *filter*, *foldl*, and *zipWith*, of which the last two were already introduced above. The informal meaning of these functions is:

map f xs : the function f is applied to all elements in the list *xs* separately,
itn f a n : the function f is applied to the starting value a repeatedly, n times in total,
filter f xs : the result is the sublist of those values of *xs* for which the Boolean valued function f is *True*,
foldl f a xs : the (binary) function f is repeatedly applied to the accumulative value so far (starting with a) and the next element of *xs*,

[2] Listing 3 again includes some smoothed Haskell symbols: "$\leftarrow$", "N" are in Haskell written as "<-", "nqs", respectively.



(a) $zs = map\ f\ xs$ (b) $zs = zipWith\ (\star)\ xs\ ys$

(c) $z = itn\ f\ a\ n$ (d) $z = foldl\ (\star)\ a\ xs$

Figure 5. Architectural structures corresponding to higher order functions.

zipWith f xs ys : the (binary) function f is applied to the pairs of corresponding elements from *xs* and *ys*.

As an example of such a repetitive pattern, the definition of *itn* is given in Listing 4 in a recursive way.

Listing 4 *itn*.

$$itn\ f\ a\ 0 = a$$
$$itn\ f\ a\ n = itn\ f\ (f\ a)\ (n-1)$$

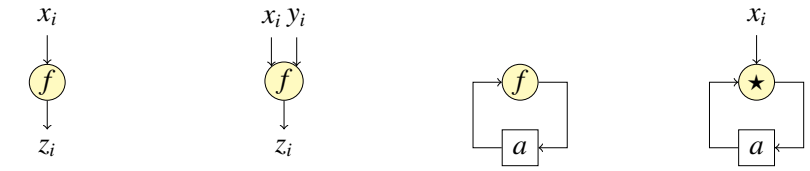
The architectural structures indicated by these higher order functions are shown in Figure 5. The length of the lists, as well as the value n in the case of *itn*, have to be known at compile time, in order to generate the corresponding hardware architectures.

Filter. The function *filter* has no direct architectural representation, since the length of the result list is not known beforehand, and thus no (fixed) architecture can realize this directly. We use two alternative functions to cater to this: *hwfilter*, and *hwfilterL*. The first marks values with *True* or *False* depending whether they have the required property or not, the second moves the elements that possess the required property to the left, and indicates in the result how many elements have that property. For example, let *even* be the Boolean valued functions that yields *True* for even integers, and *False* for odd integers, then:

$$\begin{aligned} &filter\ even\ [1\,..\,6] && \Rightarrow [2,4,6] \\ &hwfilter\ even\ [1\,..\,6] && \Rightarrow [(False,1),(True,2),(False,3),(True,4),(False,5),(True,6)] \\ &hwfilterL\ even\ [1\,..\,6] && \Rightarrow ([2,4,6,4,5,6],3) \end{aligned}$$

Note, that *hwfilter* is just a specific application of *map*. Note also, that in the third case only the first three elements are relevant, the remainder is just the remainder of the original list $[1\,..\,6]$.

Sequentialization. Fully unrolling higher order functions on an FPGA to their corresponding architectural structures may require a huge amount of hardware resources, to such an extent that the design will not fit on an FPGA. The alternative is not to unroll a higher order function completely, but to execute it on the FPGA over time, that is, requiring several clock cycles. In Section 1 we illustrated this issue with a sequentialization over time of the polynomial function, leading to the definition of $f_t(x)$, which specifies the architecture in Figure 2. The generalization of this sequentialization for the higher order functions presented above is shown in Figure 6, where it is intended that every clock cycle the values x_i, y_i arrive.



(a) *map* (b) *zipWith* (c) *itn* (d) *foldl*

Figure 6. Higher order functions over time.

In sub-figures 6(c) and 6(d) the accumulative nature of *itn* and *foldl* requires some memory elements. In these memory cells, the starting value a is indicated.

Where in the fully unrolled architectural structures (as in Figure 5) the number of times that a function has to be applied is explicit in the structure, there has to be an additional provision to count the number of applications when the higher order function is executed over time. That is not shown in Figure 6, but will become clear later on.

Finally, we remark that it is not necessary to execute just one application per clock cycle, as shown in Figure 6. Instead, several applications may be executed in parallel during the same clock cycle, and these may be repeated over time to reach the total number of function applications. It falls outside the scope of this paper to discuss all possibilities for space-time trade-offs, we refer to [8], generalized in [9].

Polynomial function. At this point we shortly return to the introductory example on the polynomial function in Section 1, and define two of the three variants of the polynomial function using the higher order functions introduced above. Using higher order functions opens the possibility to formulate the definitions in a generic way that works for polynomials of any degree. The general form of a polynomial function is

$$f\,(x) = a_{n-1}{*}x^{n-1} + a_{n-2}{*}x^{n-2} + \cdots + a_0{*}x^0$$

First, we define the exponentiation function *pow* as repeated multiplication ($(x*)$ is the function that multiplies with x):

$$pow\ x\ i = itn\ (x*)\ 1\ i$$

Thus, we have

$$map\ (pow\ x)\ [n{-}1, n{-}2, \ldots, 0]\ \Rightarrow\ [x^{n-1}, x^{n-2}, \ldots, x^0]$$

Note, that we also have

$$zipWith\ (*)\ [a_{n-1}, a_{n-2}, \ldots, a_0]\ [x^{n-1}, x^{n-2}, \ldots, x^0]\ \Rightarrow\ [a_{n-1}{*}x^{n-1},\ a_{n-2}{*}x^{n-2},\ \ldots,\ a_0{*}x^0]$$

Now let the sequence of co-efficients $as = [a_{n-1}, a_{n-2}, \ldots, a_0]$, then me may write the general form of a polynomial in Haskell as follows:

```
f' x = foldl (+) 0 ys
  where
    n  = length as
    xs = map (pow x) [n-1,  n-2..0]
    ys = zipWith (*) as xs
```

Intuitively, the architecture corresponding to f' is given in Figure 7, and resembles Figure 1(a).

