Introducing Kansas Lava

Andy Gill, Tristan Bull, Garrin Kimmell,
Erik Perrins, Ed Komp, and Brett Werling

Information Technology and Telecommunication Center
Department of Electrical Engineering and Computer Science
The University of Kansas
2335 Irving Hill Road
Lawrence, KS 66045
{andygill,tbull,kimmell,esp,komp,bwerling}@ittc.ku.edu

Abstract. Kansas Lava is a domain specific language for hardware description. Though there have been a number of previous implementations of Lava, we have found the design space rich, with unexplored choices. We use a direct (Chalmers style) specification of circuits, and make significant use of Haskell overloading of standard classes, leading to concise circuit descriptions. Kansas Lava supports both simulation (inside GHCi), and execution via VHDL, by having a dual shallow and deep embedding inside our `Signal` type. We also have a lightweight sized-type mechanism, allowing for MATLAB style matrix based specifications to be directly expressed in Kansas Lava.

1 Introduction

In the Computer Systems Design Lab (CSDL) at KU we build systems in hardware and software. We are also avid users of Haskell. Lava [1], an Embedded Domain Specific Language (EDSL) for expressing hardware level concerns, is a natural way for a CSDL member to think about constructing and expressing our systems. In this paper, we introduce our version of Lava, called Kansas Lava, and describe how we use modern functional language techniques including applicative functors and type functions to improve the overall expressiveness of our hardware simulation and synthesis toolkit, and work towards a unified development story of specification to implementation.

Lava is the name given for a family of Haskell hosted Embedded DSLs for expressing (typically) gate-level hardware descriptions. In general, Lava is a *design pattern* for EDSL construction, when trying to capture hardware concerns. This section provides an overview of the well-known Lava design pattern, and in the next section we introduce our variant of Lava.

The central idea in Lava is that, under the correct conditions, we can observe a function as a circuit. Consider this half adder description.

```
halfAdder :: (Bit,Bit) -> (Bit,Bit)
halfAdder (a,b) = (carry,sum)
  where carry = and2 (a,b)
        sum   = xor2 (a,b)
```

Given suitable input, it is possible to execute this function directly.

```
> halfAdder (high,low)
(low,high)
```

We can extract the truth table for the `halfAdder` by applying it to all possible inputs.

```
> [ (b1,b2,halfAdder b1 b2) | b1 <- [low,high], b2 <- [low,high] ]
[ (low,low,(low,low),
  (low,high,(low,high),
  (high,low,(low,high),
  (high,high,(high,low) ]
```

This is classical functional programming. As well as executing this `halfAdder`, in Lava we can also *extract* the internal wiring of the function by applying `halfAdder` to suitability constructed dummy arguments. Consider the following implementation of Lava.

```
data Bit = High | Low | Xor2 Bit Bit | And2 Bit Bit | Var String

and2 (a,b) = And2 a b
xor2 (a,b) = Xor2 a b
high = High
low = Low
```

This is a traditional deep embedding of a domain specific language. In this case, the language is Lava itself. Now, if we apply `halfAdder` with suitably annotated `Var`s, we get a data structure that contains the internal structure of the `halfAdder` function.

```
> halfAdder (Var "a",Var "b")
(And2 (Var "a",Var "b"),Xor2 (Var "a",Var "b"))
```

From structures that represent these wiring diagrams, we can generate structural VHDL that represents the wire routing between established components. In this way, compiling combinational circuits is straightforward, if tedious. Compiling sequential circuits, however, exposes a critical shortcoming with the original Lava design pattern. Specifically, there is no easy way to observe the wiring cycles that exist in sequential circuits. Addressing this issue led to a fork in the design specifics of Lava implementations.

Consider the following circuit for computing the parity of an ongoing signal.

```
-- Parity specification
parity :: Bit -> Bit
parity input = output
  where
    output = xor2 (delay output,input)
```

`parity` is defined as the `xor2` of the current `input` value with the value of `parity` on the previous cycle. The `delay` combinator takes a signal that changes over time, and *delays* the output by one clock cycle.

The earlier trick of using `Var` as a dummy argument inside our deep DSL does not work directly. Assuming we have augmented our deep DSL to include `delay`, applying `parity` to an instance of `Var` gives an infinite result.

```
> parity (Var "x")
Xor2 (Delay (Xor2 (Delay (Xor2 (Delay (...
```

There are two common solutions to the problem of infinite computation resulting from circular definitions. One solution is to use monads (or similar categorical structure) to wrap the result in a way that circularity becomes observable. This is the approach taken by Singh [2]. Using monads has the advantage that it avoids unsafe Haskell constructs and can be expressed in idiomatic Haskell. The disadvantage is that the type of parity changes, as well as the specific form of the specification body, compromising the declarative flavor of the hardware description.

A second solution to reifying specifications like `parity` relies on the fact that the internal definition of `parity` is represented using a cyclic graph structure. With the ability to observe sharing [3,4] we can extract these cycles, though we need to be careful not to lose some equational reasoning options. In practice, observable sharing does not interfere with Haskell's pure idioms, and is arguably more declarative.

