Visually Representing and Manipulating Hardware Descriptions in Viskell

Wander Nauta
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
w.nauta@student.utwente.nl

ABSTRACT

This paper investigates how the Viskell visual Haskell programming environment can be used for describing complex hardware in CλaSH. In particular, two concepts are discussed that are important in hardware design and CλaSH programming specifically, but do not normally arise when writing a functional program. Fist of all, in CλaSH, functions can contain memory and therefore have state. Secondly, when designing hardware it is useful to simulate the system that is being designed, showing changes to its state over time. We have added these concepts to the Viskell environment. To show the feasibility of using the combination of Viskell and the CλaSH compiler to design real-world hardware, we have developed a synthesizer chip that can play a simple tune.

Keywords

architecture design, visual and functional programming, Haskell

1. INTRODUCTION

Integrated circuit design was historically a manual process. As the design of integrated circuits became larger and more complex, the process of designing these circuits was automated. Schematic capture tools aided in creating hardware from schematics. Hardware description languages and logic synthesis tools started to appear, which could automatically place and route components to match a description of the intended behaviour.

Both hardware description languages in common use today, VHDL and Verilog, first appeared in the 1980's. They have since been extended and improved. However, we believe that the level of abstraction that these languages offer is still not high enough, and that more productivity gains can be achieved. The topic of this paper is to evaluate whether the visual programming environment Viskell, paired with the hardware description language CλaSH, could be a good approach to hardware design.

Section 2 will give a short overview of the problem domain and the related work this research is built on, both regarding the hardware design aspect as well as the visual programming aspect. The following section, section 3, will describe the problems that needed to be solved and changes that needed to be made to make Viskell work with CλaSH. We continue with a description of our demonstration project, in section 4, and finish with suggestions for future work and conclusions.

24th Twente Student Conference on IT Jan 22nd, 2016, Enschede, The Netherlands.

2. BACKGROUND

The research builds on top of both the CλaSH [1] and Viskell [3] projects and seeks to combine the two into a tool that is useful for hardware designers. This section is meant as a short background on both projects. A description of CλaSH is given in section 2.1, while Viskell is discussed in more detail in section 2.2.2.

2.1 CλaSH

CλaSH is a compiler that allows writing hardware descriptions in (a subset of) the Haskell programming language. It uses Haskell itself as a high-level hardware description language. Programs written in CλaSH are valid Haskell programs, and can be executed to simulate and test the described hardware. CλaSH code can also be translated into a hardware description in VHDL, Verilog or SystemVerilog by the CλaSH compiler. This hardware description can then be used to synthesize actual hardware and program FPGAs.

At a very high level, the CλaSH compiler applies the following transformations:

- *Function definitions* are seen as 'blueprints' for a component, describing both their structure and their behaviour.
- *Function application* is then the instantiation of such a blueprint. In other words, a function application will be synthesized into an actual piece of hardware.
- *Static recursion*, recursion where the recursion depth is known (decidable) at compile time, is transformed into repetition: repeating components and placing them side by side. Recursion implicitly leads to parallel hardware in CλaSH: the various invocations of the components run in parallel.
- *Choice*, which would arise from Haskell constructs like pattern matching, is transformed into components for calculating each of the alternatives ('branches'), as well as a multiplexer for choosing the alternative that applies.

Not all possible Haskell programs can be compiled with CλaSH. Programs that contain unbounded recursion, where the recursion depth is not known at compile time, do not have a sensible representation in hardware. Also, values have to be representable as series of bits (wires), which means that there are some restrictions on the types that can be used. The length of a Haskell list, for example, is not known at compile time. Regular lists can therefore not be used with CλaSH. Instead, CλaSH offers a fixed-size vector type, Vec, which offers much of the same operations. In a similar fashion, functions can be passed around as in Haskell, but CλaSH has to be able to remove this abstraction at compile time to be able to produce actual hardware. In practice, CλaSH code looks very much like Haskell code (see figure 6 for an example).

