Declarative FPGA Circuit Synthesis using Kansas Lava

Andy Gill
Information Technology and Telecommunication Center
Department of Electrical Engineering and Computer Science
The University of Kansas
Email: andygill@ittc.ku.edu

***Abstract*—Designing and debugging hardware components is challenging, especially when performance requirements demands a complex orchestra of cooperating and highly synchronized computation engines. New language-based solutions to this problem have the potential to revolutionize how we think about and build circuits. In this paper, we describe our language-based approach to semi-formal co-design. Using examples, we will show how generative techniques, high-level interfaces, and refinement techniques like the worker/wrapper transformation can be used to take descriptions of specialized computation, and generate efficient circuits. Kansas Lava, our high-level hardware description language built on top of the functional language Haskell, acts as a bridge between these computational descriptions and synthesizable VHDL. Central to the whole approach is the use of Haskell types to express communication and timing choices between computational components. Design choices and engineering compromises during co-design become type-centric refinements, encouraging architectural exploration.**

I. Introduction

Generated VHDL offers many advantages over hand-written VHDL. Kansas Lava is a framework for expressing synthesizable hardware and generating VHDL, with the express purpose of exploring computational models for describing hardware level concerns. In this paper, we will describe the benefits of generated code, supported models of computation, and a look at a number of applications we have constructed using Kansas Lava.

Reconfigurable computing and co-design provides the opportunity to be flexible about how and where something is computed. The larger fabrics provided by modern FPGAs open the option of non-specialists writing VHDL programs, much in the same way early microprocessors allowed assembly language programming for efficient numerical computation. For highly customized parallel computations, well designed FPGA fabrics can perform at breathtaking scale, unmatched even by modern multi-cores. Yet tool support for migrating highly-computational tasks to IP cores on FPGAs is rudimentary, at best, and from a tool perspective, certainly comparable with early microprocessor development tools.

When considering generating circuit descriptions, VHDL and Verilog are sibling languages, at least from a language semantic point of view, and many FPGA tools support both as input to be compiled for FPGA fabrics. In this paper, we will examine a VHDL generator, and use VHDL as our language in examples. The same ideas could be retargeted to use Verilog with only nominal engineering effort.

VHDL provides both a way of expressing structure, or connections between existing components, as well a Register Transfer Level (RTL) style of expressing how a component acts based on register contents and a state machine. Both are used in practice by engineers programming FPGAs. There are other, more-advanced computational programming models, for example the atomic transaction model used by BlueSpec [1]. Kansas Lava provides a straightforward way of expressing the basic model of connected components, and a highly customizable methodology for providing RTL and other models on top of the connectivity. In this way, Kansas Lava becomes as test bed of ways of customizing models for expressing solutions to hardware-level problems. Ultimately, the vision is having custom languages, tailored to specific problem classes, that allow experts in the problem classes to program FPGAs effectively.

Hardware components communicate using buses and wires. VHDL uses a signal abstraction to represent almost all such communications. As such, a VHDL signal is fundamentally physical. Using Kansas Lava, we can explicitly represent higher-level communication abstractions, including partial values, bus protocols, and wiring inside multi-clock or hyper-clocked environment. This richness and depth of communication options brings opportunities that would be completely unreasonable to expect an engineer to directly manage; the Kansas Lava abstractions build on top of primitives provided in VHDL instead.

The analogy of VHDL being an assembly language for FPGAs is an timely one. As a community, we are providing new computational and communication abstractions for FPGA programmers, and hope it will lead to a critical mass of useful applications and happy users. In this paper, we will overview our specific system, Kansas Lava, and explain how it both helps users and provides abstraction opportunities. We will not go into details about the *internals* of Kansas Lava; we refer interested readers to our earlier explanations [2], [3] for specifics. Instead, we focus on using Kansas Lava. To do so, some knowledge of functional programming and Haskell [4] is needed, because this is the language-based foundation on which we build our tools.

II. HASKELL

Haskell is the premier pure functional programming language. Haskell supports many forms of abstraction, and provides a robust foundation for building Domain Specific Languages (DSLs). In this section, we give a terse introduction to Haskell, sufficient to make this paper self-contained.

Haskell is all about *types*. Types in Haskell, like types in other languages, are constraining summaries of structural values. For example, in Haskell `Bool` is the type of the values `True` and `False`, `Int` is the type of machine-sized words, `Double` is the type of double precision floating point values; and this list goes on in the same manner as `C`, `C++`, `Java` and other traditional languages. All these type names in Haskell start with an upper-case letter.

On top of these basic types, Haskell has two syntactical forms for expressing compound types. First, pairs, triples and larger structures can be written using tuple syntax, comma separated types inside parenthesis. So `(Int,Bool)` is structure with both an `Int` and a `Bool` component. Second, lists have a syntactical shortcut, using square brackets. So `[Int]` is a list of `Int`.

Haskell also has other container types. A container that *may* contain one `Int` has the type `Maybe Int` which is read `Maybe` of `Int`. These container names also start with upper-case letters. Types can be nested to any depth. For example, we can have a `[(Maybe (Int,Bool))]`, read as list of `Maybe` of (`Int` and `Bool`).

