Higher-Order Symb Comput
DOI 10.1007/s10990-013-9098-7

Types and associated type families for hardware simulation and synthesis

The internals and externals of Kansas Lava

Andy Gill · Tristan Bull · Andrew Farmer · Garrin Kimmell · Ed Komp

Abstract In this article we overview the design and implementation of the second generation of Kansas Lava. Driven by the needs and experiences of implementing telemetry decoders and other circuits, we have made a number of improvements to both the external API and the internal representations used. We have retained our dual shallow/deep representation of signals in general, but now have a number of externally visible abstractions for combinatorial and sequential circuits, and enabled signals. We introduce these abstractions, as well as our abstractions for reading and writing memory. Internally, we found the need to represent unknown values inside our circuits, so we made aggressive use of associated type families to lift our values to allow unknowns, in a principled and regular way. We discuss this design decision, how it unfortunately complicates the internals of Kansas Lava, and how we mitigate this complexity. Finally, when connecting Kansas Lava to the real world, the standardized idiom of using named input and output ports is provided by Kansas Lava using a new monad, called `Fabric`. We present the design of this `Fabric` monad, and illustrate its use in a small but complete example.

Keywords Domain specific languages · Hardware · Synthesis · Types

A. Gill (✉) · T. Bull · A. Farmer · G. Kimmell · E. Komp
The University of Kansas, 2001 Eaton Hall, 520 West 15th Street, Lawrence, KS 66045-7621, USA
e-mail: andygill@ittc.ku.edu

T. Bull
e-mail: tbull@ittc.ku.edu

A. Farmer
e-mail: afarmer@ittc.ku.edu

G. Kimmell
e-mail: kimmell@ittc.ku.edu

E. Komp
e-mail: komp@ittc.ku.edu

Published online: 12 October 2013

1 Introduction

Domain Specific Languages (DSLs) can be rapidly developed by hosting them in a general-purpose language, taking advantage of that language's features and toolchain. Haskell, a pure language based on lambda calculus [34], is well suited to host DSLs due to its purely functional nature and best-in-class support for abstraction. Haskell DSLs are typically combinator libraries supporting the fundamental operations of the chosen domain, leaving the full power of the Haskell language at the disposal of the DSL user [11].

Kansas Lava is a modern implementation of a Haskell-hosted hardware description language that uses Haskell functions to express hardware components, and leverages the abstractions in Haskell to build complex circuits. Lava [7], the given name for a family of Haskell-based hardware description libraries, is an idiomatic way of expressing hardware in Haskell that allows for simulation and synthesis to hardware. In this article, we explore the internal and external representation of a `Signal` in Kansas Lava, where `Signal` is the type used for the connections between sub-components in a hardware circuit. We also look at how different representations of `Signal`-like concepts work together in concert. We assume a working knowledge of Haskell in this article, especially regarding the advanced use of types, up to and including the associated type families extension to the Glasgow Haskell compiler [8].

By way of introducing Kansas Lava, consider the problem of counting the number of instances of `True` in each prefix of an infinite list. Here is an executable specification of such a function in Haskell:

```
counter :: [Bool] -> [Int]
counter inc = [ length [ () | True <- take n inc ]
              | n <- [0..]
              ]
```

Of course, this function is not a reasonable implementation as it duplicates a lot of work. In practice, we would use a function where the value for each prefix is defined in terms of the value for the previous prefix.

```
counter :: [Bool] -> [Int]
counter inc = res
   where res = [ if b then v + 1 else v
               | (b,v) <- zip inc old ]
         old = 0 : res
```

Haskell programmers are accustomed to using lazy lists as one of the replacements for traditional assignment. The `counter` function here can be considered a mutable cell, with streaming input and output, even though referential transparency has not been compromised.

Functional programmers and hardware designers take similar approaches to designing cooperating processes that communicate using lazy streams. Lava descriptions of hardware are simply Haskell programs, similar in flavor to the second `counter` function, that tie together primitive components using value recursion for back-edge creation. Our `counter` example could be written as follows in Lava.

```
counter :: Signal Bool -> Signal Int
counter inc = res
  where res = mux2 inc (old + 1,old)
        old = register 0 res
```

In this example, mux2 uses its first argument to make a choice between its second and third argument, and register is a flip-flop that remembers its input by outputting values delayed by one clock cycle. Lava programs are constructed out of functions like counter, and blocks of functionality with stored state communicate using signals of sequential values, just like logic gates and sequential circuits in hardware. These descriptions of connected components get translated into hardware gates, other entities, and signals between them.