To conclude the example on the polynomial function, we also give the definition of f_2 from Section 1 using higher order functions:

$$f_2'\ x = foldl\ (\lambda t\ a \rightarrow t * x + a)\ 0\ as$$

The architecture corresponding to this definition is already shown in Figure 2(a).

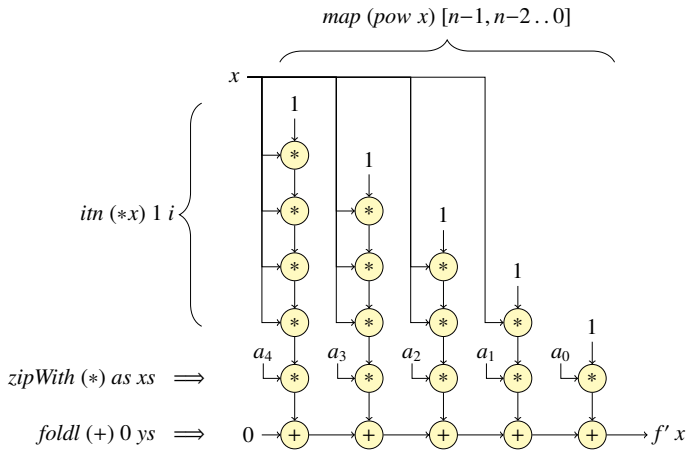


Figure 7. Architecture for polynomial function using higher order functions.

3.4. Removing List Comprehension: queens$_2$

In a list comprehension it is difficult to recognize automatically the intended architectural structure, so we now transform the list comprehension from Listing 3 using higher order functions, since these all specify specific structures. This list comprehension was the following expression:

$$[\ qs <: p \mid qs \leftarrow queens_1\ (n-1),\ p \leftarrow [1\,..\,N],\ safeAll\ qs\ p\ ] \tag{3}$$

In this expression, qs is extended with those p values from the list $[1\,..\,N]$ for which *safeAll qs p* is *True*.

For a given sequence qs this can also be realized by first filtering the allowable p-values from the total list, using the higher order function *filter*:

$$ps = filter\ (safeAll\ qs)\ [1\,..\,N]$$

and then concatenating all the elements of ps to qs:

$$map\ (qs <:)\ ps$$

Using the "pipelining" operation \$, this can be combined in the function *extensions*, defined in Listing 5. Note that here again partial application is exploited: both (qs<:) and (*safeAll qs*)

Listing 5 *extensions*.

```
extensions qs = map (qs<:) $ filter (safeAll qs) [1 .. N]
```

are functions which still need a value p before they can be executed.

Now we may proceed with the same recursion as in Listing 3, and apply the function *extensions* to every sequence qs at level $n-1$ using the higher order function *map*. Note that we have to *concatenate* the results, since *extensions* yields a *sequence* of sequences. This leads to Listing 6.

Note that, the higher order functions in *queens*$_2$ (possibly nested in *extensions*) explicitly indicate that processing may be done in parallel, whereas in the equivalent list comprehension expression 3 this is not directly visible. In fact, the evaluation of expression 3) proceeds purely sequential.

Listing 6 $queens_2$.

```
queens_2 0 = [[]]
queens_2 n = concat $ map extensions $ queens_2 (n-1)
```

3.5. Removing Explicit Recursion: $queens_3$

Listing 6 is still recursive, hence difficult to map to an FPGA. However, in this case the recursive pattern is of the form of the higher order function *itn* as presented in Listing 4, which leads to Listing 7. Here, ∘ denotes *function composition*[3], corresponding to the first

Listing 7 $queens_3$.

```
queens_3 n = itn (concat ∘ map extensions) [[]] n
```

$ in Listing 6. Just like $, function composition has a low priority, so the function that is repeatedly applied by *itn* first applies *map extensions*, and then applies *concat*.

By way of illustration of testability, we set N=5 and execute the expression

$$queens_3\ N$$

in Haskell. The outcome is (with integers as a shorthand notation for lists of digits):

$$[\,02413, 03142, 13024, 14203, 20314, 24130, 30241, 31420, 41302, 42031\,]$$

This list may be checked by actually putting queens on a $N \times N$ chessboard, or one may compare it with the branches in Figure 4. Replacing $queens_3$ by $queens_2$ or $queens_1$ gives the same result. Besides, all definitions are provably equivalent.

3.6. Towards Fixed Length Lists: $queens_4$

The definition of $queens_3$ requires the higher order function *filter*, which is used in the definition of *extensions*. As mentioned before, on hardware lists cannot change in length, hence, we still have to make sure that the design only exploits fixed length lists. Changing *filter* to *hwfilter* will take care of this, but will also cause a change in the result type of the filtered values: they then will be 2-tuples of a Boolean value and a position instead of just a position. Hence, the change of *filter* into *hwfilter* has consequences for the other definitions.