Polymorphic values, which are analogous to the type `Object` in Java, or `void*` pointers in C, are expressed using lower-case letters. These polymorphic values can have constraints expressed over them, using the Haskell equivalent of an object hierarchy. Finally, a Haskell function that takes a list, and returns a list is written as using an arrow: `[a] -> [a]`.

We can now give an example of a Haskell function.

```
sort :: (Ord a) => [a] -> [a]
sort []     = []
sort (x:xs) = sort before ++ [x] ++ sort after
  where
        before = filter (<= x) xs
        after  = filter (> x) xs
```

This function sorted a list, using a variant of quicksort in which the pivot is the front of the list.

- The first line is the type for `sort`. This is ∀`a`, such that `a` can be `Ord`ered (admits comparisons like <=), the function takes and return a list of such `a`'s.
- The second line says that an empty list is already sorted.
- The remaining lines state that a (non-empty) list can be sorted by taking the first and rest of the list (called `x` and `xs`, respectively), and sorting the values before this pivot and after this pivot, and concatenating theses intermediate values together.
- Finally, intermediate values can be named using the `where` syntax; in this case the values of `before` and `after`.

Haskell is a concise and direct language. Structures in Haskell are described using types, constructed and deconstructed, but never updated. The entire language functions by chaining together these structural processors, which ultimately take input, and produce output. Side-effects are described using a `do`-notation, for example:

```
main :: IO ()
main = do
  putStrLn "Hello"
  xs <- getLine
  print xs
```

In this example a *value* called `main` uses the `do`-notation to describe an interaction with a user. Actually, the `do`-notation captures this as a structure called a Monad; purity is not compromised. For more details about how `do`-notation and Monads can provide an effectful interface inside a pure language like Haskell, see [5]. For the purposes of Kansas Lava, `do`-notation is a way of providing syntax and structure that looks like interaction.

III. KANSAS LAVA PRIMER

Kansas Lava is a Haskell library that models circuits as well as generating VHDL. In this section, we introduce the two main value descriptor types, and system modeling capabilities of Kansas Lava, returning to to the actual *generation* VHDL in section VI.

A. Combinational Values

Operations on combinational values are the basis of hardware computations. Examples include an addition function operating over numerical values, or a nand gate operating over boolean values. To take an example, and code it in Kansas Lava, consider a half adder.

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

The type indicates that this function, `halfAdder` operates over two arguments, both `Comb Bool`, or combinational booleans. The result, the `carry` and `sum`, are return as a 2-tuple of `Comb Bool`. (The type would typically be written on a single line after the function name, but is folded over two lines to fit into a single column.) Inside the function definition, the intermediate values `carry` and `sum` are declared using the Haskell `where` keyword. This definition of a half adder is simple, direct, declarative and clear.

`Comb` is an *active* container for representing combinational values, where the type argument of `Comb` is the value being represented. `Comb Bool` is the type of a combinatorial boolean, in VHDL this would be a `std_logic`. `Comb Word8`, a combinatorial 8-bit value could be represented in VHDL using a `std_logic_vector`, or even an IEEE `signed`.

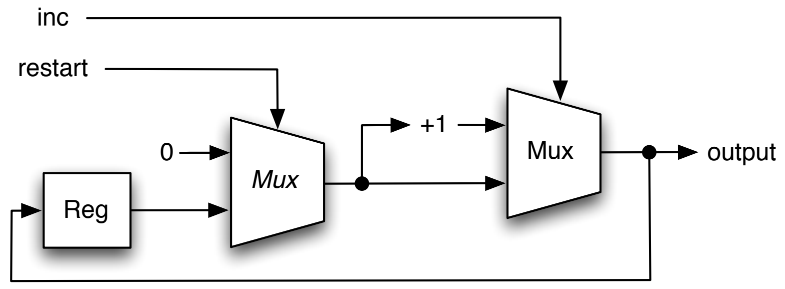

Fig. 1. `counter` schematic

Executing our half adder example (as a simulation) is as simple as calling the `halfAdder` function. (The user interaction is notated starting with the `GHCi>` prompt.)

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

In summary, `Comb` x represents a single value x, computed using a combinatorial circuit.

B. Sequential Values and Clocks

Combinatorial logic by itself is completely uninteresting, and needs multiple invocations on different values, constructing sequential circuits, to do useful computations. Kansas Lava has a type `CSeq`, a mnemonic for Clocked Sequence, for representing a stream of values created over time, using sequential logic. A specific clock is used to interpret when to sample the values communicated over a sequence. We do not assume any global clock, rather we use types to attempt to contain their interpretive utility.

There are two Kansas Lava functions that are the primitive sequential circuit builders, `register`, and `delay`.

```
register :: (Clock c, sig ~ CSeq c)
         => a -> sig a -> sig a
delay    :: (Clock c, sig ~ CSeq a)
         =>      sig a -> sig a
```

`register` is a classical "D" edge-triggered flip-flop, where the clock is implicit. `register` takes two arguments, the initial value (at startup and after a reset), and the input sequence. `delay` is a version of `register` where the initial value is intentionally undefined. The syntax for the constraint on the type of both `register` and `delay` states there is a clock called `c`, and there is a signal called `sig`, which is interpreted using this clock. In this way, the types state that the input and output of both functions are in the same clock domain.