2 Kansas Lava

Starting in 2009, we developed a new version of Lava, which we call Kansas Lava [18], in order to generate a specific set of rather complex circuits that implement high-performance high-rate forward error correction codes [16, 20, 27]. This paper discusses our observations regarding the shortcomings of our original Kansas Lava design, and what changes were made to address these shortcomings.

The 2009 version of Kansas Lava had three distinguishing features:

– Dual-use Signal. That is, the same Signal could be used for interpretation and for generation of VHDL circuits. The above circuit example could be directly executed inside GHCi, or reified into VHDL without *any* changes to our circuit specification.
– Use of lightweight *sized-types* to represent sized vectors. Our vector type, Matrix was parameterized by two types, representing the size of the vector and the type of the elements, respectively.
– Use of IO-based reification [14] for graph capture. The loop in the counter example above could be observed as a graph of connected primitives, making VHDL generation straightforward.

So what went wrong and what worked well with Kansas Lava, circ. 2009? We found the need to make the following changes for our current (2012) version:

– Add a phantom type [25] for clock domains, which allows us to unify sequential and combinatorial circuits in a type-safe way by using universal quantification (Sect. 3).
– Introduce the possibility of unknown values into our simulation embedding (Sect. 4).
– Introduce functions for commuting types that contain our Signal, giving a representation agility we found necessary (Sect. 5).
– With these building blocks, provide various simple *protocols* for hardware communication, using types to specify usage (Sect. 6).
– To connect to the real world, introduce an abstraction called Fabric, the IO monad of Kansas Lava. Using Fabric, an untyped abstraction with named signals, we connect directly and indirectly to real circuits using generated VHDL (Sect. 7).

Furthermore, this paper makes the following contributions:

– We document the shortcomings of our original straightforward implementation of the Lava ideas as well as our new solutions and design decisions.
– Some Haskell language features we employ were not available to the original Lava developers. In particular, associated type families [8] were not available. This paper gives evidence of the usefulness of type families, and documents the challenges presented by using type families in practice.
– Representation agility, that is the ability to be flexible with the representations used for communication channels, turned out to be more important than we anticipated. We document why we need this flexibility and our solution, a variant of commutable functors.

This article is an updated and expanded version of [19]. In addition to improving the presentation and adding a front-to-back example, the splitting of signals into sequential and combinatorial circuits has a completely new form. Previously, this was done with distinct data types, lift functions, and a unifying type class, while in this paper we distinguish between sequential and combinatorial using phantom types. We have also completely rewritten the discussion of protocol support, and have added the presentation of the internals and externals of the new `Fabric` abstraction. The `Fabric` API, but not the implementation, was previous summarized in an invited position paper [15].

3 Sequential and combinatorial circuits

Haskell is a great host for Domain Specific Languages. Flexible overloading, a powerful type system, and lazy semantics facilitate this. An embedded DSL in Haskell is simply a library with an API that makes it feel like a small language customized to a specific problem domain. There are two flavors of embedded DSLs:

- DSLs that use a **shallow embedding** perform computation directly on values in the host language. Most definitions of monads in Haskell are actually shallow DSLs, for example. The result of a computation in a shallow DSL is a value.
- DSLs that use a **deep embedding** build an abstract syntax tree. The result of a computation inside a deep DSL is a structure, not a value. This structure can in turn be used to compute a value, to generate object code, or for static analysis. The technique of deep embedding a DSL is what allows the generation of VHDL from Lava descriptions.

In our first iteration of the embedded DSL Kansas Lava [18], we decided to provide a principal type, `Signal`, and all Kansas Lava functions used this type to represent values over wires that change over time. Kansas Lava was unusual in that `Signal` contains both a shallow and deep embedding. This is so that the same `Signal` could be both executed and reified into VHDL, as directed by the Kansas Lava user. The shallow embedding was encoded as an infinite stream of direct values and the deep embedding was an abstract syntax tree with a phantom typed parameter. Slightly simplified for clarity, we used:

```
data Signal a = Signal (Stream a) (D a) -- To be refined

data Stream a = a :~ Stream a           -- No null constructor

type D a = AST                    -- Wrapper with phantom type

data AST = Var String                   -- Simplified AST
         | Entity String [AST]
         | Lit Integer
```

We can see that `Signal` is an abstract tuple of the shallow `Stream a`, and the deep abstract syntax tree, `D a`. Using `Signal` in this form, we could write operations that support both the shallow and deep components of `Signal`. As an example, consider `and2`, which acts over `Signal Bool`.

```
and2 :: Signal Bool -> Signal Bool -> Signal Bool
and2 (Signal a ae) (Signal b be) = Signal (zipWith (&&) a b)
                                          (Entity "and2" [ae,be])
```

Here, we can see that the definition of `and2` splits both arguments into the deep and shallow components, then recombines them. The shallow result is implemented using an appropriately typed `zipWith` over the boolean stream, and the deep result a single node with the name `"and2"`.