The hardware descriptions that CλaSH generates can be used to build application-specific integrated circuits (ASICs), or to

program field-programmable gate arrays (FPGAs). Application-specific integrated circuits are, as the name implies, application-specific: they are designed and built for a specific purpose. FPGAs, on the other hand, can be reprogrammed after they are made ('in the field') to perform different functions. For this research, we have focused on FPGAs only; although FPGAs are expensive, the cost of building an ASIC production line for a single demonstration unit, which would then be used once, would have been prohibitive.

2.2 Visual programming languages

Almost all programming languages are visual: they represent abstract concepts or computations using symbols. Most of these languages are textual: they use keywords, usually from English, and punctuation, usually from mathematics, to convey structure and meaning. The term 'visual programming language' is usually reserved for languages that are *not* textual. For this section, and in the rest of the paper, we will limit the definition of the term 'visual programming language' even further to 'a programming language that use a visual notation with nodes and lines'.

2.2.1 Domain-specific languages

The nodes-and-lines concept is sometimes used in places where a lot of flexibility is required, but where the user is not necessarily a programmer.

For example, the node-based compositor interface that exists in some 3D modeling packages like Blender [2] allows non-programmers to create materials by combining images, values, and functions in a visual manner. These materials are not programs in and out of themselves: they do not have control flow, only data flow, and there is no way to do iteration or recursion. However, they do allow artists to create images that would otherwise require hand-written shader programs.

Some game engines, including the Unreal Engine, have domain-specific visual scripting languages that blur the line between 'merely' a visual data representation and a programming language. These so-called 'blueprints' [4] are written in a visual representation of a simple (typed) imperative scripting language that includes arrays, control flow, and function abstraction.

2.2.2 Viskell

The Viskell [3] project started in the beginning of 2015 as a second-year design project. It was inspired by the Lambdas with Bowties project by Philip Hölzenspies, and it tries to find a readable and compact way to visualize functional programs. So far, it has produced a proof-of-concept visual environment that allows the user to write programs in a (growing) subset of Haskell.

Viskell follows a boxes-and-wires model, where boxes represent the application of Haskell functions, and wires represent the flow of data from one function application (use of a specific function) to another. For example, to add two numbers together, they would be connected to the *input anchors* of an application of the (+) function. Functions can be partially applied, as in Haskell, by dragging arguments from the input side (top) to the output side (bottom), as the *fib* function is in figure 3.

A few language constructs from Haskell have Viskell counterparts. Lambda expressions, which define anonymous functions, become teal-colored lambda blocks in Viskell, which are discussed in more detail in section 3.1. To represent choice, a case block (in maroon) is available. The alternative (lane) that gets used depends on the condition, which can be a pattern (in light blue) or a boolean value (a *guard*). Choice blocks are read from left to right: the leftmost lane that has a matching pattern, or a guard that evaluates to True, gets used. Both these constructs are used in figure 4.

Some commonly-used functions also have special blocks. For example, the (,) function that combines two values into a tuple can also be represented with a joiner, the funnel-shaped blue block in figure 1. Finally, constant values are in pink-colored blocks.

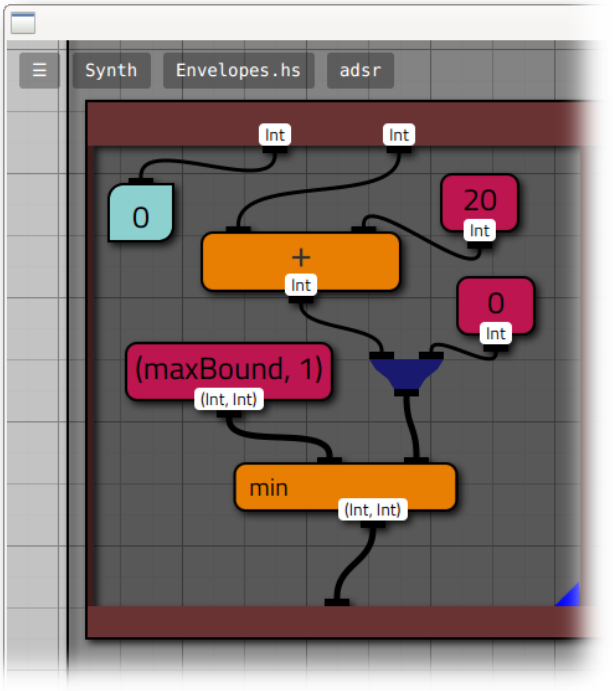

Figure 1. Part of a screenshot of Viskell.