Both solutions for resolving cycles result in a netlist structure of gates and wiring. From this netlist, generating VHDL is straightforward. Lava becomes a macro language inside Haskell for writing combinational and sequential circuits. A VHDL synthesizer compiles the generated VHDL to implementations in FPGA or another silicon technology. The Lava concept has been both influential and successful. Lava has been used to build a number of FPGA based hardware solutions [2,5], and has also had tremendous success in helping teach hardware design at both the graduate and undergraduate level [6].

2 Kansas Lava

Kansas Lava is an effort to extend the Lava design pattern with modern functional programing technology. In particular, we attempt to scale up the ideas in Lava to operate on larger circuits and with larger basic components.

- Kansas Lava uses a single `Signal` type for all types of signals. Some versions of Lava use overloading to interpret constructs in either a synthesis or simulation mode. Our experience is that a single concrete type is easier to work with in practice, and we have included the two main interpretations into our `Signal` type. Ultimately this allows a closer fit between our specifications of behavior and synthesizable code. We give an example of this process in section 8.
- Like other Lava implementations before it, Kansas Lava supports both synthesis and simulation. This supports a workflow where first a simulation model is developed, then refined to a synthesizable variant, then further refined to satisfy performance constraints.

- Kansas Lava uses modern Haskell idioms. We define `Signal` as an applicative functor [7]. Arithmetic is overloaded over `Signal`, so we can use standard arithmetic operators and represent constants. This leads to simpler and more Haskell-like circuit specifications.
- Kansas Lava has direct support for including new blocks of existing VHDL libraries as new, well typed primitives. This allows Kansas Lava to be used as a high-level glue between existing solutions.
- Kansas Lava includes simple type checking over binary representations. What might be used as a single polymorphic entity inside Lava will be instantiated to a specific, monomorphically sized implementation in VHDL. This type checker is lightweight, as described in section 6.
- Kansas Lava uses an implementation of sized types, built using type functions. This library includes sized 1 and 2 dimensional matrices, along with sized signed and unsigned numerical representations. In Haskell, requiring a 14-bit unsigned value is unusual, but in hardware, we often know and want to enforce a specific width and format. We describe our sized type mechanism in section 4.

The primary contribution of our work so far is bringing together all the above elements into a single modern framework. One of our target applications – wireless communication circuits – makes heavy use of matrix operations to express encoding and decoding mechanisms [8], so we pay careful attention to support a straightforward encoding of such operations. In particular, the use of type functions to implement sized types makes matrix operations clear and straightforward. Furthermore, we believe our use of sized types for both ranged values and indices is novel and useful when specifying hardware.

3 `Signal` for Synthesis and Simulation

Building up small Lava circuits for bit-level operations is a well understood process. One aspect that is unusual about Kansas Lava is the coupling between the model and the synthesizable circuit, which both are embedded in the single `Signal` type. In this section, we introduce the Kansas Lava `Signal` type and give examples of its use.

A `Signal` is a value that is typically represented by a `signal` in VHDL, and implemented by a physical vector of wires. A `Signal` of a specific type represents an infinite sequence of elements of this type. Semantically, we model `Signal` as

$$\texttt{Signal}\ \alpha = Nat\ \rightarrow\ \alpha$$

where Nat is a clock cycle count.

Kansas Lava provides basic primitives that act over types such as `Signal Bool`. For example, a simple `xor` over two signals has the type

```
xor2 :: Signal Bool -> Signal Bool -> Signal Bool
```

Literally, `xor2` takes two signals of boolean values, and return a signal of booleans. The half adder from section 1 can be given the descriptive type

```
halfAdder :: (Signal Bool,Signal Bool) -> (Signal Bool, Signal Bool)
```

We denote an infinite stream of boolean values using `low` or `high` for streams of `True` and `False` respectively.

```
low  :: Signal Bool
high :: Signal Bool
```

Many interesting signals can be constructed; for example a multiplexer

```
mux2 :: Signal Bool -> (Signal a, Signal a) -> Signal a
```

where the `mux2` selects between two signals based on a boolean signal argument.

Furthermore, `Signal` is an applicative functor [7]. The applicative functor provides a clean way of expressing computation over time-varying streams of values. For `Signal` the applicative functor operators have the type

```
pure  :: a -> Signal a
(<*>) :: Signal (a -> b) -> Signal a -> Signal b
(<$>) :: (a -> b) -> Signal a -> Signal b
```

We can generate infinite sequences of a single specific value, and we can merge a sequence of functional values with a sequence of arguments using individual applications, giving a sequence of results. This raises the question of how to realize a `Signal` of functional values in hardware.

`Signal` α is a dual representation, combining an infinite sequence of α values with a deep embedding of the structure of computation. With this shared representation, all synthesizable circuits can be simulated directly, but not all circuits can be synthesized.