The first issue we faced was one of semantic conciseness. There are actually two useful versions of `and2`. One, as above, operates over *sequential* inputs, representing streams of values over time. The second version operates on *combinatorial* inputs, or base values. Combinatorial circuits have no notion of history and state, analogous to pure functions in Haskell. As such, they are often easier to compose and reason about than their sequential versions. (We acknowledge but do not elaborate on a third alternative, a sequential variant of `and2` that delays its output by a clock cycle, as this can be constructed from existing primitives.)

Rather than have two versions of `and2`, we initially defined two signal types, with lifting functions to turn combinatorial circuits into sequential ones.

```
data Comb a = Comb a (D a)         -- rejected implementation
data Seq a = Seq (Stream a) (D a) -- rejected implementation

class Signal sig where
  liftSeq0 :: Comb a -> sig a
  liftSeq1 :: (Comb a -> Comb b) -> sig a -> sig b
  liftSeq2 :: (Comb a -> Comb b -> Comb c) -> sig a -> sig b
                                                    -> sig c
```

Using instances of the `Signal` class for `Comb` and `Seq`, circuits could be combinatorial, sequential or overloaded to be either. For instance, `and2` could be defined thus.

```
and2 :: (Signal sig) => sig Bool -> sig Bool -> sig Bool
and2 = liftSeq2 (\(Comb a ae) (Comb b be) ->
                        Comb (a && b) (Entity "and2" [ae,be]))
```

The challenge, however, was one of complexity of design. This overloading was pervasive, and in practice most circuits were lifted into sequential circuits. Further, we needed multiple lift functions, as the trick of using Applicative Functors [26] to build varargs-style lifts did not work for technical reasons that will be elaborated in Sect. 6.

Having two different types for our two families of signals, though conceptually sound, was cumbersome in practice, leaving us looking for a different approach. A key insight is that combinatorial circuits have no clock. Accordingly, we have since adopted a different approach, using a single type `Signal` with a phantom type [25] representing the clocking strategy used.

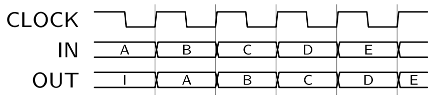
```
data Signal clk a = Signal (Stream a) AST
```

By using a phantom type argument, all circuits act over the unified `Signal`, but the parameterization characterizes the permissible interpretations. We used the following schema for the interpretation of `Signals` with clocks.

– *`Signal` can be given a concrete type witness for a specific clock.* By convention, the standard board clock is taken to be an element of type `CLK`, but there can be many clocks. In this way, `Signal CLK a` is equivalent to `Seq a` in the discussion above. As an example, `counter` can be defined in terms of the default board clock:

```
  counter :: Signal CLK Bool -> Signal CLK Int
```

Fig. 1 Waveform of the `register` primitive with initial value of `I`



– *`Signal` can be given a type argument which admits membership of the* `Clock` *class.* The circuit can then assume that there is a clock available, though which specific clock is unspecified. As such, this circuit can make use of functions requiring a clock, such as registers and latches. Defining `counter` to work with any clock:

```
counter :: (Clock c) => Signal c Bool -> Signal c Int
```

Note that we are specifying the same clock for the input and output signals, but we do not know what it is, because the circuit is overloaded over all clocks.

– *`Signal` can be given a type argument that is universally quantified.* Universal quantification means the circuit can assume nothing about the signal's clocking strategy. A circuit that takes and returns a `Signal` with a universally quantified parameter could equally well be combinatorial or sequential. The fact that the circuit has this universally quantified type is a witness to the fact that the circuit used no primitive that requires clocking, such as a register or latch. The `counter` example above could never have such a universally quantified type, because the counting depends on having a ticking clock. However, as discussed above, `and2` can be either combinatorial or sequential.

```
and2 :: forall c . Signal c Bool -> Signal c Bool
                                 -> Signal c Bool
```

This schema neatly compartmentalizes the various circuit types into known clocks, any clock, and circuits that explicitly do not use a clock. One advantage of this unified use of `Signal` is that we no longer need any explicit lifting functions to use a combinatorial sub-circuit—just using the sub-circuit specializes the universally quantified clock to a specific clock.