Viskell is also a *live* programming environment: the user's program is type-checked and executed while the user is working on it. Errors, specifically type errors, are highlighted immediately and automatically, and widgets such as sliders allow for quick adjustment of input values. This immediate feedback allows for easy and quick exploration.

The environment is designed to be used on touch-sensitive devices, like tablet computers and table-sized multi-touch enabled screens. When used on a large multi-touch screen, it can be used by multiple users at once, for pair programming or joint experimentation. Viskell is also usable with mouse and keyboard.

Besides being immediately executed, the Haskell code that Viskell generates can also be viewed and exported. However, the generated code is not intended to be human-readable, as can be seen in figure 5.

2.2.3 Related projects

Other experimental programming environments for functional programming languages exist.

The Lamdu [5] live programming environment shares a few important characteristics with Viskell: it supports live type checking and live execution of programs, like Viskell does. However, the language that Lamdu targets is a language based on Haskell, while Viskell targets Haskell itself. Lamdu does not use the nodes-and-lines concept that Viskell uses. Instead, program logic is visualized as a tree that is automatically formatted.

Visual Haskell is an earlier visual notation for Haskell, introduced by John Reekie in [7]. Like in Viskell, boxes represent the application of functions, and lines represent the flow of data from one such application to another.

There are also differences. Visual Haskell intends to find a visual syntax for every construct in the Haskell language, while Viskell does not. It also intends to allow an end user to switch between textual and visual representations of the same program, and mixing textual and visual elements. These are not current goals for Viskell. There are superficial differences as well: Visual Haskell makes more use of icons and iconography than Viskell does, for example to show the types of values that lines carry. In Viskell, data flows

by convention from top to bottom, while in Visual Haskell, it mostly flows from right to left.

Reekie introduces a way of using Visual Haskell to program digital signal processors (DSPs). This, too, has a few similarities with Viskell/CλaSH: Visual Haskell programs can work with signals, which are called *streams* in the thesis but work and will be discussed in section 3.2. In addition, Visual Haskell is intended to look similar to a hardware block diagram.

3. DESIGN

This chapter describes the changes that were applied to Viskell to make it usable for working with CλaSH.

3.1 Large programs and hierarchy

The CλaSH compiler, as mentioned in section 2.1, translates Haskell function definitions into hardware descriptions. Each Haskell function gets its own module (*entity*) in the resulting hardware description language (HDL) code, to make it more obvious which HDL code represents which part of the CλaSH program. To support this, we have added support for Haskell function definitions ('toplevels') to Viskell. With this addition, there are two ways of abstracting and reusing functionality in Viskell/CλaSH:

Lambda blocks define a lambda function: an anonymous function where the argument and result types are unnamed. These blocks can be used like regular blocks: their result is a lambda. The output anchor of lambda blocks can be connected to blocks that expect a function, like the Haskell *map* function. Lambda blocks are teal-colored in Viskell.

Lambda blocks can optionally have a fixed type, as well as a name. Because they have a name, functions defined in definition blocks can be used in different places in a Viskell program without having to drag a connection between them, making definition blocks useful for reusing common subexpressions.

Toplevels are not blocks themselves, but containers for blocks with a name and type. They map to functions in Haskell, which then correspond to HDL files when the output is compiled with CλaSH.

Viskell/CλaSH projects are then a hierarchy of these containers. A *project* contains multiple (Haskell) *modules*, each of which can have a number of toplevel function definitions. Toplevel functions are made up of blocks, including lambda and definition blocks. The hierarchy can be navigated by using a breadcrumb navigation bar, as shown in figure 1: the user is currently editing the 'adsr' toplevel function, inside the 'Envelopes' module, which is part of the 'Synth' project, our example synthesizer.