As an example of using `register`, consider:

```
counter :: (Rep a, Num a, Clock clk, CSeq clk ~ sig)
        => sig Bool -> sig Bool -> sig a
counter restart inc = loop
   where reg = register 0 loop
         reg' = mux2 restart (0,reg)
         loop = mux2 inc (reg' + 1, reg')
```

This circuit connects two multiplexers, an adder, and a `register` to give a circuit that counts the number of clocked pulses on a the signal `inc`. The circuit takes two clocked signals, and returns a clocked signal that explicitly operates using same clock, because they share the same type. Figure 1 gives the circuit intended for this description.

We can execute sequential circuits with the same directness that combinational functions we invoked.

```
GHCi> toSeq (cycle [True,False,False])
T : F : F : T : F : F : T : F : F : ...
GHCi> counter low (toSeq (cycle [True,False,False]))
1 : 1 : 1 : 2 : 2 : 2 : 3 : 3 : 3 : ...
```

Both `Comb` and `CSeq` provide a direct representation of signals in VHDL, or more precisely, a meaning allowing interpreting a VHDL signal. Both `Comb` and `CSeq` also provide lifting, allowing the representation of unknown (in VHDL, `X`) values. On this basis, we build Kansas Lava.

C. Signals

In VHDL, signals are a generalization of `Comb` and `CSeq`. In Kansas Lava we allow a third class of value, a `Signal`, which is a Haskell class admitting both `Comb` and `CSeq`. The type for `and2` actually is:

```
and2 :: (Signal sig)
         => sig Bool -> sig Bool -> sig Bool
```

This use of `Signal` literally means either a `Comb` or a `CSeq` with a `Clock` can be used as a signal. In this way, `Signal` brings together combinational and sequential values, allowing overloaded primitives to be either.

Circuits are composed of sequential and combinational components, but values generated by sequential and combinational circuits have different types. We allow combinational circuits (which can only be run once) to be prompted, or lifted, into circuits that can be run sequentially many times. So we have

have primitive combinational operators, sequential operators like `register`, and a way of *lifting* combinator functions into either combinatorial *or* sequential functions.

IV. ROMs and Lifted Functions

As well as this trio of basic signal types, we can build circuits that operate on Haskell functions directly, provided the domain of the function is finite. We use the `Rep` constraint to signify that we can enumerate all possible representable values in a type, giving the funMap function.

```
funMap :: (Rep a, Rep b, Signal sig)
       => (a -> Maybe b)
       -> sig a -> sig b
```

The generated circuit is implemented using a ROM, and we can generate control logic directly in terms of Haskell functions and data-structures. As a example, consider a small ROM that stores the square of a value.

```
squareROM :: (Num a, Rep a, Signal sig)
                 => sig a -> sig a
squareROM = funMap (\ x -> return (x * x))
```

In this way, direct Haskell functions can be lifted into `Signal` world. Notice how `squareROM` function is not specific about size, but is completely generic, as long as the argument stream, of type "`a`", is representable as a number.

`funMap` used directly behaves as an asynchronous ROM. A synchronous ROM can be constructed out of a `funMap` followed by a `register` or `delay`. The specific decision *how* to represent a ROM is postponed to VHDL generation time. The simulator simply stores and looks up values using the function directly.

V. Structures and Types

Kansas Lava has some general purpose types that turn out to be especially useful with constructing descriptions of hardware. Specifically Kansas Lava has support for generically sized fixed-width signed and unsigned numbers, and generically sized fixed-width matrixes. We use a sized type [6], notated as $\mathtt{X}_n$, to annotate sizes onto types. Examples include `Unsigned X4`, a 4-bit unsigned number, `Signed X8`, a 8-bit signed number, and `Matrix X6 Bool`, a (one-dimensional) matrix with 6 elements that are of type `Bool`. Use of irregular-sized values is common in Hardware, while having a 14-bit adder is unheard-of in software. We also provide type shortcuts for common sizes, giving `U4` for the 4-bit unsigned number, and using $\mathtt{S}_n$ and $\mathtt{M}_n$ as names for signed numbers and matrixes, respectively.

On top of our signal types (`Comb`, `CSeq` and `Signal`) Kansas Lava builds a number of useful type-oriented utilities. The major one is the `pack` and `unpack` combinators. `pack` provides a way of taking a bundle of signals, and packing them into a single signal. `unpack` does the reverse operation, unpacking a single signal into a bundle of signals.

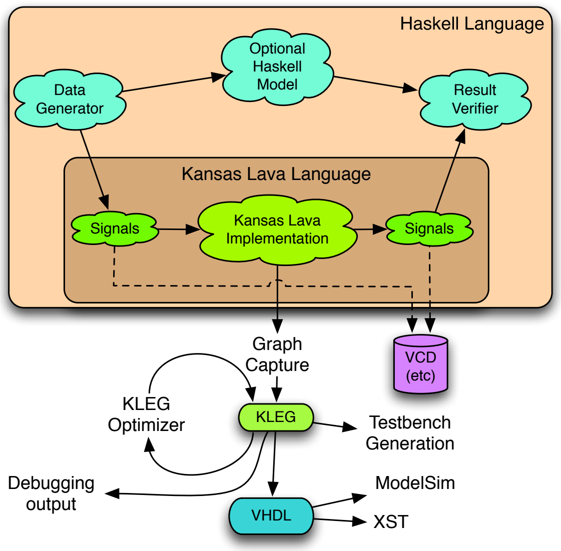

Fig. 2. Kansas Lava Architecture

As a concrete example, consider a pairing of a `Boolean` signal, and an 8-bit unsigned signal (`U8`), with the names `a` and `b`:

```
(a,b) :: (Signal sig) => (sig Bool, sig (U8))
```