For the same reason, *qs* must have length N, that is, we need a value n to indicate the initial part that is already decided to be safe. This gives rise to a 2-tuple, thus a (partial) configuration now will be of the form:

$$(qs, n)$$

If *qs* contains a conflict, the value of n will be -1. The operation <:: ensures that a value p that can safely be added to *qs* is inserted into *qs* in the correct position, and n is increased by 1. If p cannot be safely added to *qs*, then n is changed into -1. Clearly, if n already equals -1, then it remains like that.

Likewise, the initial value used by $queens_4$ now also has to be changed from the empty list into a value of the form (*qs*, 0), for example (*replicate N* 0, 0) (*replicate* produces a list of N zeros), leading to Listing 8. Note that the sequence of distances *ds* can now be

[3] In Haskell, function composition is expressed by the "." operator.

Listing 8 $queens_4$

```
safeF p q d = d ⩽ 0 ∨ (p ≠ q ∧ abs (p−q) ≠ d)
safeFAll (qs, n) p = foldl (∧) True $ zipWith (safeF p) qs ds
  where
    ds = [n, n−1 .. n−N + 1]
extensionsF (qs, n) = map ((qs, n)<::) $ hwfilter (safeFAll (qs, n)) [1 .. N]
queens4 n           = itn (concat ∘ map extensionsF) [(replicate N 0, 0)] n
```

calculated, but may give rise to negative values, causing the additional condition $d \leqslant 0$ in the definition of *safeF* (the "*F*" stands for "fixed"). Comparing these definitions with the corresponding earlier versions shows that the changes are well tractable, and the structure in the definitions remains the same.

3.7. Final Remarks

We end this section with the remark that *all* functions $queens_i$ are valid Haskell definitions, and so they are executable in a standard Haskell environment. Every expression of the form $queens_i\ N$ calculates the list of correct queen configurations on a chessboard of size $N \times N$, so that outcomes can be tested and compared. We experience this a great advantage in hardware design.

Note that $queens_4$ is very inefficient, since now all possible sequences of length N will be calculated, and only the correct ones will be marked with the length N, whereas the vast majority will be marked with -1. Hence, N^N configurations will be calculated. For example, for N=5, 3125 configurations will be calculated from which only 10 are correct (see Figure 4). This can be checked by evaluating the expressions

```
queens4 N
map fst $ filter ((≡ N) ∘ snd) $ queens4 N
```

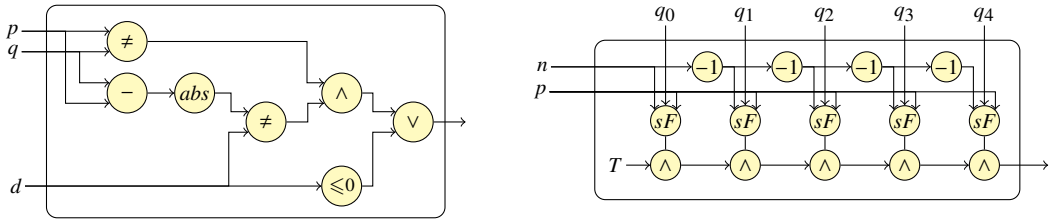
in Haskell. Here, *fst*, *snd* choose the first and second element of a 2-tuple, respectively. Thus, the second expression first filters away those 2-tuples of which the second element is not N, and then chooses the first element of the remaining 2-tuples.

As already mentioned in Section 3.3, this also means that if the definitions of $queens_4$ is mapped directly onto an FPGA, using the architectural structures as described in Section 3.3, the result will be a fully parallel implementation of the N queens problem, noting that it will fit on an FPGA for small N only. Instead of unrolling all higher order functions according to the corresponding architectural structures, we need to calculate (some of) them over time. This is discussed in the next section, in combination with a transformation into Mealy Machines.

Finally, as a preliminary indication of the direct correspondence of a functional definition to hardware architecture, Figure 8 shows the architectures of the definitions of *safeF* and *safeFAll* (*sF* is shorthand for *safeF*).

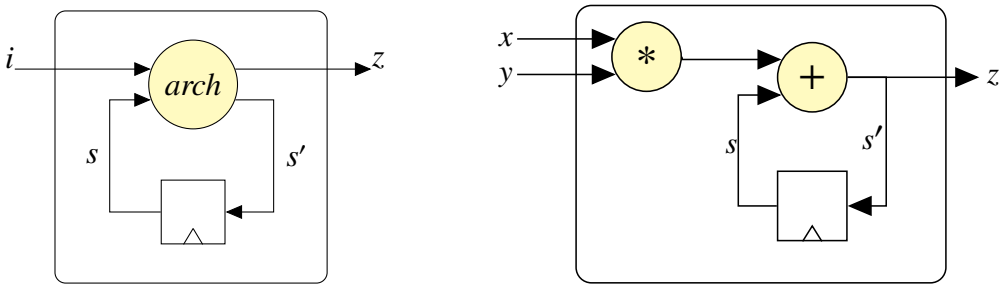
4. Towards Hardware

In this section we will discuss how to proceed from the above given definitions towards specifications in the format of a Mealy Machine, which then can be translated into synthesizable VHDL by the CλaSH compiler such that actual hardware can be generated by a synthesis tool. Below we will first describe the general format of a Mealy Machine in Haskell, and



(a) *safeF* (b) *safeFAll*

Figure 8. Architectures of *safeF*, *safeFAll*.



(a) *General format* (b) *Multiply−accumulate*

Figure 9. Mealy Machines.

after that we will reformulate the Haskell definitions given before into the format of a Mealy Machine.