3.2 State and signals

The *Signal* type from CλaSH is a type that is conceptually similar to an infinite sequence of values. Every element of the sequence represents a stable value at a single moment in time. The *Signal* type allows the CλaSH programmer to work with memory, and more generally with state.

In CλaSH, the *register* function is a function that returns its input signal delayed by one clock cycle. In other words, the *register* function behaves like a hardware register that is continuously overwritten: every cycle, it will output the previous value and read the next. (Contrary to a 'real' (hardware) register, the *register* function has an additional argument that supplies its initial value.)

In Viskell, connections that carry Signals are presented in a different color than connections that carry 'regular' values: normal connections are black, while Signals are blue. (A Signal-carrying connection is shown in figure 2). Some common operations can be used on signals directly; for example, when the (+) function is used to sum two Signals of numbers, it behaves like an adder would in hardware, adding each pair of values from both signals

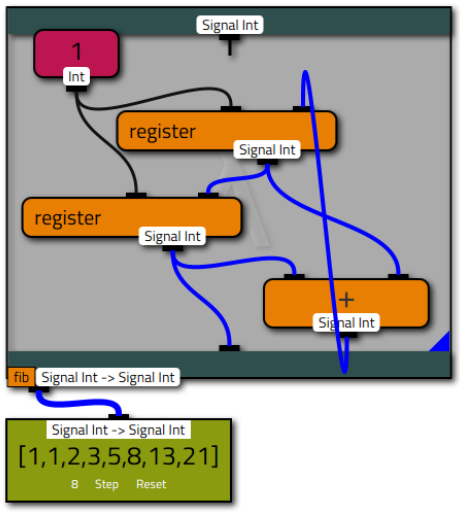

Figure 2. Fibonacci generator, implemented with signals

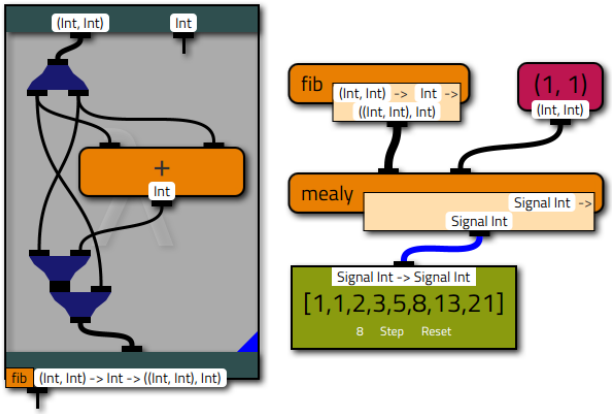

Figure 3. Fibonacci generator, implemented with *mealy*

together. Other functions can be *lifted* so that they work on signals, either using the *fmap* function from CλaSH or the 'lift' button from Viskell's right-click menu.

Figure 2 shows an implementation of a simple function with state in Viskell/CλaSH. The function contains two registers, which both store integer numbers. The initial value of both registers is a constant, the integer one. Every clock cycle, the two (old) numbers are summed using the Signal-aware version of (+). This sum gets written to the first register through a mechanism known as *feedback*. The second register meanwhile takes the old value of the first, which means it is delayed by two clock cycles total. As shown in the simulation block, the entire *fib* function generates $F_n = F_{n-1} + F_{n-2}$, which is the Fibonacci sequence. (The function's input is disconnected and ignored.)

The function in figure 3 also generates a Fibonacci sequence, but uses a Mealy machine [6] instead of using signals in the implementation itself. A Mealy machine is a function that, given a previous state and an input, gives a new state and an output. In our example, the 'fib' function is such a function. (Note that, like in figure 2 but unlike in most Mealy machines, the input value is ignored.) The state in the example is a tuple (pair) that contains the previous two Fibonacci numbers. This tuple is immediately split using a splitter block. The Fibonacci number on the right in the old state switches places in the new state: it moves to the other position in the tuple. The other element of the new state is the sum of both numbers in the old, added together using the regular (+) function from Haskell. Finally, the new state and the

output value (the oldest value) are combined and connected to the function's output. The new state is therefore (F_n, F_{n+1}) and the output is simply F_n.