We can use `pack` to pack this pairing into a single signal

```
pack (a,b) :: (Signal sig) => sig (Bool,U8)
```

As an example, we can define `(a,b)`, then pack the result inside the Kansas Lava simulation mode.

```
GHCi> let a = toSeq (cycle [True,False])
GHCi> a
T : F : T : F : T : F : T : F : ...
GHCi> let b = toSeq [1..]
GHCi> b
1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : ...
GHCi> pack (a,b)
(T,1) : (F,2) : (T,3) : (F,4) : (T,5) : ...
```

`pack` and `unpack` turn out to be extremely useful in practice, and support the objective of allowing highly flexible type representation for data flowing inside an implemented program.

Finally, there are structural types for expressing common protocols used in hardware. For example `Enabled a` means a value `a` that has an extra `Boolean` value signifying validity. Another example is `Write addr val`, which may contain a write request sent to a memory.

Overall, the type system and structures of the core Kansas Lava system described here provide a useful replacement for describing and simulating circuits, comparable to VHDL and ModelSim, but with richer and more descriptive types. Our earlier paper [2] gives detailed justification for each feature in this core from a language point of view. The interesting work is what can be built on top of this foundation.

```
inStdLogic         :: (sig ~ CSeq ())
                   -> String
                   -> Fabric (sig Bool)
inStdLogicVector   :: (Rep a, sig ~ CSeq ())
                   => String
                   -> Fabric (sig a)
outStdLogic        :: (sig ~ CSeq ())
                   -> String
                   -> sig Bool
                   -> Fabric ()
outStdLogicVector :: (Rep a, sig ~ CSeq ())
                   => String
                   -> sig a
                   -> Fabric ()
```

Fig. 3. Fabric API

VI. Generating VHDL

The main purpose of Kansas Lava is to generate useful VHDL. As the earlier examples illustrated, the Haskell system can be used to directly simulate circuit behavior. Taking this behavior, and correctly mapping it to efficient VHDL is done using a technique called reification.

Figure 2 gives an overview of the the major components of a successful invocation of Kansas Lava. A Kansas Lava program is executed in terms of generated test vectors, or signals, and these signals can be optionally compared with output generated by a high-level model. The same program can be extracted [7], without needing the specific signal context, and captured as a KLEG (Kansas Lava Entity Graph), which is for all intents and purposes a simple net-list.

If we take our `counter` example from above, the question is how to turn this into a VHDL architecture (from the *body* of the function), and VHDL entity (from the *type* of the function). In VHDL, all arguments in an entity are named and unordered, in Haskell and Kansas Lava, they are unnamed and order matters. Rather than just inventing names (which turns out to be brittle to even minor changes) we explicitly name arguments use a mathematical construction called a monad.

A monad is a way of expressing computation. For the purposes of the discussion of this specific monad, we can think of `Fabric`, the monad in question, as an abstract data structure. The interface for specifying inputs and output in VHDL is concise, and summarized in figure 3.

In figure 3 `inStdLogic` names an input, giving a `Fabric` that returns `Seq Bool`. `outStdLogic` names an output, and provides a `Seq Bool`, and returns a `Fabric`. `inStdLogicVector` and `outStdLogicVector` are versions of `inStdLogic` and `outStdLogic` respectively, that operate on a vector rather than an individual bit.

We can build a fabric around our counter, using the following fabric-building code.

```
entity counter is
  port(rst : in std_logic;
       clk : in std_logic;
       clk_en : in std_logic;
       restart : in std_logic;
       inc : in std_logic;
       output : out std_logic_vector(3 downto 0));
end entity counter;
architecture str of counter is
  signal sig_2_o0 : std_logic_vector(3 downto 0);
  ...
begin
  sig_2_o0 <= sig_5_o0 when (inc = '1')
                      else sig_6_o0;
  sig_5_o0 <= std_logic_vector(...);
  sig_6_o0 <= "0000" when (restart = '1')
                    else sig_10_o0;
  sig_10_o0_next <= sig_2_o0;
  proc14 : process(rst,clk) is
  begin
    if rst = '1' then
      sig_10_o0 <= "0000";
    elsif rising_edge(clk) then
      if (clk_en = '1') then
        sig_10_o0 <= sig_10_o0_next;
  ....
end architecture;
```

Fig. 4. Fragments of VHDL Generated for `counter`

```
counterFabric :: Fabric ()
counterFabric = do
        restart <- inStdLogic "restart"
        inc     <- inStdLogic "inc"
        let res :: (Clock clk) => CSeq clk U4
            res = counter restart inc
        outStdLogicVector "output" res
```

This use of fabric *must* clarify the specific types being operated on; in this case we are returning a signal of `U4`.