In Kansas Lava, the distinction between synthesizable and non-synthesizable hinges on the presence of applicative functor constructs. The applicative functor provides a convenient interface for specifying behavior but is unsuitable for synthesizable circuits. This distinction induces the design flow illustrated in figure 1. We start with a Haskell model, then use applicative functors to rebuild the model in a way that understands time, then we factor out the applicative functor, where the remaining circuit is now synthesizable.

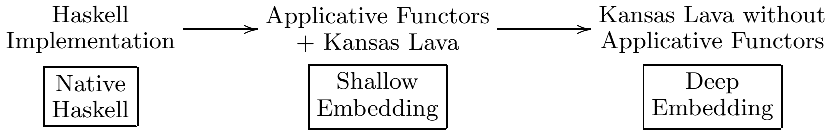
Haskell Implementation
Applicative Functors + Kansas Lava
Kansas Lava without Applicative Functors
Native Haskell
Shallow Embedding
Deep Embedding

Fig. 1. Kansas Lava Design Flow

4 Sized Types and Sized Matrices

Many specifications of behaviors for error correcting codes are expressed in terms of matrices. In this section, we describe matrices in Kansas Lava, which use a sized type implementation to catch size mismatches statically. The basic type of a matrix is `Matrix ix a`, where `ix` is a type encoding of the *size*, and `a` is the type of elements of the matrix. A vector of boolean values of size 32 has the type `Matrix X32 Bool`. The sized types from `X1` to `X256` are provided, and other larger numbers are straightforward to construct. We consider matrices as a hybrid between lists and tuples. Like tuples, the type completely determines the number of elements that the matrix contains, and like lists every element in the matrix has the same type.

Matrices are created by coercion from a standard list.

```
> :t matrix
matrix :: (Size i) => [a] -> Matrix i a
> matrix [1..4] :: Matrix X4 Int
[ 1, 2, 3, 4 ]
```

When creating a matrix, you must specify the type. We display matrices as list of elements, with traditional spacing between elements. These matrices are functors, so we can `fmap` (functor map) over these matrices.

```
> let m = matrix [1..4] :: Matrix X4 Int
> fmap (*2) m
[ 2, 4, 6, 8 ]
```

Kansas Lava also supports multi-dimensional matrices. As in the case of single dimension matrices, we create them from a *flat* list, with the size determining the partitioning of the input list.

```
> matrix [1..12] :: Matrix (X3,X4) Int
[ 1,  2,  3,  4,
  5,  6,  7,  8,
  9, 10, 11, 12 ]
```

For two dimensional matrices, the show routine renders the matrix as a single list, using layout to denote the partitioning. From this basis, we can build combinators to operate over matrices, performing functions like transpositions, splicing, and joining.

Incorporating matrix sizes allows general functions to be defined. For example, `identity` creates an identity matrix.

```
> :t identity
identity :: (Size x, Num a) => Matrix (x, x) a
> identity :: Matrix (X4,X4) Int
[ 1, 0, 0, 0,
  0, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 0, 1 ]
```

The type of `identity` states that the result must be a square matrix, and the elements must admit `Num`.

The $\texttt{X}_n$ type has a concrete realization in the form of a natural between 0 and $n-1$. This represents the range of possible valid index values for a specific array. The `ix` type is not just a phantom type [9] to represent size, it also is one of the ways we index into a `Matrix`. Indexing has the type

```
(!) :: (Size ix) => Matrix ix a -> ix -> a
```

Sized types are useful for ensuring consistences between the matrix sizes. For example, the definition of matrix multiply is

```
mm :: (...) => Matrix (x,y) a -> Matrix (y,z) a -> Matrix (x,z) a
mm a b = forAll $ \ (i,j) -> sum [ a ! (i,r) * b ! (r,j) | r <- all ]
```

The type captures exactly the requirement that the number of columns in the first matrix must match the number of rows in the second matrix. The `forAll` function creates a new matrix from a function that takes a matrix index and returns the element at that index.

Sized types allow computation on types. The type of `beside`, which places two matrices side by side is

```
beside (...) => Matrix (x,y1) a -> Matrix (x,y2) a -> Matrix (x,y3) a
```

Ignoring the type constraint for a moment, an example of its use is

```
> let i = identity :: Matrix (X4,X4) Int
*Main> i ‘beside‘ i
[ 1, 0, 0, 0, 1, 0, 0, 0,
  0, 1, 0, 0, 0, 1, 0, 0,
  0, 0, 1, 0, 0, 0, 1, 0,
  0, 0, 0, 1, 0, 0, 0, 1 ]
```

So what is the actual type of `beside`, including class constraints? Consider possible ways we may use sizes at compile time.

– We want to know the total size of the result matrix, if we know the sizes of the two arguments.
– Alternatively, we want to be able to infer the type of a specific argument, if we know the size of the other argument and the result.

We provide these capabilities using explicit type functions [10], providing an `ADD` and `SUB` at the type level. These are type *functions* and provide inference in a single direction. For example, the type `ADD X2 X3` maps to `X5`.