Working with Signals is not obligatory: the two implementations in figures 2 and 3 could be used interchangeably. When the second implementation is combined with the *mealy* function, they have the same type and the same behaviour.

The first approach has the advantage that it will likely look similar to its implementation in hardware: it will probably be synthesized as two registers and an adder, just as the Viskell implementation contains two *register* blocks and a (+) block. It also makes the registers explicit, which makes the data flow between the registers more obvious.

However, the *mealy* approach is arguably higher-level. The important logic in the example is described as a pure, functional Haskell function. Where the state is stored is implicit. When some logic can be cleanly described as a Mealy machine, that is, a function taking an input and a state and resulting in an output and a state, the *mealy* function can help avoid having to work with signals directly.

Both approaches are valid, and it will depend on the problem at hand which approach is clearer or easier to understand.

3.3 Time and debugging

To aid in debugging and testing stateful functions and functions that work on signals, we have added the *simulation block* (the green block in figures 2 and 3). It is an interface for the CλaSH built-in *simulate* function that takes a function as its input and simulates it, generating a simple increasing sequence as its input and showing the result of evaluating the function up to that point.

The simulation block has a *stepping function*: every time the 'Step' button is touched, the CλaSH function is evaluated for one additional 'clock cycle'. This would be comparable with clicking through a program in an ordinary debugger.

Of course, it is possible to combine the input that the simulation block generates with a different input signal (stimulus). If the stepping ability is not required, the *simulate* function from CλaSH can also be used directly.

4. DEMONSTRATION PROJECT

A major part of the research was to discover whether it was possible to build synthesizable CλaSH descriptions of nontrivial hardware in Viskell. We decided to show this possible by doing it: we have built a demonstration project in CλaSH/Viskell.

The demonstration chip is a synthesizer: a device that has a number of keys for input, and generates sound as its output.

The synthesizer is *polyphonic*: multiple notes can be generated at the same time. A number of oscillators are available, as well as an attack-decay-sustain-release envelope generator or ADSR. The oscillators are responsible for generating sound waves, which can be either square-, sawtooth- or sine-shaped. The ADSR controls the amplitude of the generated sound waves.

On real musical instruments, the amplitude of the sound is not constant but instead follows a curve. The ADSR roughly emulates this curve, basing the amplitude on the time since a key was pressed or released. When a key on a piano is pressed, it takes a short time (the attack time) for the sound to reach its maximum loudness. After it reaches its peak, the sound gets quieter (the decay time). If the sustain pedal is pressed, the volume becomes more or less constant after this decay time. Finally, when the key is released, pianos have a damper for every key that quickly (during the release time) dampens the sound. The demonstration chip simulates this by having the loudness of the generated sound follow a similar curve, making the output sound less artificial.

The CλaSH code for the demonstration project has been implemented in Viskell. However, the entire implementation would be too large to legibly fit here. Instead, we will show and describe a representative part, the sine wave oscillator, below.

4.1 Sine wave oscillator

The sine wave oscillator is responsible for generating a sine wave. A sine wave is a sound wave that has no harmonics: it sounds 'smoother' than other waves.

To generate a mathematically accurate sine wave on the fly, or even a polynomial approximation of one, would be relatively intensive. Instead, the sine wave oscillator from our demonstration chip uses a *wave table*. A block of read-only memory contains half of a sine wave, stored as a vector of sixteen-bit numbers. Only the positive half of the sine wave is stored (i.e. the values of $sin(x)$ where $0 \leq x < \pi$).

The table contains 1024 entries (*samples*), but it can return 2048 different values. When the index requested is above 1024, the address is wrapped around and the result is inverted (since $sin(x) = -sin(x - \pi)$).

The Viskell code for this part of the demonstration project is shown in figure 4. The *asyncRomFile* function is from CλaSH, and reads a file on the machine running the CλaSH compiler. That file, here abssin.dat, is put into a block of read-only memory on the final device. The size of the memory is fixed. The *asyncRomFile* function finally takes a read address, which is attached to the lambda's argument. The result, coming from the block of memory, is unpacked into a signed 16-bit integer. If the address is below 1024, it is passed along as-is. (The identity function *id* in Haskell returns its argument unchanged.) If not, the sample is negated and then returned. All blocks are then wrapped in a definition block named *sinOsc*.