As well as building a Fabric, this is the Kansas Lava method for specifying port names. We can reify (capture) this `Fabric`, and the call to `counter` inside it, using a function called `fabricReify`.

```
fabricReify :: Fabric () -> IO KLEG
```

`KLEG` is an abstract representation of our net-list, and can be printed, optimized, or written into VHDL. This program is all the code needed to generate valid VHDL.

```
main = do
  k <- fabricReify counterFabric
  writeVHDL "counter" "counter.vhd" k
```

The contents of `"counter.vhd"` are shown in Figure 4.

The summary regarding VHDL generation is a simple narrative. We build a thin veneer around our circuit functions called a `Fabric`, which specifies how our circuits are invoked, and what to name our ports. These declarations, and the behavior of the underlying circuit, is realized in the form of a VHDL entity and architecture. As already stated, there is no reason that Verilog could not be generated instead; indeed we use a generic netlist rendering tool that can already target VHDL or Verilog.

```
parityCount :: (Clock clk, sig ~ CSeq clk)
            => sig DecodeCntl
            -> sig (Write SZ Bool)
            -> sig (Enabled U16)
parityCount control parityWrite = pack (stop,res)
  where (en,datam) = unpack parityWrite
        (addr,val) = unpack datam
        start = liftPred (== DecodeShare minBound)
                         (delay control)
        stop  = liftPred (== DecodeShare maxBound)
                         (delay control)
        res = counter start (en `and2` val)
```

Fig. 5. Parity Counting in LDPC

VII. Example: parity checking

To take a front to back example, consider a block of logic that listens on a channel for parity errors, counting them. We want to know if there were *any* parity error, but for debugging and error reporting, we report a parity error count. There is a control signal, typed `sig DecodeCntl`, which encodes when to start and stop counting parity counts. Figure 5 gives an example implementation, taken from an implementation of a LDPC forward error corrector [8].

There are a number of typical illustrative aspects to this example.

- The use of `pack` and `unpack` is an integral part of the description.
- The call to `counter`, our earlier example, can be seen as a simple function invocation on the final line.
- `liftPred` uses a Haskell-level equality test, checking to see if the control signal is at the minimum (start) or maximum (termination) delimiter.
- The output is only valid on the `stop` cycle, with validity communicated using a `pack`ed boolean value.
- Finally, the *type* is a detailed description of what this component requires. In this case, a control signal and a set of write "packets", giving a 16-bit result signal, with a validity bit attached.

We can now simulate and independently test this circuit, by creating two input streams. For this test, we construct a smaller control data-structure, where the minimum `DecodeShare` value and the maximum `DecodeShare` value is 3. We can now execute our example.

```
GHCi> let control = ...
            ++ [DecodeShare n | n <- [0..3]]
            ++ ...
GHCi> let writes = ...
           ++ [ Write (0,m)
              | m <- [True,False,True,False]
              ]
           ++ ...
GHCi> parityCount control writes
... : Nothing : Nothing : Nothing : Just 2 : ...
```

Once we are happy with this component, we can plug it into a larger Kansas Lava circuit, or render VHDL as a stand-alone VHDL component, by creating a `Fabric` round a call to `parityCount`.

VIII. Discussion

The core Kansas Lava system is an academic project, but mature and complete enough to engineer large examples. There are three principal things you can do with (a system like) Kansas Lava, beyond simply write code in Kansas Lava. You can (1) solve problems and express solutions using generative techniques; (2) build new models of computation and other abstractions on top of the existing interface; and (3) use the language as a target of refinements from a yet-higher level of specification of abstract behavior.

Generative Programming [9] is an idiomatic name for the general case of programs generating programs. In Kansas Lava, any circuit generator shares aspects of this powerful idiom. For example, a recursive divide and conquer algorithm can be used to generate a circuit, when the base-case generates small, hand crafted solutions, and the division invocations connect sub-circuits together. We discuss a specific example of using recursion for circuit generation in section X.

A generalization of these ideas is build new abstractions on top of the the basic syntax. When constructing circuits in VHDL, often a Register Transfer Level (RTL) idiom is used, where hardware behavior is expressed as clocked transitions between states with additional memory. In section IX we present an extension on top of Kansas Lava that provides an RTL interface abstraction, building on the Haskell `do`-notation.

Finally, Kansas Lava, with or without the generative programming extensions, can be a target language of a refinement methodology. A clear short description of an algorithm can be connected to a real implementation of a circuit, using a set of formal or semi-formal refinement steps. In section XI we discuss a larger example of showing a relationship between a specification and implementation discussed in section X.

IX. Example: Register Transfer Language API

Sometimes state-based thinking leads to clear descriptions of behavior. The core Kansas Lava system is stream based, with conditionals handled by `mux2` and related combinators. It is possible to build a state-based API that models RTL on top of core Kansas Lava.