To see the phantom clock type in action, consider the basic clocked primitive `register`, which starts with a given initial value, and delays the output by one cycle. Figure 1 gives a timing diagram for this primitive.

```
register :: (..., Clock c) => a -> Signal c a -> Signal c a
register a (Signal ss sd) = Signal (a :~ ss) (...)
```

This principal primitive forces all sequential circuits to have a `Clock` overloading.

Further, we observe that it is still possible to require combinatorial sub-circuits, by the use of rank-2 polymorphism. An example is the Kansas Lava function `iterateS`:

```
iterateS ::  (..., Clock c)
         => (forall k . Signal k a -> Signal k a)
         -> a -> Signal c a
iterateS f start = out
   where out = register start (f out)
```

The expression `iterateS (+1) 0` would return a `Signal` of ascending numbers, starting at 0. By making use of rank-2 polymorphism, the type-checker enforces that we do not pass a sequential circuit (with an overly specified clock) to `iterateS` by mistake.

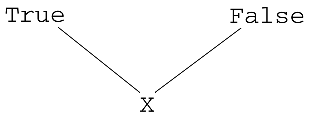
Finally, we observe that in the examples above, it is common that the input and output `Signals` are specialized to a common clock. We use the ~ syntax, originally intended for helping type code with associated type synonyms, to provide a local type synonym for `Signal`. For example, `register` can be re-written as

```
register :: (Clock c, sig ~ Signal c, ...) => a -> sig a
         -> sig a
```

We use this idiom pervasively throughout Kansas Lava, and will use the idiom where possible for the remainder of the paper.

4 Venturing into the unknown

Often in hardware, the value of a wire is unknown. Not defaulting to some value like zero or high, but genuinely unknown. The IEEE definition of a bit in VHDL [22] captures this with the notation `X`. Such unknowns are introduced externally, or from reset time, and represent a value outside the possible values of valid primitive types. For example, when modeling hardware and transmitting a boolean, we essentially want a lifted domain.



In this case we can use the type `Maybe Bool`, but the situation is more complex for structured types. Consider a `Signal` that represents `(Bool,Bool)`. Through experience, we want the two elements of the tuple to have *independently* lifted status. If we give the `Signal` a single unknown that represents both elements, then circuits in practice will over-approximate to unknown, hampering our shallow embedding simulation. This makes sense if we consider our hardware targets, in which a two-tuple of values will be represented by two independent wires.

We solve this problem using associated type families [8]. We introduce a new type class, `Rep`—**Rep**resentable in hardware—which captures the possibility of unknowns and other issues concerning the representation of the values in wires.

```
class Rep w where
    data X w                   -- X are lifted values on a wire
    fromX :: X w -> Maybe w    -- extract value from lifted type
    toX :: Maybe w -> X w      -- inject value into lifted type
    ...
```

This class provides an associated data family, `X`, which provides a lifted version of its type argument, and a way to extract a value from, as well as inject a value into, the lifted type. We use a data family, instead of a synonym family, because in this case we desire injectivity (which is provided by the data instance constructor). As we will soon see, calls to `fromX` and `toX` are quite common, and if `X` is not injective, every call to `fromX` would need a type ascription in order to determine `w`.

For example, our instance for `Bool` makes direct use of `Maybe`.

```
instance Rep Bool where
    data X Bool  = XBool (Maybe Bool)
```

```
      toX (Just b) = XBool (return b)
      toX Nothing  = XBool Nothing
      fromX (XBool (Just v)) = return v
      fromX (XBool Nothing)  = Nothing
```

Our instance for tuples uses tuples of `X`.

```
  instance (Rep a, Rep b) => Rep (a,b) where
      data X (a,b)        = XTuple (X a, X b)
      toX (Just (a,b))    = XTuple ( toX (return a)
                                   , toX (return b))
      toX Nothing         = XTuple ( toX (Nothing :: Maybe a)
                                   , toX (Nothing :: Maybe b))
      fromX (XTuple (a,b)) = do x <- fromX a
                                y <- fromX b
                                return (x,y)
```

Diagrammatically, we represent `(Bool,Bool)` that can admit failure using:

$$\texttt{X (Bool,Bool)} \cong \texttt{(X Bool,X Bool)} \cong \left(\begin{tikzcd}[column sep=tiny] \texttt{True} \arrow[dr, dash] & & \texttt{False} \arrow[dl, dash] \\ & \texttt{X} & \end{tikzcd} , \begin{tikzcd}[column sep=tiny] \texttt{True} \arrow[dr, dash] & & \texttt{False} \arrow[dl, dash] \\ & \texttt{X} & \end{tikzcd} \right)$$

So there are 9 possible values for the pairing of the two boolean signals, three for left component multiplied by three for the right component.