The raw Haskell code that Viskell generates from this is shown in figure 5. Viskell's output is a graph of let-expressions: each block is converted into a let-expression, and refers to other blocks by name. As mentioned in section 2.2.2, the generated output is not intended to be human readable: it is only ever read by GHC (or CλaSH in this case), which will simply ignore (inline) most of the trivial names. With some manual inlining and reformatting, the code is equivalent to the Haskell source in figure 6.

The generated code from figure 5 can be put into a file, which can then be translated by CλaSH into a number of VHDL files. These VHDL files can in turn be compiled into a register-transfer level (RTL) design by a VHDL compiler like Quartus. The VHDL source for our sine wave oscillator would be too long to reproduce here, but Quartus can generate a diagram of the compiled design, which is included as figure 7.

When looking at the RTL diagram at a 90-degree angle, it is easy to draw parallels between it and the original Viskell implementation of figure 4. There are a few differences between the end result and the Viskell version; for example, the *unpack* and *id* functions have been optimized away entirely, and the read-only memory we specified turns out to be a RAM memory with its write-enable port stuck on 0. However, the comparison (LessThan0), negation (Add0) and conditional (bodyVar) parts are clearly visible.

5. RESULTS AND EVALUATION

All components of the demonstration project can indeed be implemented in Viskell, and the generated code can be compiled by CλaSH to produce actual hardware.

The information density of Viskell is lower than that of textual Haskell, and therefore lower than that of textual CλaSH. In our experience, a logic fragment implemented in Viskell will take up more screen space than its textual equivalent or even a description in English (compare figures 4 and 6) at legible text sizes.