For each of the arguments and the result, we provide a single type function that can compute the sized type, given the following type to `beside`.

```
beside
  :: ( Size m, Size left, Size right, Size both
     , ADD left right ~ both
     , SUB both left ~ right
     , SUB both right ~ left
     ) => Matrix (m, left) a -> Matrix (m, right) a -> Matrix (m, both) a
```

In English, this means that `m`, `left`, `right`, and `both` are all sized types. The result column count `both` is the sum of the `left` column count and the `right` column count. The `~` operator indicates type equality.

Given the ability to represent sizes at the type level, we can use them for both the sizes of matrices as well as the size of numerical representations. Haskell libraries provide a small number of specifically sized signed and unsigned representations, but often in circuits more precise control of sizes is required, such as `Int9` (a signed 9-bit integer) or `Word34` (an unsigned 34-bit integer). Sized matrices are used to encode arbitrarily sized signed and unsigned numbers.

```
data Signed ix = Signed (Matrix ix Bool)
data Unsigned ix = Unsigned (Matrix ix Bool)
```

In Kansas Lava, we provide instances of `Num`, `Enum`, `Bits`, and other standard classes for both `Signed ix` and `Unsigned ix`. The implementation given here is the specification; for efficiency we utilize an underlying encoding of `Signed` and `Unsigned` using an `Integer`. The `Signed` and `Unsigned` types provide a standard interface for modular arithmetic in Haskell.

```
> let x = 100 :: Signed X8
> x
100
> x + 1
101
> x * x
16
```

5 Sized Types and Hardware Representations

In Kansas Lava, like many Haskell programs, types reveal a great deal about implementation. Kansas Lava uses a small set of types from which all circuits are constructed. Table 1 gives a list of the basic types used. For these types, we have selected specific VHDL implementations.

– `Signal Bool` is a boolean that changes over time. In hardware, it is represented by a single wire, either for control or data.

```
        i0 : in std_logic
```

– `Signal (Unsigned ix)` is a unsigned number.

```
        i0 : in std_logic_vector(ix-1 downto 0);
```

– `Matrix ix1 (Signal (Unsigned ix2))` is a group of signals, where each signal represents a signed number that changes over time. In addition, we know the number of elements in this group statically.

```
        i0,i1,i2,...,iix1 : in std_logic_vector(ix2-1 downto 0);
```

Table 1. Types in Kansas Lava

Type	Hardware Representation
`Signal` α	Clocked value of type α that changes over time
`Matrix` ϕ α	1 dimensional matrix of α with ϕ elements
`Matrix` (ϕ_1, ϕ_2) α	2 dimensional matrix of α with ϕ_1 column, and ϕ_2 rows.
`Signed` ϕ	Signed number represented using ϕ bits.
`Unsigned` ϕ	Unsigned number represented using ϕ bits.
`Bool`	a boolean; `True` or `False`.

– `Signal (Matrix ix1 (Unsigned ix2))` is a *single* time-varying value, that represents a group of unsigned numbers. It is represented by a single signal of size `MUL ix1 ix2`, which is the concatenation of the matrix elements.

```
        i0 : in std_logic_vector(ix1 * ix2 - 1 downto 0);
```

In this way, the choice of type at the Lava level is directly reflected into the choice of type at the VHDL level. The user can tune their representation choice to take into account the Haskell level model and the interface required in VHDL.

6 Implementation of Kansas Lava

The implementation of Kansas Lava follows the patterns of its predecessors by using a deep embedding of `Signal` to represent the circuit shape. We use IO-based observable sharing [4], and make heavy use of overloading. We currently have three back ends: VHDL, schematic, and a debugging output format. All three back ends share the same reification implementation and type inference mechanism for VHDL-level signals.

The `Signal` type is a tuple of a shallow and deep embedding.

```
data Signal a = Signal (Seq a) (Driver E)
```

The shallow embedding is a `Seq`, which is an infinite stream with the ability to include unknown values, and includes an optimization for constant streams.

```
data Seq a = (Maybe a) :~ (Seq a)
           | Constant (Maybe a)
```

`Seq` is an applicative functor, so the standard applicative functor interface can be used to construct shallow embedded behaviors.

A `Driver` is a data structure that represents a wire, which may be an (named) output from another entity, a global input pad, or a constant integer.

```
data Driver s = Port Var s
              | PathPad [Int]
              | Lit Integer
```

The type E is a wrapper around an Entity, used to simplify reification.

```
data E = E (Entity (Ty Var) E)
```

Entity is the central data type inside our embedding.

```
data Entity ty s = Entity Name [Var] [(Var,Driver s)] [[ty]]
```

An Entity is a globally scoped name representing a specific function, a list of named output ports, a list of input ports with the values that drive them, and finally some type information about the entity. An entity inside Kansas Lava corresponds one-to-one to an entity in VHDL, though we do not choose to implement all our entities this way.

Consider xor2 high (bitNot low), which has two entities xor2 and bitNot. The result Signal becomes a tree of entities.

```
Signal (Constant (Just False))
   (Port "o0" (E (Entity "xor2"
                         ["o0"]
                         [("i0",Lit 1),
                          ("i1",Port "o0" (E (Entity "bitNot"
                                                     ["o0"]
                                                     [(i0,Lit 0)]
                                                     [...]))))]
                        [...]))
   )
```

This shallow and deep embedding inside Signal is constructed by having every primitive operator split Signal into shallow and deep components, performing operations on each, then rebuilding a new Signal.