We can now give our complete type for `Signal`:

```
  data Signal clk a = Signal (Stream (X a)) (D a)
       -- actual implementation
```

Shallow `Signals` are `Streams` of lifted values. Deep streams remain the same as before; VHDL and other hardware description languages have the concept of under-defined built in. Having undefined values in our alphabet allows us to have one new clocking combinator.

```
  delay :: (Rep a, Clock c, sig ~ Signal c) => sig a -> sig a
```

`delay` is a variant of `register` that issues an unknown for its initial value, which is a useful, realistic representation of an uninitialized flip-flop.

One complication is that all function primitives now need to be written over the `X` type. We can use the built-in support for the `Maybe` monad where possible. In practice, some functions are straightforward, because the `X` type maps to (say) the `Maybe` data type. `Maybe` is an Monad, so `liftM2` can be used. To give a concrete example, consider the definition of `(+)` in Kansas Lava.

```
  instance (Num a, Rep a) => Num (Signal clk a) where
      -- actual implementation of (+)
      s1 + s2 = primS2 (+) "+" s1 s2
      ...
```

We define `(+)` in terms of a utility, `primS2`.

```
  -- A lifting function for pure, 2 argument functions.
  primS2 :: (Rep a, Rep b, Rep c)
         => (a -> b -> c) -> String
         -> Signal clk a -> Signal clk b -> Signal clk c
```

```
primS2 f nm = primXS2 (\ a b -> toX $ liftM2 f (fromX a)
                                                (fromX b)) nm


-- Create an arity-2 Signal function from an 'X' function.
primXS2 :: (Rep a, Rep b, Rep c)
        => (X a -> X b -> X c) -> String
        -> Signal clk a -> Signal clk b -> Signal clk c
primXS2 f nm (Signal a1 ae1) (Signal a2 ae2)
        = Signal (S.zipWith f a1 a2)
                 (... construct AST for this prim ...)
```

On the other hand, some primitives handle unknown values directly. For example, `and2` always returns `False` if *either* argument is `False` (even if the other argument is unknown). Rather than lift Haskell's `&&` function with `primS2`, we provide this short-circuiting behavior directly with `primXS2`. Here is the actual implementation.

```
and2 :: (sig ~ Signal clk) => sig Bool -> sig Bool -> sig Bool
and2 = primXS2 (\ a b -> case (fromX a, fromX b) of
         (Just True , Just True)  -> toX $ Just True
         (Just False, _)          -> toX $ Just False
         (_         , Just False) -> toX $ Just False
         _                        -> toX $ Nothing) "and2"
```

5 Signals and commutable functors

If we allow `Signals` of tuples and other compound types, which of these two types for `halfAdder` is more natural?

```
halfAdder :: (sig ~ Signal c) => (sig Bool, sig Bool)
          -> (sig Bool, sig Bool)
halfAdder :: (sig ~ Signal c) => sig (Bool,Bool)
          -> sig (Bool,Bool)
```

The first allows us to more naturally construct the argument tuple and pattern match on the resulting tuple. The second can be more naturally connected to other circuit components, such as a `register`. We found ourselves going around in circles between the two approaches. We wanted to support both forms, with a means to *manually* coerce one form into the other form. To enable this, we again employ type families.

The class `Pack` signifies that a `Signal` can be used inside or outside a specific structure. The type translation from the packed representation (structure inside, `Signal` on the outside) to the unpacked representation (structure outside, `Signal` on the inside) is given by a synonym family inside the `Pack` class:

```
class Pack clk a where
    type Unpacked clk a
    -- Pull the sig type *out* of the compound data type.
    pack :: Unpacked clk a -> Signal clk a
    -- Push the sig type *into* the compound data type.
    unpack :: Signal clk a -> Unpacked clk a
```

This class says we can pack an Unpacked representation, and unpack it back again. Note that, in Haskell, type families are not injective, which is actually desirable in this case. Two packed Signals may have the same unpacked representation. Packed Signals are not necessarily isomorphic to their unpacked representation, as long as the unpacked type is 'bigger' (that is, can represent all the values in the packed Signal).

Reconsidering the halfAdder example above, which was over the *structure* two-tuple, we can give the Pack instance for pairs.

```
instance (...) => Pack clk (a,b) where
    type Unpacked clk (a,b) = (Signal clk a, Signal clk b)
    -- pack   :: (sig ~ Signal clk) => (sig a, sig b)
              -> sig (a,b)
    -- unpack :: (sig ~ Signal clk) => sig (a,b)
              -> (sig a, sig b)
```

(The types are given rather than the tedious details of the implementation here.) As can be seen from the types, pack packs the two-tuple structure inside a signal, and unpack lifts this structure out again.