Consider the problem of generating a fractionally rated boolean signal. This may be used, for example to trigger a sampling of an input port. When sampling higher-rate data transfers, the ratio of clock cycles to sample rate within acceptable tolerance might not have an integral reciprocal of a fraction. For example, when using a 50MHz clock, sampling a signal at 6MHz requires waiting $8\frac{1}{3}$ clock cycles between samples; clearly not possible. Instead, a sample rate of 6MHz overall can be achieved by waiting 8 *or* 9 cycles between samples, and averaging the gap to $8\frac{1}{3}$ clock cycles. This can be solved with Bresenham's Line Algorithm, which has to approximate a fractional rate of a line using whole pixels.

```
rate :: forall x clk . (Clock clk, Size x) => Witness x -> Rational -> CSeq clk Bool
rate Witness n
  | step * 2 > 2^sz = error $ "bit-size " ++ show sz ++" too small for punctuate Witness " ++ show n
  | n <= 0  = error $ "can not have rate less than or equal zero"
  | n > 1 = error $ "can not have rate greater than 1, requesting " ++ show n
  | otherwise = runRTL $ do
     count <- newReg (0 :: (Unsigned x))
     cut   <- newReg (0 :: (Unsigned x))
     err   <- newReg (0  :: (Signed x))
     CASE [ IF (reg count .<. (step + reg cut - 1)) $
              count := reg count + 1
          , OTHERWISE $ do
              count := 0
              CASE [ IF (reg err .>=. 0) $ do
                       cut := 1
                       err   := reg err + nerr
                   , OTHERWISE $ do
                       cut := 0
                       err   := reg err + perr
                   ]
          ]
     return  (reg count .==. 0)
           where sz :: Integer
                 sz = fromIntegral (size (error "witness" :: x))
                 num = numerator n
                 dom = denominator n
                 step = fromIntegral $ floor (1 / n)
                 perr = fromIntegral $ dom - step       * num
                 nerr = fromIntegral $ dom - (step + 1) * num
```

Fig. 6. RTL-based sample pulse builder

Figure 6 gives a complete implementation of such a function, using the RTL extension. As can be seen, the example reflects the RTL style of VHDL state-based programming, but keeps the type-based approach of Kansas Lava intact. To explain the example, `rate` takes a witness (approximately the analog of a generic argument in VHDL) and a fractional argument, to return a clocked `Bool` that will be punctuated at the given rate. Again, the generative motive appears; this function takes runtime arguments, to generate a circuit.

After checking for various pre-conditions, `rate` allocates three registers, and conditionally either increments a counter, or resets it and records an error from ideal. The key function is `runRTL`, which has type:

```
runRTL :: (Clock c)
       => (forall s . RTL s c a) -> a
```

`runRTL` takes a structure, called a `RTL`, that looks like a list of assignments using `do`-notation, and converts this list guarded assignments into regular Kansas Lava signal-based code. `runRTL`, in a real sense, provided the semantics and implementation of the RTL abstraction. In this case, a straightforward clocked atomic transfer semantics have been encoded, but there is no reason why a more powerful semantics, like the semantics of the high-level modeling language BlueSpec [1] could be coded. Indeed, this framework allows the exploration of new RTL-based abstraction models with extremely low cost of development, because so much of the existing infrastructure is reused. For comparison purposes, the entire RTL abstraction, including `runRTL`, is implemented in around 150 lines of commented Haskell.

X. Example: Recursion and Circuit Generation

In a recent project, we needed to generate a computational fabric to perform Low Density Parity Check (LDPC) forward error correcting [10], based on operations to a large sparse matrix. Specifically, using the notation from the standard reference on error correcting codes [11], the LDPC fabric needed to compute three things as quickly as possible:

For each (m, n) where $A(m, n) = 1$:

$$\eta^{[l]}_{m,n} = -\tfrac{3}{4}\left(\llparenthesis \min{}^{\dagger} \rrparenthesis_{j\in\mathcal{N}_{m,n}}\ (\eta^{[l-1]}_{m,j} - \lambda^{[l-1]}_{j})\right)$$

where

$$\min{}^{\dagger}(x, y) = sign(x) * sign(y) * min(|x|, |y|)$$

For each n:

$$\lambda^{[l]}_n = \lambda^{[0]}_n + \textstyle\sum_{m\in\mathcal{M}_n}\ \eta^{[l]}_{m,n}$$

For each n:

$$\hat{c}^{[l]}_n = 1, \text{if} \lambda^{[l]}_n > 0, \text{otherwise} = 0.$$

The input, $\lambda^{[0]}$ is a soft input, being a representation of likelihood of a symbol, with zero representing unknown. $\mathcal{M}_n$ represents $\{m\ :\ A_{m,n} = 1\}$, and $\mathcal{N}_{m,n}$ represents $\{n : A_{m,n} = 1\} \setminus n$.

A is a sparse matrix, approximately 7K by 3K, with a density of around 6 non-zero elements per row. After analyzing the problem domain and the (fixed) matrix, we decided to generate a two dimensional fabric of small unit blocks, that would be controlled by a central controller. These smaller

```
operateOnMatrix :: (Clock clk, sig ~ CSeq clk)
  => Options
  -> A (Int,Int)
  -> sig DecodeCntl
  -> [ sig (Write SZ FLOAT) ]
  -> [ Maybe (Int,sig (Write SZ FLOAT)) ]
 -> ( [ Maybe (Int,sig (Write SZ FLOAT)) ],
      [ Maybe (sig (Write SZ FLOAT)) ],
      [ Maybe (sig (Write SZ Bool)) ]
    )
```

Fig. 7. Type of the Recursive LDPC Fabric Generator

blocks would communicate with each other in a serial manner, cutting down internal wiring overhead.