We observe sharing over the E type using the data-reify package. The reify function has the type

```
reify :: E -> IO (Graph (Entity (Ty Var))

data Graph e = Graph [(Int,e Int)] Int
```

We reuse the Entity type after reification, where Drivers are replaced with node identifiers representing the driving node. This structure is a netlist, and variants of this exist in all the recent variants of Lava. VHDL generation is simply a pass that traverses the netlist graph, mapping each Entity to either a VHDL entity instantiation or an equivalent behavioral expression. Figure 2 in section 7 shows an example fragment of VHDL generated from Lava.

6.1 Type Inference for VHDL Generation

Performing reification allows us to observe cycles in a circuit specification, which is necessary to generate VHDL from the Kansas Lava deep embedding. Unfortunately, the ability to observe the circuit structure doesn't provide sufficient information needed to generate VHDL. This is because the sized type information—from which it is possible to derive VHDL signal widths—is not maintained in the deep embedding.

Kansas Lava sized types ensure that a circuit is *constructed* correctly, but when decomposing the deep embedding of the circuit, this information has been discarded. The representation of an `Entity` maintains all of the input drivers for the `Entity` in a single homogeneous list, which requires the type parameter for `Signals` to be removed. By not maintaining type information in the deep embedding, we gain flexibility and the ability to include externally-defined VHDL entities without having to adapt the deep embedding data structure. On the other hand, throwing away the type information requires us to reconstruct the information to generate VHDL. This reconstruction is performed in parallel with reification. Type reconstruction is performed as an inference step, implemented using equivalence sets of types, where the equivalence is dictated by the Haskell types; a straightforward implementation of substitutions and their unification.

As an example, consider the generation of type equivalences for the polymorphic `mux2` function.

```
mux2 :: Signal Bool -> (Signal a,Signal a) -> Signal a
```

Assuming the inputs are called “cond” (the `Signal Bool`), “a” and “b”, and the output is called “r”, the inference algorithm infers the following partition. The second element in the outer set indicates that the signals “a”, “b” and “c” all inhabit the same type equivalence set.

$$\Big\{ \{ \texttt{bit}, \text{cond} \}, \{ \text{a}, \text{b}, \text{r} \} \Big\}$$

Given the equivalence relation for a single entity, the equivalence relation for an entire circuit is constructed by iteratively merging equivalence classes. If the input for one entity is connected to the output of a second entity, the equivalence classes of the input and the output are merged.

This process repeats for each input/output connection. When the inference has completed, each equivalence class should have a single ground (i.e. monomorphic sized) type, which is then assigned as the type to all of the nodes within that equivalence class. If an equivalence class contains *no* ground types, then the circuit is polymorphic, which we report as an error. This can happen if there is an unused loop of signals with no connecting type “edges” to give the loop a type.

7 Representing Addressable Memory

The Kansas Lava `delay` construct is used to create registers. When generating VHDL, Kansas Lava will represent these registers as primitive flip-flop elements. Each delay element will result in a collection of primitive flip-flops of the requisite width. While modern FPGA fabrics are reasonably register-rich, there remains a limit on the number of registers available. Moreover, a register is only capable of storing a single value: to represent addressable memory as registers, it is necessary to construct the address decoding logic as a multiplexer in the Lava design. Furthermore, each element in the address space will consume a number of flip-flops resources equal to the data width stored in the memory. As a consequence, memories with an address space of even moderate size represented in this way can quickly consume the available register resources.

As an alternative to distributed memory implemented using flip-flops, most FPGAs also contain a number of dedicated components which allow that allows the implementation of larger memories without consuming register resources. These memories, termed BRAMs in Xilinx technical documentation, are less plentiful than flip-flops, yet allow for the implementation of a smaller number of larger memory elements.

In addition to the restricted number of dedicated BRAM resources, these elements exhibit a different timing behavior than distributed RAM. For example, BRAM reads take two clock cycles, as compared to the single-cycle read time of registers. The two-cycle latency of BRAMs complicates their use in Lava designs, which—when restricted to using only delays for memory elements—has a purely synchronous stream semantics, with values produced on every clock cycle. The addition of BRAM elements complicates this semantics, due to the introduction of a read latency. While the performance impact of this latency can be minimized, as reads can be pipelined, an engineer using Lava must take extra care to account for read latencies when designing circuits.