4.1. Mealy Machine

We will specify a hardware component in the form of a *Mealy Machine* (see Figure 9(a)), that is, as a function with two arguments: the first argument represents the internal *state s* of the architecture, the second argument the external *input i*. The result of such a function, called *arch* below, is a 2-tuple consisting of the *updated state* s' and the *output z*, where both should be specified by appropriate local definitions:

```
arch s i = (s', z)
    where
    s' = ···
    z  = ···
```

Hence, the function *arch* describes what happens during a single clock cycle.

Note that an architecture in the form of a Mealy Machine thus can be written as a Haskell function. To simulate how such an architecture behaves over time, we need to execute it many times in a row, consuming a sequence of inputs, producing a sequence of outputs, and meanwhile updating the internal state. This is realized by the function *sim* (see Listing 9), a higher order function that proceeds by applying its first argument f, which denotes the Mealy Machine that is being simulated, to the current state s and the current input i. That results in the updated state s' and the output z, after which *sim* continues with state s' and the rest of the inputs *is*. Note, that the value z in this result list can be replaced by any expression, for example, to inform the designer how the contents of certain registers in the state s develops over time. In this definition, the operation ":" puts an element in front of a list. Thus, *sim* generates a sequence of values z, meanwhile updating the state s.

As an example, we mention a multiply-accumulate architecture (see Figure 9(b)), defined by the following Mealy Machine:

Listing 9 *sim*.

```
sim f s []     = []
sim f s (i:is) = z : sim f s' is
                 where
                   (s',  z) = f s i
```

```
macc s (x,  y) = (s',  z)
  where
    s' = s + x * y
    z  = s
```

Thus, according to the pattern of the Mealy Machine as given above, the new state is the current state plus the product of the values on the input (note, that the input now consists of two values x and y), and the output is the current content of the state. Taking the list $[(1,1),(2,2),\ldots]$ as testinput, and 0 as the initial value of the state, the simulation using the function *sim* will proceed as follows:

$$\begin{aligned} \mathit{sim}\ \mathit{macc}\ 0\ [(1,1),(2,2),\ldots] &\Rightarrow 0 : \mathit{sim}\ \mathit{macc}\ 1\ [(2,2),(3,3),\ldots] \\ &\Rightarrow 0 : 1 : \mathit{sim}\ \mathit{macc}\ 5\ [(3,3),(4,4),\ldots] \\ &\Rightarrow 0 : 1 : 5 : \mathit{sim}\ \mathit{macc}\ 14\ [(4,4),\ldots] \\ &\Rightarrow \cdots \end{aligned}$$

4.2. Naïve approach

A direct way to build a Mealy Machine from the definitions given so far, is to leave the state empty, indicated by the empty tuple (), and let the function queens_4 calculate the output straight away as in Listing 10. In this code, the underscore means that the argument at the

Listing 10 Mealy Machine $\mathit{queensM}_0$.

```
queensM0 () _ = ((),  queens4 N)
```

input position also is not relevant, so a clock tick as input signal will be sufficient. However, this means that the output neither depends on the state, nor on the input. Hence, after translation to VHDL by CλaSH, a modern synthesis tool will optimize the architecture away since it can completely calculate the outcome at compile time.

But even though this problem can be solved by taking, for example, the sequence $[1\,..\,N]$ as an input argument, this fully parallel solution still is naïve. The reason is, as already remarked in Section 3.7, that the area taken by queens_4 is huge and will extend outside the borders of the FPGA. Consequently, the fully parallel nature of $\mathit{queensM}_0$ has to be changed such that (parts of) the architecture are sequentialized and executed over time. As described in section 3.3 one way to do so is to choose the sequential variant of one or more higher order functions.

There are several ways to do so. A first attempt is to sequentialize the function *itn* in the definition of queens_4 in Listing 8, the last line of which is repeated here:

$$\mathit{queens}_4\ n\ =\ \mathit{itn}\ (\mathit{concat}\ \circ\ \mathit{map}\ \mathit{extensionsF})\ [(\mathit{replicate}\ \ 0,0)]\ n$$

The Mealy Machine sequentializing *itn*, that is, executing it over time, has as its body the function the is iterated by *itn*. Thus, in this case the function in the body of the MEaly Machine is:

$(concat \circ map\ extensionsF)\ [(replicate\ \ 0, 0)]\ n$

The resulting Mealy Machine is given in Listing 11.

Simulating the execution of this Mealy Machine by means of the simulation functions *sim* must start from the same initial value as for *itn* in the definitions of $queens_4$. This is expressed in the test expression *TestM1* in Listing 11.

Note that, during this simulation every clock cycle the Mealy Machine $queensM_1$ outputs the current state, and updates the state s with the function f. In $testM_1$ the simulation will

Listing 11 Mealy Machine $queensM_1$.

```
queensM1 s _ = (f s,  s)
  where
    f = concat ∘ map extensionsF
testM1 = sim queensM1 initstate [0..N]
  where
    initstate = [(replicate N 0,  0)]
```

run $N + 1$ clock cycles. Clearly, real hardware does not stop after $N + 1$ clock cycles, but we will not go into that.