Figure 7 gives the type of our recursive implementation. Though involved, the type spells out exactly what gets passed to, and returned from, `operateOnMatrix`. Specifically, `operateOnMatrix` is a recursive function. `operateOnMatrix` checks to see if `A` is at a threshold. If the threshold is satisfied, `operateOnMatrix` generates a small circuit we call a cell, otherwise two recursive calls are made to `operateOnMatrix`, to generate two sub-circuits, and they are combined to return a larger circuit. The technical details of this implementation can be found in [8].

XI. Example: Derivation of a Forward Error Corrector

As a final example, we connect together the LDPC example from section X to a specification of LDPC, also written in Haskell. Consider again the main equation in the LDPC algorithm.

For each (m, n) where $A(m, n) = 1$:

$$\eta^{[l]}_{m,n} = -\frac{3}{4}\left((\!|\min^{\dagger}|\!)_{j\in\mathcal{N}_{m,n}}\ (\eta^{[l-1]}_{m,j} - \lambda^{[l-1]}_{j})\right)$$

In Haskell, a transliteration of the same operation *is* a semi-formal and executable specification, and was sufficiently fast to generate the required test vectors.

```
eta' = [ ( (m,n)
         , -0.75 * (fold minDagger
                        [ eta ! (m,j) - lam ! j
                        | j <- toList (rows a ! m)
                        , j /= n
                        ]))
       | (m,n) <- toList a
       ]
```

This is a classic example of Haskell being used as a specification, with the whole decoder being specified in around 20 lines of Haskell. The motivation for starting at this specification was one of cautiousness, because implementing LDPC is notoriously tricky. So a refinement exercise was undertaken, inside our text editor.

- The original Haskell-based specification was refined so that the computations were performed in specific computation cells, and cells were connected by streams of values.
- The individual sub-components of these cells were replaced systematically with Kansas Lava primitives, in a correctness preserving manner.
- Further refinements were done, to push out from these sub-components, until the whole original model was a large (around 800 line) Kansas Lava program, as previous discussed. At this point, the Kansas Lava program can be executed to generate the circuit.

In all, 17 distinct versions of the LDPC were developed, including the semi-formal model, each of which was executable. The final version, when executed, generated VHDL which was efficiently realizable in hardware. Every refined version was a cut-and-pasted version of its predecessor, applying by hand the refinements provided by worker/wrapper [12] and other fusion transformations. The whole exercise was time consuming, and did not allow for easy backtracking (any design change in an earlier version needed changed by hand in all subsequent programs). The only bugs found were missteps in our refinements, the final version was not compromised in terms of performance, with the hardware implementation of LDPC performing within 10% of the estimated clock-rate/performance.

Critical to this whole refinement was the hand-use at every step of the worker/wrapper transformation, a mechanism for refining types based on an algebra of type coercing function. The path through the LDPC refinement chain was planned by deciding the specific types of communicating sub-components at each step. From these, type coercion functions are constructed, and simple fusion properties of the coercions were used to allow the new implementations to be derived. Most of the process was mundane, after the key type-based planning had been done.

To ground this refinement description, figure 8 gives an example of a refinement step on our key implementation function. This function takes the sparse matrix A, which represents the specific parity code, and generates a program that takes and returns data. At this point in the refinement (version 9 to version 10), we have two inputs and two outputs. To break down our type, reading from out to in,

```
[ Stream (Matrix SZ (Maybe FLOAT)) ]
```

has the meaning

`[...]`	List representing the connection to a specific number of computational "cells";
`Stream ...`	of a stream of unclocked sequential values, one per iteration of LDPC;
`Matrix SZ ...`	of a matrix, size `SZ`;
`Maybe ...`	and an optional value;
`FLOAT`	of a custom sized, quantized, real number.

Version 9

```
(...) => A (x,y)
      -> [ Stream (Matrix SZ FLOAT) ]
      -> [ Stream (Matrix SZ (Maybe FLOAT)) ]
      -> ( [ Stream (Matrix SZ (Maybe FLOAT)) ]
         , [ Stream (Matrix SZ (Maybe FLOAT)) ])
         )
```

Version 10

```
(...) => A (x,y)
      -> [ Stream (Matrix SZ FLOAT) ]
      -> [ Maybe (Stream (Matrix SZ FLOAT)) ]
      -> ( [ Maybe (Stream (Matrix SZ FLOAT)) ]
         , [ Maybe (Stream (Matrix SZ FLOAT)) ])
         )
```

Fig. 8. Refinement Source and Target Types

In this step, we want to move (or more specifically, commute) the `Maybe`, to between the list and the `Stream`. This step enables a key optimization of the implementation, specifically the ability to eliminate the circuitry of parity sub-maxtrixes that are encoded using the zero matrix. In the block-circuit versions of the LDPC, over half the cell-sized blocks are, by design, zeros. The preconditions for performing this specific type refinement using worker/wrapper hinges on making sure that a specific sub-block *always* generates the optional `FLOAT` value, or *never* generates an optional `FLOAT`. The step-specific coercions, as required by worker/wrapper, are straightforward to write, and using this pre-condition, also straightforward to validate for use. The worker/wrapper transformation can then be applied, and any sub-coercions can be pushed inside the definition of the implementation of version 9, deriving version 10 through equational reasoning and fusion laws.

A detailed description of this refinement can be found in [13]. The overarching narrative is that using the same underlying language for the specification (straight Haskell) and the implementation (Kansas Lava) has the benefit of allowing Haskell refinement techniques to use to connect, at least semi-formally, the implementation and specification.