7.1 Modeling Memories in Kansas Lava

Kansas Lava models a BRAM as a function mapping a memory operation to a value. A memory operation can either be a read, containing an address, or a write, containing both an address and a value to be written.

```
data MemOp a d = R a | W a d
type Memory a d = Signal (MemOp a d) -> Signal d
```

Kansas Lava implements this memory model in the shallow embedding of Kansas Lava using a Haskell `Map` for storing memory contents and a queue of values that captures the read latency of BRAMs. The single-step interpretation of a memory operation is shown in the `memop` definition below.

```
type MemRep a d = (Map a d, [d])
memop :: Ord a => MemRep a d -> MemOp a d -> (MemRep a d,d)
memop (m,ds) (R a) = ((m,vs),v)
  where val = M.lookup a m
        (v,vs) = dequeue (enqueue val ds)
memop (m,ds) (W a d) = ((m',vs),v)
  where m' = M.insert a d m
        (v,vs) = dequeue (enqueue 0 ds)
```

The `memop` function takes a map (`m`), a queue of delayed values (`ds`), and a memory operation. The function returns a new map and queue, along with a value. This single-step memory interpretation can be lifted to a `Seq`–based interpretation by using an accumulating map over `Seq`s.

The deep embedding of a BRAM element is implemented as a special `Entity` that takes a single input (the memory operation) and generates a single output. When rendered to VHDL, the memory operation for a BRAM with an address of a bits and with a data value size of d bits will result in an $(a + d + 1)$ bit signal. The least significant bit represents the BRAM write-enable signal, while bits a downto 1 represent the address for both reads and writes, and bits $a+d+1$ downto $a+1$ represent the data value for a write. Kansas Lava provides `readMem` and `writeMem` functions which perform the relevant bit packing, as there is currently no way to directly represent Haskell data values in VHDL. These functions are lifted to the `Signal` type, using the Haskell constructors for the shallow embedding and performing bit concatenation for the deep embedding.

The deep embedding of a memory component is rendered to behavioral VHDL, as described by the Xilinx synthesis documentation, rather than directly instantiated. Figure 2 shows a fragment from the VHDL produced for an 8-bit address $\times$ 8-bit data BRAM. The type `sig_o_2_ram_type` declares a VHDL array type of the requisite size, used by the BRAM signal `sign_o_2_ram`. The `sig_o_5` assignment represents the packing of a read operation (with zeros for the high data bits and the least-significant write-enable bit). The address signal, `i2`, is exposed as an input port to the enclosing VHDL entity, which is not shown. The signal assignments `sig_o_7`, `sig_o_6`, and `sig_o_4` perform the bit slicing from elements of the memory operation. In the synchronous `synch` process, the bit `sig_o_4` determines if the memory is written or read.

8 A Extended Example of Kansas Lava

We are using Kansas Lava to construct hardware implementations of communication circuits for forward error correction (FEC) codes over wireless fading channels. One component in a (FEC) circuit, an interleaver, performs a reordering of coded bit sequences to mitigate the effects of burst channel noise.

Coded bits within a communication frame are transmitted out-of-order, which allows bit errors due to short noise bursts to be distributed in reasonably even fashion across the frame. This makes it less likely that adjacent coded bits will be corrupted, a condition from which it is challenging to recover the intended bit transmission using our chosen error correction scheme. In the implementation of the interleaver, we use a *permutation* on the order of bits in the sequence to be communicated.

The permutation f is applied in the transmission circuit, and the inverse permutation f^{-1} is applied in the receiving circuit. Permutations are applied on a per-frame basis, the domain (and range) of the permutation function operates over a finite domain. We can model the permutation f as a mapping from logical bit address to transmitted bit address.

Representing such a permutation as a function is a challenge due to the requirement that the permutation appear random. In general, a random permutation cannot be described in more compact form than just enumerating the input to output address mapping. Moreover, the particular properties of the communication channel may impose additional characteristics on the permutation, for

```
signal sig_o_2 : std_logic_vector(7 downto 0);
signal sig_o_7 : std_logic_vector(7 downto 0);
signal sig_o_6 : std_logic_vector(7 downto 0);
signal sig_o_4 : std_logic;
signal sig_o_5 : std_logic_vector(16 downto 0);
type sig_o_2_ram_type is array(0  to 255) of std_logic_vector(7 downto 0);
signal sig_o_2_ram : sig_o_2_ram_type;
begin
  sig_o_5 <= to_unsigned(0,8)&unsigned(i2)&to_unsigned(0,1);
  sig_o_7 <= sig_o_5(16 downto 9);
  sig_o_6 <= sig_o_5(8 downto 1);
  sig_o_4 <= sig_o_5(0);
  synch: process (clk,i1,sig_o_4,sig_o_7) is
    begin
      if rising_edge(clk) then
          if sig_o_4='1' then
              sig_o_2_ram(conv_integer(sig_o_6)) <= sig_o_7;
              sig_o_2 <= (others => '0');
          else
              sig_o_2 <= sig_o_2_ram(conv_integer(sig_o_6));
          end if;
      end if;
    end process;
```

Fig. 2. VHDL generated for memory components

example, that adjacent input bits be separated by a minimum distance in the transmitted sequence. These requirements combine to make an algorithmic definition of the permutation difficult, if not impossible.