Consider the alternative implementations of the two halfAdder circuits.

```
-- unpacked version
halfAdder :: (sig ~ Signal c) => (sig Bool, sig Bool)
          -> (sig Bool, sig Bool)
halfAdder (a,b) = (a `xor2` b, a `and2` b)

-- packed version
halfAdder :: (sig ~ Signal c) => sig (Bool,Bool)
          -> sig (Bool,Bool)
halfAdder inp = pack (a `xor2` b, a `and2` b)
  where (a,b) = unpack inp
```

Both styles, Signals of tuples and tuples of Signals, can be used by the user as needed without a huge impact on clarity because of the generic nature of the pack/unpack pair.

Sometimes, the Unpack class allows access to the underlying representation. For example, consider Signal (Maybe Word8). This is a Signal of optional Word8s. A hardware representation might be 8 bits of Word8 data, and 1 bit of validity. Our Unpack instance for Maybe reflects this representation.

```
instance (...) => Pack clk (Maybe a) where
    type Unpacked clk (Maybe a) = (Signal clk Bool,
                                   Signal clk a)
    -- pack   :: (sig ~ Signal c) => (sig Bool, sig a)
              => sig (Maybe a)
    -- unpack :: (sig ~ Signal c) => sig (Maybe a)
              -> (sig Bool, sig a)
```

In general, the unpacked structure *must* be able to denote the complete space of the signal to be packed. Transposing the order of the type constructors does not give an isomorphic value in general, as the Maybe instance demonstrates. Table 1 lists the types pack and unpack supports. Nested packing and unpacking is also supported, by chaining pack and unpack together.

This Pack class has turned out to be really useful in practice. This ability cuts to the heart of what we want to do with Kansas Lava: use types in an agile way to represent the intent of

Table 1 Packed and unpacked pairs in Kansas Lava

Packed	Unpacked
`sig (a,b)`	`(sig a, sig b)`
`sig (a,b,c)`	`(sig a, sig b, sig c)`
`sig (Maybe a)`	`(sig Bool, sig a)`
`sig (Matrix sz a)`	`Matrix sz (sig a)`
`sig (StdLogicVector sz)`	`Matrix sz (sig Bool)`

the computation. These transposition-like operations might look familiar to some readers. The `pack` and `unpack` operations over pairs of *functors* are sometimes called `dist` [26]. In our case `pack` and `unpack` are tied to our `Signal` overloading; we are commuting (or moving) the `Signal`.

6 Protocols for signals

In addition to using types to more accurately model the hardware domain, we have created a number of type idioms that make constructing complex circuits easier. These idioms form three primary unidirectional protocols: validity (`Enabled`); memory (`->`); and updates to memory (`Write`).

6.1 The `Enabled` protocol

Validity of a value on a wire is a general concept. Often, this validity is communicated using an *enable* boolean signal that accompanies the value signal. Figure 2 shows this protocol in action. When the enable signal is `True`, the value signal is present, and should be consumed. When the enable signal is `False`, the value signal is arbitrary, and should be ignored. In Kansas Lava, we define this as a type:

```
type Enabled a = Maybe a
```

Using `Enabled`, we signify a value that may not always be valid. Kansas Lava provides combinators that allow combinatorial circuits to be lifted into `Enabled` circuits. Note that the concept of `Enabled` is distinct from that of unknowns denoted by the type function `X` in Sect. 4. `Enabled` is a *user-observable* phenomenon, and decisions can be based on the validity bit; whereas unknown values are a lower-level concept allowing the shallow-embedding to more accurately model the semantics of hardware signals.

6.2 The Memory `Read` Protocol

Memory is a `Signal` of functions from address to data.

```
-- example that returns a memory signal.
example :: (..., Rep a, Rep d, sig ~ Signal clk) => ...
        -> sig (a -> d)
```

In FPGA block RAMs and block ROMs, there are two ways of reading memory, synchronously (data is returned on the cycle following the address request) or asynchronously (data is returned on the same cycle). We support both in Kansas Lava.