The state of $queensM_1$ is a list of 2-tuples of the form $(qs,\ \ n)$, where qs is a (possibly partial) queens configuration, and n denotes the initial part of qs that is correct. Note, that in $testM_1$ the initial state of the simulation is the same value as the starting value of the *itn* function in $queens_4$. The result of $testM_1$ is the sequence of the $N + 1$ states that the Mealy Machine went through, including all the incorrect and incomplete queen configurations. In order to see the end solutions, we have to select from the result of $testM_1$ those 2-tuples for which $n = N$. For example, the following expression calculates the end result:

$map\ fst\ \$\ filter\ ((\equiv N) \circ snd)\ \$\ concat\ testM_1$

However, evaluating the expression

$map\ length\ testM_1$

gives the outcome (for $(N = 5)$:

$[\,1, 5, 25, 125, 625, 3125\,]$

That means that the state should be big enough to contain 3125 2-tuples, in general, N^N. Given that for $N = 5$ only 10 configurations are correct, this is still a very inefficient usage of resources and will mean that the Mealy Machine will fit on an FPGA for small N only.

4.3. Further Sequentializations

In $queensM_1$ the only higher order function which was sequentialized was *itn*, all other higher order functions were executed in parallel, that is, within a single clock cycle. In order to sequentialize the architecture further, we have to consider the other higher order functions as well. There are many possible choices to realize that. In this section we will discuss one of them. We will first discuss the sequentialization of $queens_3$, that is, the solution which does not yet take into account the fact that on hardware lists have to be of fixed length. The reason to choose $queens_3$ is that it is easier to manipulate and test, and it nevertheless gives a good view on the transformation steps. Experience showed that in general it is a valuable approach

```
1.  safe p q d      = p ≠ q ∧ abs(p−q) ≠ d
2.  safeAll qs p    = foldl (∧) True $ zipWith (safe p) qs ds
                      where
                        n  = length qs
                        ds = [n, n−1..1]
3.  extensions qs   = map (qs <:) $ filter (safeAll qs) [0..N−1]
4.  queens_3 n      = itn (concat ∘ map extensions) [[]] n
```

Figure 10. Overview of the definitions needed for $queens_3$.

to first abstract away from hardware aspects, such as the necessity of counters, and to do domain exploration in a more canonical Haskell format.

The definition of $queens_3$, for convenience repeated in Figure 10, contains several higher order function shown that are possible candidates for sequentialization: *itn*, *map* (twice), *filter*, *foldl*, and *zipWith*. In the context of this paper we choose to restrict ourselves to the sequentialization of *itn* and both *map* functions leaving the other higher order functions parallel. Note, that this gives rise to a parallel kernel which is repeated over time.

Thus, we assume that the list *ps* that is the result of *filter* on line 3 is produced in parallel, but the processing of its elements by the *map* function on line 3 is sequential. That is to say, only one element *p* from *ps* is processed at a time, the others have to wait. Clearly, for that some memory is needed. Since (*qs*<:) is the function that not only has to be applied to *p*, but later in time also to the waiting remainder of *ps*, there is also memory needed for keeping *qs*.

Note that *extensions* produces a list of qs' lists for every list *qs* it processes, the *concat* function is necessary. However, on actual hardware *concat* consists of simply passing over the values, either in space or in time. Thus, executing *itn* over time means that it can continue immediately with qs'. The *filter* function on line 3 again delivers, in parallel, a list ps' of elements which each individually have to be glued to qs'. As before, now one element of ps' is taken to be processed further, and the rest of ps', as well as qs' have to wait to be processed later in time.

One way to solve this is by using a *stack*. In Figure 11 a global overview of this solution is given, and the specification is shown in Listing 12. Figure 11 shows that each item in the stack consists of the lists *qs* and *ps*, where *qs* is an already computed correct partial queen configuration, and *ps* contains the possible extensions. The *head* of *ps* is glued to *qs*, and given to the function *safeAll* (indicated as *sA* in the figure). The result of this is used by *filter* to calculate ps', the next values to be glued to qs'.

The stack is implemented as a list of 2-tuples of the form (*qs*, *ps*). A case analysis (not shown in Figure 11) is required to update the stack, based on the questions how long *qs* already is, how many elements *ps* still contains, and whether ps' has any new elements to offer or not. These cases are described in the definition of the new stack $stack'$ in the **where** clause in Listing 12: the first condition following the "|"-symbols that is *True* determines which expression after the "="-symbol is calculated as the new value of the stack. For this purpose, the stack entities top' and *nexttop* are defined, and where necessary put on top of the stack. It is tedious but straightforward to check these conditions.

The first argument of $queensM_H$ is its state, that is, the stack. This argument is given as *top* : *stack*, meaning the the first entity in the stack is called *top*, whereas the rest of the stack is called *stack*. Note, this also means that if $queensM_H$ is applied to an empty stack, this will result in an error, since we did not define $queensM_H$ for the empty stack. On the first line of the **where** clause, the entity *top* is "unpacked", and its parts are called *qs*, *ps*, respectively.

The output is of a *Maybe* type: if there is no meaningfult result, the output *Nothing* is

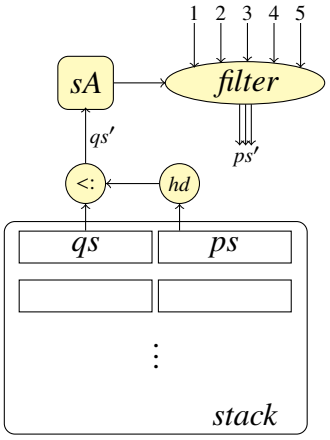


Figure 11. Haskell model with stack.