We have developed a general interleaver circuit that utilizes a BRAM component to implement the permutation, as shown in figure 3. A user describes the permutation as a list of pairs `[(Addr,Addr)]` mapping input address to output address. The inverse permutation is constructed by reversing the order of the pair elements. The circuit initializes a BRAM with the contents of this mapping. As bits arrive in sequence, a counter provides an input address to the ROM, which will yield the address in the permuted frame where the bit should be positioned.

The input bit is written to a buffer at the generated address. To allow the circuit to continually generate permuted values, the circuit uses a double-buffering technique, where the input bits for one frame are written to a BRAM, while at the same time the permuted input bits for the *previous* frame are read. In this way, the circuit allows permutation phases to be pipelined, with a one-frame latency.

In the circuit schematic, the `toggle` counter will change at every rollover of the address counter. The toggle output is connected to multiplexers in front of each buffer BRAM. In one mux, a high toggle output will select the read operation for the buffer, while a low toggle output will select the write operation. For the other mux, the selection is reversed. Finally, a multiplexer connected to the output ports of the BRAMs will select the output from the buffer that is currently being read, based on the toggle output.

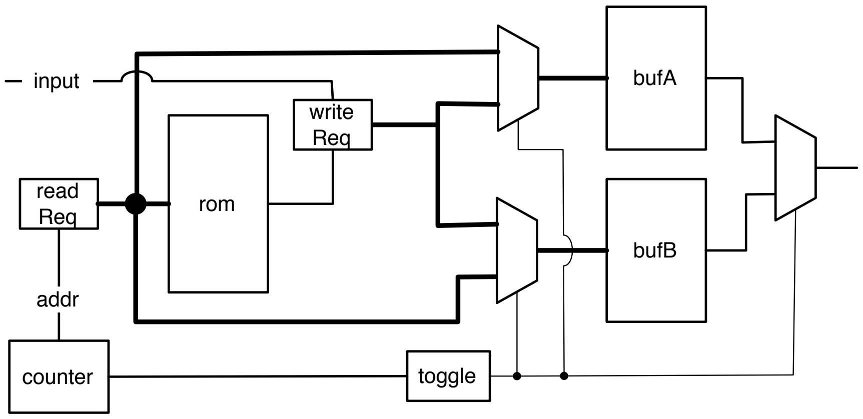

Fig. 3. Permutation Circuit Schematic

```
permute :: (...) =>  [(d, d)] -> Time -> Signal Bool -> Signal Bool
permute permutation clk input = out
  where out = mux2 toggle_z_zz bufA bufB
        addr = counter clk
        permRead = readMem addr

        rom = bram permutation clk permRead

        writeReq = writeMem rom input_zz
        readReq = readMem addr_zz

        muxA = mux2 toggle_z readReq writeReq
        muxB = mux2 toggle_z writeReq readReq

        bufA = bram initBuf clk muxA
        bufB = bram initBuf clk muxB
        initBuf = [(i,False) | i <- [minBound..maxBound]]
        toggle = delay clk high (toggle `xor2` overflow)
           where overflow = (addr .==. 0)

        addr_zz = delayN 2 clk addr
        input_zz = delayN 2 clk input
        -- the 'toggle' has a built-in 1-cycle delay,
        -- so we only delay 1 cycle
        toggle_z = delayN 1 clk toggle
        toggle_z_zz = delayN 2 clk toggle_z
```

Fig. 4. Permutation Circuit in Lava

Figure 4 shows the description of the permutation circuit in Kansas Lava. The `permutation` parameter defines the mapping from input bit address to output bit position. The `input` parameter is the frame bit input sequence, and the circuit output is the bit sequence of the permuted frame. Within the circuit definition, the `bram` function instantiates a BRAM with the supplied contents. The remainder of the internal signals corresponds to those in the figure.

This Lava definition demonstrates a complication of using memory components. It is necessary to manually add `delay` components to compensate for read latencies. This is indicated in the definitions by a `_z` suffix to a signal name. The result of the ROM read introduces a 2-cycle latency, so it is necessary to add two delays in the `input` signal to insure that data values line up with write addresses. Similarly, the `toggle` signal is delayed two cycles for the input to the memory read/write operations, and then a further two cycles for the buffer read output, since that too introduces a two-cycle delay.

9 Related Work

The idea of having a program that generates code is an old one, as are the ideas behind Lava itself. The original ideas for Lava can be traced back through work on the Ruby [11] hardware description language and prior to that, μFP [12]. Both of these rely upon the close similarity of circuits and functional languages. Both involve taking input and returning output, and both can represent state (using registers or using streams) as feedback loops. A good summary of the principles behind Lava specifically can be found in [1].

ForSyDe [13] is a system that is close in spirit to Kansas Lava. Like Kansas Lava, ForSyDe is intended to support the modeling of system level concerns, and is also embedded in Haskell. Kansas Lava circuits are clocked circuits, with all `Signal` computations based on a stream-based model of computation. ForSyDe offers support for several additional models of non-terminating computation in addition to clocked synchronous `Signals`. ForSyDe directly supports both a shallow and deep embedding of `Signals`. The two embeddings provided as separate implementations with the same interface, and the ForSyDe programmers can use Haskell `import` directives to choose which to invoke. Finally, ForSyDe uses type classes (rather than the more recently developed type functions) to implement basic type-level arithmetic to model arbitrary sized wiring patterns. Kansas Lava's use of type functions allows for overloading of sized types and indexing. In particular, in Kansas Lava, the sizes are themselves first class values, and can be used for indexing, giving cleaner and lighter weight matrix specifications.