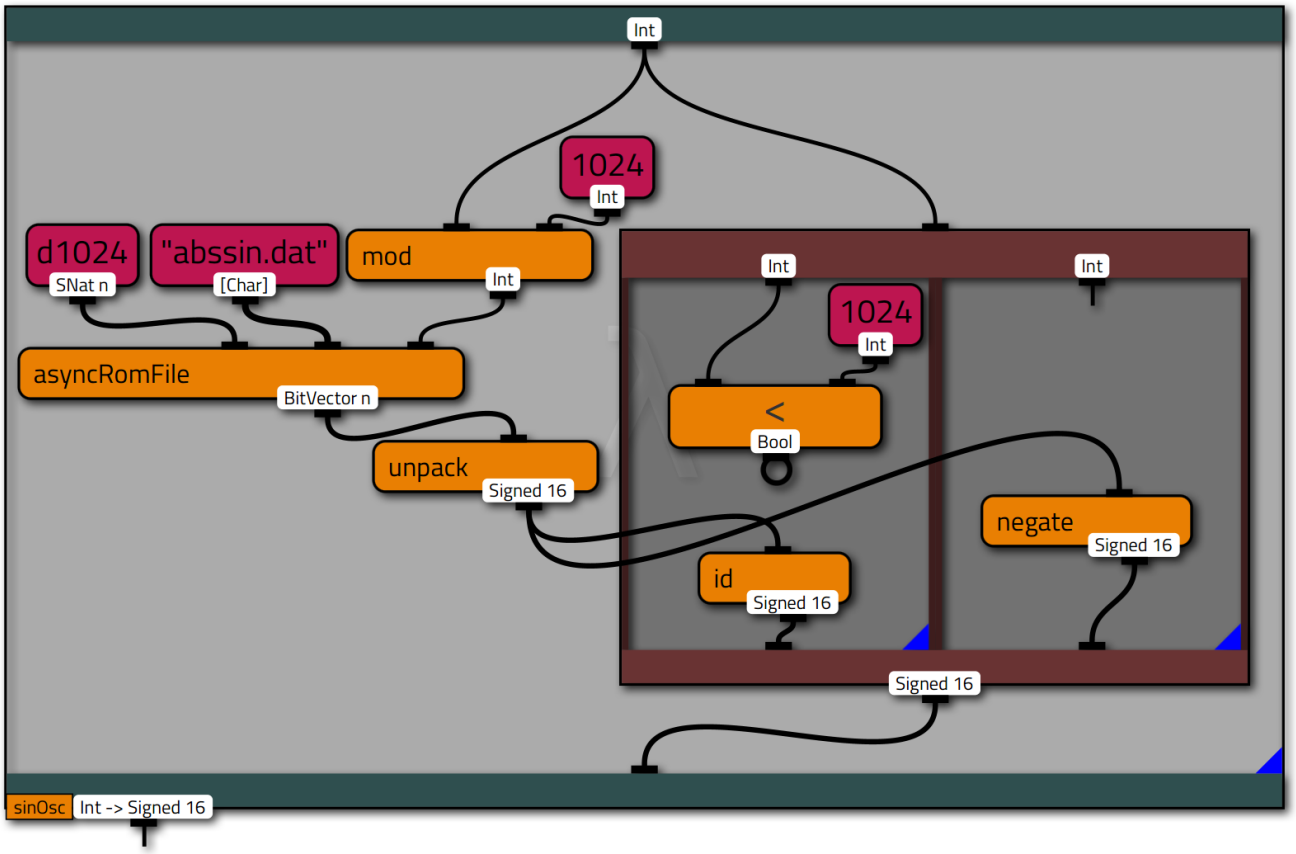

Figure 4. Viskell for the sine wave oscillator

```
(let {} in (\ a_0__71b6dfcb -> (let {val__267acf83 = (1024); res__1f20efe2 = (unpack
    res__7f8d0877); res__7b653b4b = ((mod a_0__71b6dfcb) val__267acf83); val__7f95f0fb = (
    "abssin.dat"); val__68cc9325 = (d1024); res__7f8d0877 = (((asyncRomFile val__68cc9325)
     val__7f95f0fb) res__7b653b4b); choiceoutput__581bbaf5 = case (()) of {() |
    res__49377758 <- (((<) a_0__71b6dfcb) val__267acf83), True <- res__49377758,
    res__1ae92258 <- (id res__1f20efe2), True -> res__1ae92258; () | val__1444fb56 <- (
    otherwise), True <- val__1444fb56, res__5485e1a3 <- (negate res__1f20efe2), True ->
    res__5485e1a3; }; } in choiceoutput__581bbaf5)))
```

Figure 5. Viskell-generated Haskell code for the function in figure 4

```
sinOsc t = if (t < 1024) then (id sample) else (negate sample)
           where sample = unpack (asyncRomFile d1024 "abssin.dat" (t `mod` 1024))
```

Figure 6. Cleaned-up (but functionally equivalent) Haskell corresponding to figure 5

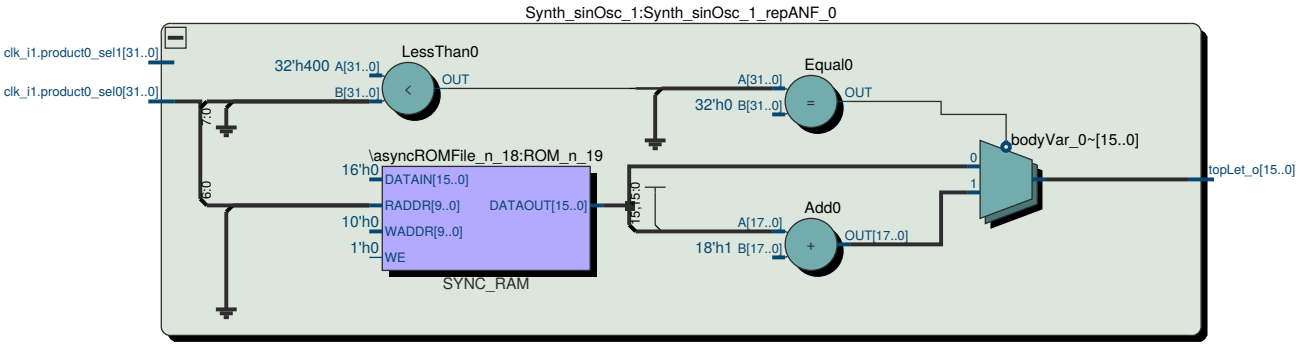

Figure 7. RTL diagram for the sine wave oscillator generated from figure 5

More time was spent on the layout of the Viskell program than on the layout of the Haskell program. Especially trying to find a layout that communicated the intended meaning well was, in the author's opinion, more difficult in Viskell than in Haskell, where well-known conventions and tools for formatting and code layout exist. Adding to the difficulty of formatting a Viskell program was that there was a balance between trying to find a compact layout, and finding a layout where the graph was mostly planar (that is, where none of the wires crossed).