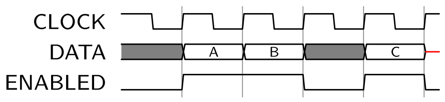


Fig. 2 Waveform showing the `Enabled` Protocol communicating `A B C` using 5 clock cycles

```
-- | Read a series of addresses. Respects the latency
--   of Xilinx BRAMs.
syncRead :: (Clock clk, sig ~ Signal clk, Rep a, Rep d)
    => sig (a -> d) -> sig a -> sig d

-- | Read a memory asynchronously. There is no clock here.
asyncRead :: (sig ~ Signal clk, Rep a, Rep d)
    => sig (a -> d) -> sig a -> sig d
```

We can see the clocking domains reflected here. The memory to be read, the address, and the result must be in the same clocking domain. We can also see that the `asyncRead` is universally quantified; an actual clock is not required. This may seem strange, but when refining Kansas Lava programs, this can be useful. For instance, `asyncRead` could be used to generate a circuit that is passed to `iterateS`. The final observation regarding `asyncRead` is that the type of `asyncRead` is the same as the type of the `<*>` combinator for Applicative Functors [26], modulo the class constraints.

6.3 The memory `Write` protocol

So how do we create a `Signal` of functions `(a -> d)`? For ROMs, there is a built in combinator that creates a lookup table from a function.

```
rom :: (Rep a, Rep d, sig ~ Signal c) -> (a -> X d)
    -> sig (a -> d)
```

RAMs are created by a sequence of write requests, which are optionally enabled address-datum pairs.

```
type Write a d = Enabled (a,d)
```

The `Write` idiom gives an *optional* write command, saying write datum (`d`) to address (`a`), if enabled.

To construct the shallow embedding of our memory `Signal`, a `Signal` of functions, we observe that, at any point in time, the contents of the memory at address `a` is equal to the most recent write request to `a`. Given access to the history of write requests (the prefix of the stream contained in the shallow embedding of the `Write Signal`), it is straightforward to generate the memory `Signal`. Specifically, in Kansas Lava, RAMs are created with `writeMemory`, a function that takes a `Signal` of `Write` requests and returns a memory `Signal`.

```
writeMemory :: (Clock clk, sig ~ Signal clk, Rep a, Rep d)
            => sig (Write a d) -> sig (a -> d)
```

6.4 Observations on protocol usage in practice

In Kansas Lava a `Signal`, like all Haskell values, can be consumed multiple times without incident. If a memory `Signal` is read by two consumers, semantically the two reads happen at the same time. On Xilinx FPGAs, however, this requires that the memory is *cloned* for each reader. Sharing memory is useful when deriving circuits, but must be carefully controlled when generating VHDL, as the hardware contains a finite number of memory elements. In general, if Xilinx tools warn that too many BRAMs are being used, memory sharing is a prime suspect.

From a hardware perspective, the clock for input stream of `Write` requests could be completely independent of the clock of the memory consumer. There is no reason, other than the complexity of implementation, that a memory *must* read and write based on the same clock. While we have experimented with a cross clock-domain variant of `writeMemory`, it remains an experimental feature and the subject of future research. For example, under this cross clock-domain paradigm, we needed to associate a clock with a specific simulation frequency, so a reasonable approximation of the cross domain sampling can be done.

Together these three types of signal protocols form a small and powerful algebraic framework over `Signal`, able to express many forms of sequential communications used by hardware designers in practice. Kansas Lava also supports bidirectional protocols, which allow for flow control; see [17] for more details.

7 The `Fabric` monad

An important reason to use Kansas Lava is to generate VHDL. Kansas Lava circuits are Haskell functions over `Signals`. Reifying a Haskell function constructed as part of a deep embedding is well understood [11, 14]. Turning this reified circuit into a usable VHDL description with a reasonable interface is more challenging. In Haskell, and therefore Kansas Lava, arguments and return values are position dependent. VHDL (and Verilog) components have named, position independent input and output ports. The challenge is to generate a VHDL circuit with named ports from a Haskell function over `Signals`.

Originally, Kansas Lava named its ports in order, with inputs (`i0`, `i1`, ...), and outputs (`o0`, `o1`, ...). The user was expected to add a VHDL wrapper that gave meaningful names to the ports. However, this was brittle and did not scale to circuits with a large number of inputs and outputs. Though the generated circuit was correctly rendered, connecting several sub-systems involved tedious and error-prone manual counting of arguments and tuple components.

After attempting to solve this problem with various kludges, we created a monad called `Fabric` to act as the layer between our functional circuits and the real world of FPGA boards. One can think of `Fabric` as the `IO` monad of Kansas Lava, or alternatively as a little DSL that handles naming and component assembly.