JHDL [14] is a hardware description language, embedded in Java, which shares many of the same ideas found in Lava. In JHDL, structural circuits are straightforward to express by writing stylized Java programs, and the computational mechanisms provided by Java can be productively used when generating these structural circuits.

There are many other hardware description languages that either use a functional language, or have a functional basis. We refer the reader to the comprehensive comparative review authored by the developers of ForSyDe [15].

10 Conclusions and Future Work

Kansas Lava represents a continuation and expansion on the heritage of Lava. The inclusion of sized types provides a straightforward embedding of a low-level hardware concern within Haskell, leveraging modern facilities provided by Haskell extensions.

Prior Lava work has identified a series of design decisions with associated trade-offs. These include various implementation strategies of shallow vs. deep embedding and of observable sharing. We have endeavored to choose an implementation strategy that leverages these design alternatives in a modular and general manner.

Kansas Lava is an ongoing and actively supported project. As such, we intend to continue to improve and extend the library to include more sophisticated data modeling capabilities. It is clear that developing efficient hardware requires close attention to the performance implications of implementation choices, which are often abstracted away in a functional language. We believe a principled approach to program manipulation [16], using type-based transformations that preserve computational behavior while modifying performance, has the potential to bridge the gap between specification and implementation by allowing an abstract specification to be incrementally refined to suitable implementations.

A further direction of inquiry involves the expression of more sophisticated control patterns in Kansas Lava. For example, the introduction of BRAM into Lava requires an engineer to manually manage deviations from the synchronous stream model of computation. Embedding timing properties within a Lava specification as types may allow the automatic insertion of control logic to compensate for timing incompatibilities. In addition to the case demonstrated by BRAMs, where timing latencies are statically known, this can be expanded to situations where timing properties are dynamic, allowing the inclusion of components such as SDRAM that exhibit non-deterministic timing behavior.

Acknowledgments

We would like to thank the members of CSDL at KU, and the IFL referees for useful and detailed feedback.

References

1. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware design in haskell. In: International Conference on Functional Programming, pp. 174–184 (1998)
2. Singh, S., James-Roxby, P.: Lava and jbits: From hdl to bitstream in seconds. In: FCCM 2001: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Washington, DC, USA, pp. 91–100. IEEE Computer Society, Los Alamitos (2001)
3. Claessen, K., Sands, D.: Observable sharing for functional circuit description. In: Thiagarajan, P.S., Yap, R.H.C. (eds.) ASIAN 1999. LNCS, vol. 1742, p. 62. Springer, Heidelberg (1999)

4. Gill, A.: Type-safe observable sharing in Haskell. In: Proceedings of the 2009 ACM SIGPLAN Haskell Symposium (September 2009)
5. Singh, S.: Designing reconfigurable systems in lava. In: International Conference on VLSI Design, p. 299 (2004)
6. Axelsson, E., Björk, M., Sheeran, M.: Teaching hardware description and verification. In: IEEE International Conference on Multimedia Software Engineering, International Symposium on Microelectronics Systems Education, pp. 119–120 (2005)
7. McBride, C., Patterson, R.: Applicative programing with effects. Journal of Functional Programming 16(6) (2006)
8. Moon, T.K.: Error correction coding: mathematical methods and algorithms. Wiley Interscience, Hoboken (2005)
9. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: 2nd USENIX Conference on Domain Specific Languages (DSL 1999), Austin, Texas, pp. 109–122 (October 1999)
10. Chakravarty, M.M.T., Keller, G., Jones, S.P.: Associated type synonyms. In: ICFP 2005: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming, pp. 241–253. ACM, New York (2005)
11. Jones, G., Sheeran, M.: Circuit design in ruby. In: Staunstrup (ed.) Formal Methods for VLSI Design. Elsevier Science Publications, Amsterdam (1990)
12. Sheeran, M.: mufp, a language for vlsi design. In: LFP 1984: Proceedings of the 1984 ACM Symposium on LISP and functional programming, pp. 104–112. ACM, New York (1984)
13. Sander, I.: System Modeling and Design Refinement in ForSyDe. PhD thesis, Royal Institute of Technology, Stockholm, Sweden (April 2003)
14. Bellows, P., Hutchings, B.: JHDL - an HDL for reconfigurable systems. In: Annual IEEE Symposium on Field-Programmable Custom Computing Machines, p. 175 (1998)
15. Jantsch, A., Sander, I.: Models of computation and languages for embedded system design. IEE Proceedings on Computers and Digital Techniques 152(2), 114–129 (2005); Special issue on Embedded Microelectronic Systems
16. Gill, A., Hutton, G.: The worker/wrapper transformation. Journal of Functional Programming 19(2), 227–251 (2009)