XII. Related Work

The original ideas for Lava can be traced back to the Ruby hardware description language [14] and prior to that, μFP [15]. A good summary of the principles behind Lava can be found in [16]. JHDL [17] is a hardware description language, embedded in Java, which shares many of the same ideas found in Lava.

The overarching use of refinement is inspired by the outstanding research undertaken by the Computing Laboratory at the University of Oxford. Specifically, the methodologies promoted by Bird, et. al. [18], [19] call for a brighter future for robust software and hardware development.

Kansas Lava is a modeling language as well a synthesis tool. It models communicating processes, via synchronous signals. There are several other modeling languages that share a similar basic computational basis, for example Esterel [20].

XIII. Summary and Conclusions

Kansas Lava is two years old. It has been used an number of projects, generating Forward Error Correctors, building communication bridges between UNIX hosts and FPGA computational engines, and other smaller components, including a double-buffered FIFO/interleaver VHDL component.

The core Kansas Lava implementation is mostly complete, and will be released as an open-source product in the near future. Experimentation with idioms, advanced APIs, and refinement engines continues. The long term goal of clear specifications, connected (via refinement or compilation) to implementation descriptions in being actively explored by the Kansas Lava project. The ideas here are general; there is no reason why there could not be a Kansas Lava analog for generating C or C++, for example, as is done in the CoPilot project [21]. Building descriptions of programs in languages like Haskell, and building further infrastructure on top of this has potential for a long-term impact to the way we build software, and design hardware.

References

[1] Arvind, R. Nikhil, D. Rosenband, and N. Dave, “High-level synthesis: an essential ingredient for designing complex asics,” in *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, nov. 2004, pp. 775 – 782.

[2] A. Gill, T. Bull, A. Farmer, G. Kimmell, and E. Komp, “Types and type families for hardware simulation and synthesis: The internals and externals of Kansas Lava,” in *Proceedings of Trends in Functional Programming*, May 2010.

[3] A. Farmer, G. Kimmell, and A. Gill, “What’s the matter with Kansas Lava?” in *Proceedings of Trends in Functional Programming*, May 2010.

[4] S. Peyton Jones, Ed., *Haskell 98 Language and Libraries – The Revised Report*. Cambridge, England: Cambridge University Press, 2003.

[5] S. L. Peyton Jones and P. Wadler, “Imperative functional programming,” in *POPL ’93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1993, pp. 71–84.

[6] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, “Introducing Kansas Lava,” in *Proceedings of the Symposium on Implementation and Application of Functional Languages*, Sep 2009.

[7] A. Gill, “Type-safe observable sharing in Haskell,” in *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, Sep 2009.

[8] A. Gill, T. Bull, D. DePardo, A. Farmer, E. Komp, and E. Perrins, “Using functional programming to generate an LDPC forward error corrector,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*, May 2011.

[9] K. Czarnecki and U. W. Eisenecker, *Generative programming: methods, tools, and applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[10] K. S. Andrews, D. Divsalar, S. Dolinar, J. Hamkins, C. R. Jones, and F. Pollara, “The development of turbo and LDPC codes for deep-space applications,” *Proc. IEEE*, vol. 95, no. 11, pp. 2142–2156, Nov. 2007.

[11] T. K. Moon, *Error correction coding : mathematical methods and algorithms*. Hoboken, N.J.: Wiley-Interscience, 2005. [Online]. Available: http://www.loc.gov/catdir/toc/ecip055/2004031019.html

[12] A. Gill and G. Hutton, “The worker/wrapper transformation,” *Journal of Functional Programming*, vol. 19, no. 2, pp. 227–251, March 2009.

[13] A. Gill and A. Farmer, “Deriving an efficient FPGA implementation of a low density parity check forward error corrector,” submitted to the 16th ACM SIGPLAN International Conference on Functional Programming.

[14] G. Jones and M. Sheeran, “Circuit design in ruby,” in *Formal Methods for VLSI Design*, Staunstrup, Ed. Elsevier Science Publications, 1990.

[15] M. Sheeran, “mufp, a language for vlsi design,” in *LFP ’84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*. New York, NY, USA: ACM, 1984, pp. 104–112.

[16] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, “Lava: Hardware design in Haskell,” in *International Conference on Functional Programming*, 1998, pp. 174–184. [Online]. Available: citeseer.nj.nec.com/bjesse98lava.html

[17] P. Bellows and B. Hutchings, “JHDL - an HDL for reconfigurable systems,” *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, p. 175, 1998.

[18] R. S. Bird and O. De Moor, *Algebra of Programming*, ser. International Series in Computing Science. Prentice Hall, 1997, vol. 100.

[19] R. Bird, *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010. [Online]. Available: http://www.cambridge.org/gb/knowledge/isbn/item5600469

[20] G. Berry, “The constructive semantics of pure Esterel,” 1999, http://www-sop.inria.fr/esterel.org/files/.

[21] L. Pike, A. Goodloe, R. Morisset, and S. Niller, “Copilot: A hard real-time runtime monitor,” in *RV*, ser. Lecture Notes in Computer Science, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, Eds., vol. 6418. Springer, 2010, pp. 345–359.