Listing 12 *queensM*$_H$.

```
queensM_H (top : stack) _ = (stack',  out)
  where
    (qs,  ps) = top
    (n,  m)   = (length qs,  length ps)
    qs'       = qs <: head ps
    ps'       = filter (safeAll qs') [1 .. N]
    (n',  m') = (length qs',  length ps')
    top'      = (qs,  tail ps)
    nexttop   = (qs',  ps')
    stack'    | n' ≡ N−1 ∧ m ≡ 1            = stack
              | n' ≡ N−1 ∧ m > 1            = top' : stack
              | n' < N−1 ∧ m ≡ 1 ∧ m' ≡ 0   = stack
              | n' < N−1 ∧ m ≡ 1 ∧ m' > 0   = nexttop : stack
              | n' < N−1 ∧ m > 1 ∧ m' ≡ 0   = top' : stack
              | n' < N−1 ∧ m > 1 ∧ m' > 0   = nexttop : top' : stack
    out | n' ≡ N−1 ∧ m' ≡ 1 = Just (qs' ++ ps')
        | otherwise         = Nothing
testM_H = filter (≠ Nothing) $ sim queensM_H initstack [1 ..]
  where
    initstack = [([ ],  [1 .. N])]
```

given, whereas, when the list qs' only needs one more element, then ps' contains maximally one element, and the final result thus is the concatenation (indicated by $+\!\!+$) of qs' and ps', marked with the keyword *Just* from the *Maybe* type. This is used by the test expression *testM*$_H$, which filters away all *Nothing* values from the output.

4.4. Towards Architecture

Now we turn to making lists of fixed length in *queensM*$_H$, that is, we will sequentialize *queens*$_4$ following the same pattern as for *queens*$_3$. That implies the introduction of several counters: n indicates the correct part of qs, m indicates the length of ps, and k indicates which element in ps is the next to be glued to qs. Besides, the function *filter* is replaced by the

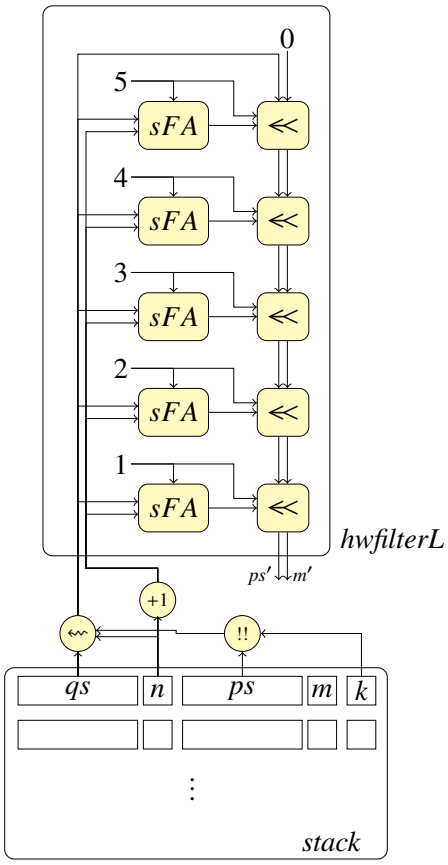


Figure 12. Architecture with stack.

function *hwfilterL*, explained in Section 3.3, since that better facilitates the sequential usage of values.

The resulting specification, together with an expression to simulate the specification, can be found in Listing 13, and the specified architecture is partially shown in Figure 12 (*sFA* stands for *safeFAll*). The notation[4] $xs \leftsquigarrow (i, \quad x)$ inserts x in list xs on position i, and $(xs, \quad n) \ll (b, \quad x)$ inserts x in xs and increases n with 1, when $b \equiv True$. Thus, the sequence of $\ll$ operations takes care of the left shifting in *hwfilterL*.

Otherwise, the global structure of the specification is very close to the specification of $queensM_H$, and we will not go into further details.

Note, the stack is still defined as a list, so it has to be extended with a counter. The maximum size of the stack can be investigated by replacing the output variable *out* in the definition of *queensM* by *out′*, and defining *out′* in the **where** clause as follows:

$$out' \;=\; length\ stack$$

It turns out that the maximum size of the stack is $N-1$, which may also be proven formally.

4.5. Finishing Touches

Apart from the issue that in of *queensM* the stack is still implemented as a list, there are a few other final topics that have to be taken care of before CλaSH can translate the specification *queensM* into synthesizable VHDL. We will shortly mention these issues.

[4]In Haskell, "⇜" and ≪ are written as "<~" and <-<, respectively. List indexing as in ps_k is written as "*ps*!!*k*".

Listing 13 *queensM*.

```
queensM (top : stack) _ = (stack',  out)
  where
    (qs,  n,  ps,  m,  k) = top
    (qs',  n') = (qs ↜ (n,  ps_k),  n + 1)
    (ps',  m') = hwfilterL (safeFAll (qs',  n')) [1 .. N]
    top'    = (qs,  n,  ps,  m,        k + 1)
    nexttop = (qs',  n + 1,  ps',  m',  0)
    stack' | n' ≡ N−1 ∧ k ≡ m−1            = stack
           | n' ≡ N−1 ∧ k < m−1            = top' : stack
           | n' < N−1 ∧ k ≡ m−1 ∧ m' ≡ 0   = stack
           | n' < N−1 ∧ k ≡ m−1 ∧ m' > 0   = nexttop : stack
           | n' < N−1 ∧ k < m−1 ∧ m' ≡ 0   = top' : stack
           | n' < N−1 ∧ k < m−1 ∧ m' > 0   = nexttop : top' : stack
    out | n ≡ N−2 ∧ m' ≡ 1 = Just (qs' ↜ (n',  ps'_0))
        | otherwise        = Nothing
testM = filter (≠ Nothing) $ sim queensM initstack [1 ..]
  where
    initstack = [(replicate N 0,  0,  [1 .. N],  N,  0)]
```

First of all, the fact that the lists occurring in *queensM* are of fixed length is not enough to develop hardware for it: the lengths have to be known *at compile time*. For that, CλaSH uses *vector types*, denoted as

Vec n xs

in which *xs* denotes the list of values in the vector, and *n* the length. By means of vector types, a designer can indicate to the CλaSH compiler, how large lists on the FPGA will be.