Whether the visual or the textual version of the synthesizer is easier to read and understand is likely to be subjective, and will depend at least in part on the reader's familiarity with Viskell, Haskell, and CλaSH.

6. FUTURE WORK

There is much work, both in research and in development, left to be done on Viskell/CλaSH and on Viskell in general. There are many open questions that could be answered and a great number of missing features that could be added.

Compilation process The process of taking Viskell's output and compiling it first with CλaSH, then taking the CλaSH output and compiling that with a HDL compiler, is not yet automated and has to be done manually. Ideally, one would integrate both compilers into Viskell so that the manual procedure is no longer necessary. However, most FPGA vendors have their own compilation and synthesis tools, and these tools usually do not work with other FPGA's. That would mean many such integrations would have to be developed.

General improvements Not all the Viskell improvements described in the research proposal have been implemented yet. Especially saving and loading programs would make Viskell a lot more useful, and additional tools for structuring programs would be very practical. Ideally, there would be an easy way to convert pieces of a Viskell program between different forms: grouping parts of a Viskell program into a lambda expression, turning a lambda expression into a toplevel, and so on, comparable to the extraction and inlining of functions in other programming languages. However, while general improvements like these would be a good avenue for further development, they would not be a good fit for further research.

User research This research focused on seeing if it was possible to describe hardware in Viskell. We have not evaluated the productivity of doing so, or how it compares to using just CλaSH, or just a hardware description language directly. It could be interesting to do this comparison, especially for different target audiences.

7. CONCLUSIONS AND DISCUSSION

We have demonstrated that Viskell, when combined with CλaSH, is a *possible* tool for developing hardware designs. The demonstration project, a polyphonic synthesizer, shows that non-trivial hardware can be implemented in Viskell, generated as CλaSH, compiled to a hardware description language and finally used on a field-programmable gate array.

However, developing the demonstration chip did not prove that Viskell is a *productive* tool for describing hardware. During the development of the demonstration project, much time was needed to make the layout of the Viskell program look just right. Some time was also lost to interfacing Viskell, the CλaSH compiler, and the HDL development environment. We believe that more work needs to be done to make Viskell/CλaSH a comfortable environment for developing programs.

Finally, more research is required to see how productivity when working on CλaSH code in Viskell compares to writing hardware descriptions in 'regular', textual CλaSH.

8. REFERENCES

[1] C. P. R. Baaij. *Digital circuits in CλaSH: functional specifications and type-directed synthesis*. PhD thesis, Enschede, Jan. 2015. http://doc.utwente.nl/93962/.

[2] Blender Foundation. Blender reference manual: Compositing. https://blender.org/manual/en/compositing/. Accessed Dec. 07 2015.

[3] M. Bruning, K. Hartsuiker, J. J. Kester, W. Nauta, and D. Snijders. Report on Viskell, July 2015. Report of a design project for Technical Computer Science, University of Twente.

[4] Epic Games. Unreal Engine 4 Documentation: Blueprint User Guide. https://docs.unrealengine.com/latest/INT/. Accessed Dec. 07 2015.

[5] E. Lotem et al. Lamdu. https://peaker.github.io/lamdu/. Accessed Oct. 07 2015.

[6] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal, The*, 34(5):1045–1079, Sept 1955.

[7] H. J. Reekie. *Realtime Signal Processing: Dataflow, Visual, and Functional Programming*. PhD thesis, Sidney, Sept. 1995. http://ptolemy.eecs.berkeley.edu/~johnr/papers/thesis.html.