The same holds for the integers that occur in the specification: the number of bits needed for the integers that denote the positions of a queen on a chessboard of size $N \times N$ is $\lceil \log N \rceil$.

We remark that also after these additions, every specification that can be compiled by CλaSH is a valid Haskell program[5]. That means that the Haskell type system can check the correctness of the typing, and also, it can *derive* types wherever possible. Hence, the designer only needs to specify types at the top level of the design, the types of all locally defined variables can be derived by the type system.

As an example of the typing, we mention the type definitions for a concrete modification of *queensM*:

```
type QNbr     =  Signed 4
type QVec a   =  Vec 5 a
type StackElm =  (QVec QNbr, QNbr, QVec QNbr, QNbr, QNbr)
type Stack    =  (Unsigned 3, Vec 8 StackElm)
```

Thus, *QNbr* is the type of four bit signed numbers, *QVec a* is the type of vectors of length 5 whose elements have type *a*, and *StackElm* is the type for stack elements, consisting of a 5-tuple. Thus, the first element of such a 5-tuple is a vector of length 5 of four bit signed

[5]The opposite is not the case.

numbers. Finally, the type for the stack itself is a 2-tuple of (in reverse order) a vector of eight stack elements, and a three bit unsigned integer indicating the pointer to the top of the stack.

In addition, CλaSH offers slight modifications of the higher order functions, which work for vectors instead of for lists. These modifications are called *vmap*, *vzipWith*, *vfoldl*, etc.

As mentioned already, the resulting specification is still a valid Haskell program, so the simulation function *sim* defined earlier is still usable for testing the code that is translated by the CλaSH compiler into VHDL, which in turn is synthesized towards actual hardware.

5. Results for FPGA

The final specification is compiled into VHDL for a few values of N. The compilation was done for an Altera Cyclone 2 FPGA (EP2C20F484C6), which is part of a DE1 board, used for educational purposes at the University of Twente. The synthesis for this FPGA was done by Altera's standard tool Quartus.

The RTL schemata of the functions *safeF* and *safeFAll* are shown in Figure 13, where the green boxes in the right half of the schema of *safeFAll* are all *safeF* components. These schemas are well comparable with the hand-made pictures of the architectures of the same functions in Figure 8.

Figure 14 shows some synthesis results for the architecture generated by CλaSH from the specification *queensM*, for a few different values of N: the compilation time needed by CλaSH, the maximum clock frequency, the number of logic cells used (between brackets: the relative part of the total area), and the number of registers.

6. Discussion and Conclusion

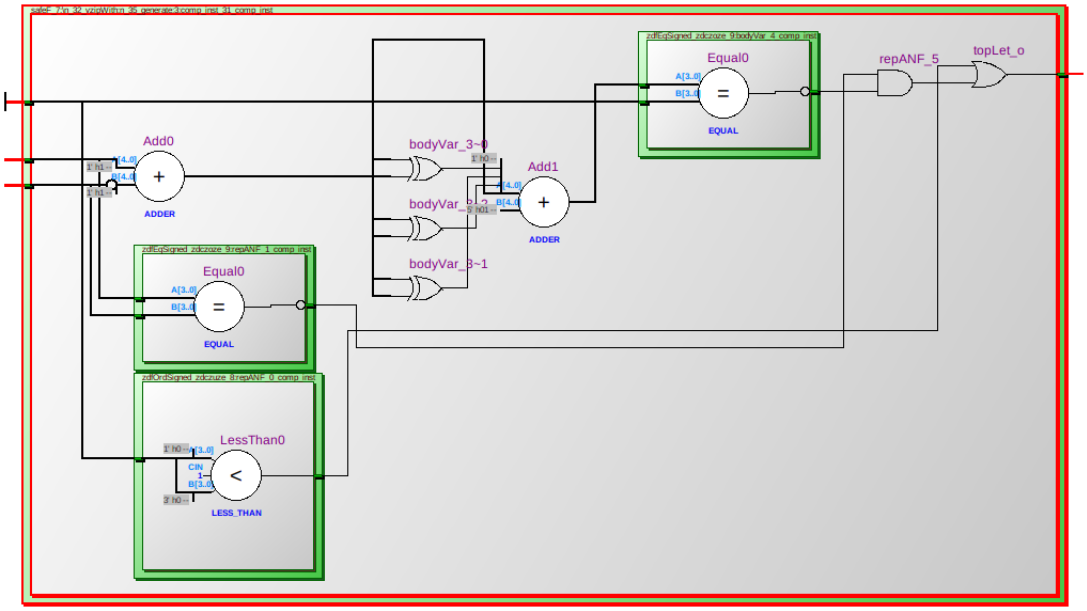
In this paper we presented a methodology for deriving a hardware architecture from an initial mathematical definition in a systematic way, and we showed some results. The programming language Haskell was chosen as the specification language. It is is very suitable as a simulation environment for the various steps in the design process. During the first phases in the design process, when the definitions are not yet in the form of a Mealy Machine, simulation amounts to just executing the code. In other words, when no representation of space and time is added yet, the Haskell definitions amount to what usually is called a *behavioural specification*.

If the Haskell definitions are transformed into the form of a Mealy Machine, simulation requires a separate function *sim*. Since the generation of actual hardware from a Mealy Machine only involves automated tools — the compiler CλaSH, and a synthesis tool — the correspondence between the Mealy Machine and the hardware is direct, and simulation using the function *sim* is adequate. Given the nature of the derivation of the Mealy Machine from the behavioural specification, we may conclude that Haskell is a general and powerful framework for designing hardware.

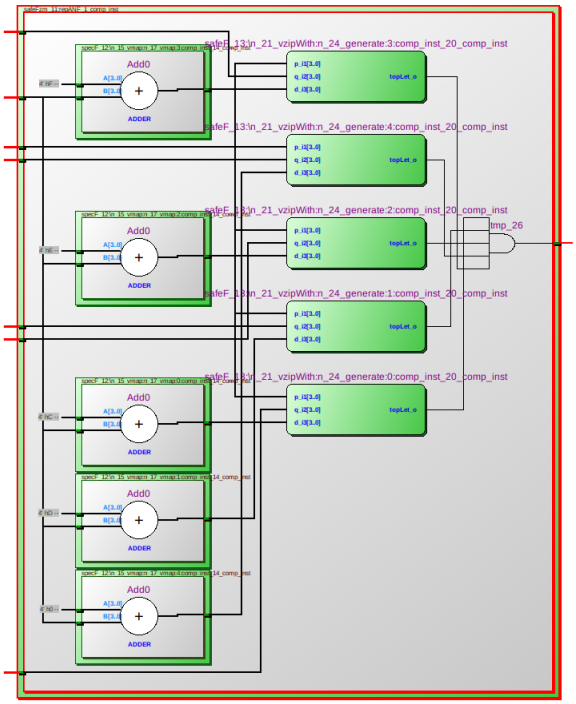
Furthermore, transformations of the code is possible, leading to various architectures. That means that Haskell offers strong support for state space exploration without having to change the semantic perspective, contrary to a design methodology based on the imperative paradigm.

Though we did not show it in this paper, the various specifications of the N queens problem that we derived along the way, can be proven equivalent. Thus, in the presented methodology to design a hardware architecture programming and mathematics amounts to the same thing to a large extent.

At the moment that space and time become considerations in the design methodology, the mathematical perspective is more or less lost, but at that moment the structure in the



(a) *safeF*



(b) *safeFAll*

Figure 13. RTL schema's of *safeF*, *safeFAll*

N	CλaSH compilation time	Maximum clock frequency	Logic cells	Registers
5	13 sec	62 MHz	910 (5%)	371
8	17 sec	41 MHz	2900 (15%)	1156
12	22 sec	32 MHz	5812 (31%)	1988

Figure 14. Synthesis results for *queensM*.

design is such that experimenting with different distributions over space and time are well manageable. For example, using transformations on higher order functions, the effects of exchanging space for time are well visible. Further possibilities for partitioning higher order functions, not discussed in the present paper, can be found in [8,9].

In this paper we did not discuss all possible transformations of the architecture to develop a better design. For example, we did not consider pipelining in order to increase the clock frequency. Also some optimizations on the stack concerning more optimal usage of space may be considered further.

The emphasis in this paper was first of all on the design methodology, and only in the second place on the efficiency of the design. There are further possibilities to improve the efficiency, such as algorithmic improvements. For example, in the design as shown here, each time all positions in a (partial) configuration have to be checked in order to decide whether a new position can be safely added. A more optimal way for that is to introduce so called blocking vectors as described in [5].

Acknowledgements

This research is conducted as part of FP7 project Programming Large Scale Heterogeneous Infrastructures (POLCA), grant agreement 610686, and the Sensor Technology Applied in Reconfigurable systems for sustainable Security (STARS) project, www.starsproject.nl.
We thank the reviewers for their valuable and detailed comments.

References

[1] Philippe Coussy and Adam Morawiec (Eds.). *High-level Synthesis, From Algorithm to Digital Circuit*. Springer Publishers, 2008.

[2] C. P. R. Baaij and J. Kuper. Using rewriting to synthesize functional languages to digital circuits. In J. McCarthy, editor, *14th International Symposium Trends in Functional Programming, TFP 2013, Provo, UT, USA*, volume 8322 of *Lecture notes in computer science*, pages 17–33, Berlin, 2014. Springer Verlag.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in haskell. *Proceedings of the third International Conference on Functional Programming (ICFP)*, pages 174–184, 1998.

[4] N-queens problem in various languages: http://rosettacode.org/wiki/n-queens.

[5] Queens@TUD project: http://queens.inf.tu-dresden.de.

[6] Thomas B. Preußer, Bernd Nägel, and G. Spallek Rainer. *Putting Queens in Carry Chains*. Technical Report, TUD-FI09-03. Department of Computer Science, Technical University Dresden, 2009.

[7] R. Wester, C. P. R. Baaij, and J. Kuper. A two step hardware design method using CλaSH. In *22nd International Conference on Field Programmable Logic and Applications, FPL 2012,* Oslo, Norway, pages 181–188, USA, August 2012. IEEE Computer Society.

[8] R. Wester and J. Kuper. Design space exploration of a particle filter using higher-order functions. In *Reconfigurable Computing: Architectures, Tools, and Applications*, volume 8405 of *Lecture Notes in Computer Science*, pages 219–226. Springer Verlag, London, 2014.

[9] R. Wester and J. Kuper. Deriving stencil hardware accelerators from a single higher-order function. In *Communicating Process Architectures, CPA 2014,* Oxford, UK. To appear, 2014.