In order to express a circuit with names ports, we want a function mapping a list of named inputs to a list of named outputs.

```
type Fabric = [(String,Pad)] -> [(String,Pad)]  -- 1st attempt
```

Here, `Pad` is a universal `Signal` type, Kansas Lava's analogue of Haskell's `Dynamic` type for type-safe casts. We use this to allow `Signals` of varying types to be stored in a list of homogeneous-typed elements.

Promoting this function into a monad via `newtype` allows us to take advantage of monadic combinators and idioms, such as do notation.

```
newtype Fabric a
      = Fabric ([(String,Pad)] -> (a,[(String,Pad)]))
      -- 2nd attempt
```

This structure has one remaining issue: Pad names (and, internally, the widths) of both inputs and outputs must be known before the internal Fabric function is called. We can "tie the knot", allowing creation of pads as needed, by maintaining the list of input pads as state and accumulating a list of output pads with a writer. From this structure, we can reify circuits with names, as well as discover both the input and output names.

Taken together, our Fabric monad has the following structure:

```
newtype Fabric a
         --     named inputs          named inputs
      = Fabric ([(String,Pad)] -> (a,[(String,Pad)],
                                    [(String,Pad)]))
                                 -- named outputs

instance Monad Fabric where ...
```

The interface for specifying inputs and outputs in VHDL is concise and monadic.

```
inStdLogic :: (...,Rep a, W a ~ X1)
           => String -> Fabric (Signal CLK a)
inStdLogicVector :: (..., Rep a)
                 => String -> Fabric (Signal CLK a)

outStdLogic :: (..., Rep a, W a ~ X1)
            => String -> Signal CLK a -> Fabric ()
outStdLogicVector :: (..., Rep a)
                  => String -> Signal CLK a -> Fabric ()
```

The W a ~ X1 constraint uses a type function W to ensure that values of the type a can be represented as a single wire.

Like IO functions in Haskell, Fabric functions in Kansas Lava are used to read from external stimuli and write to the outside world. Being able to name ports also permits us to generate complete VHDL for a specific FPGA board by allowing us to generate a UCF description, which connects named ports in VHDL to actual hardware pins.

To turn a Fabric into a circuit, we reify the Fabric into a Kansas Lava Entity Graph (KLEG), which is then transliterated into VHDL. As an example, consider a program that reads the state of two switches on the board, and lights up three LEDs based on their position and the output of a nand gate. Figure 3 gives the complete Kansas Lava program.

We can test our example using the command line:

```
ghci> example low high
high
ghci> example high high
low
```

We use test_leds to connect example to the switches and lights, via the Fabric monad. The two Fabric functions switches and leds, imported from a board-specific Haskell module, allow us to read inputs from the board (for example, switch positions), and write outputs (for example, LEDs). These functions have the following types:

```
switches :: Fabric (Matrix X4 (Signal CLK Bool))
leds :: Matrix X8 (Signal CLK Bool) -> Fabric ()
```

Finally, we generate VHDL by capturing and reifying this Fabric into a KLEG graph, and then rendering the KLEG graph as a VHDL file. Appendix A lists the output of invoking

```
{-# LANGUAGE TypeFamilies #-}
import Language.KansasLava
import Hardware.KansasLava.Boards.Spartan3e
import Data.Sized.Matrix

-- The Kansas Lava example: a nand gate
example :: (sig ~ Signal c) => sig Bool -> sig Bool
        -> sig Bool
example in1 in2 = in1 `nand2` in2

-- The Kansas Lava Fabric: switches and leds around
-- the nand gate
test_leds :: Fabric ()
test_leds = do sw <- switches
               let in1 = sw ! 0
                   in2 = sw ! 1
                   out1 = example in1 in2
               leds (matrix $ [ in1, in2, out1 ]
                     ++ replicate 5 low)

-- Reify the Fabric, and generate VHDL for our example
main = do kleg <- reifyFabric test_leds
          writeVhdlCircuit "main" "main.vhd" kleg
          writeUCF "main.ucf" kleg
```

Fig. 3 A complete example of using Kansas Lava to generate a circuit for an FPGA

this example, with the invocation of `nand` on Line 33. The VHDL is verbose; most of the coercions here are introduced by the generic nature of the `Fabric`. (We have put no effort into solving this verbosity because almost any synthesis technology used today will automatically remove the needless coercions, leaving the core circuit.) We also automatically generate the UCF file, based on what ports are utilized inside the `Fabric`.

Without access to a hardware board, it is still possible to test specific circuits, but how do we test the connections to board components such as LEDs or toggle switches? Towards this, we have developed a framework for proving board simulators. These simulators take a compliant `Fabric`, as generated above, and execute it using the shallow embedding. Appendix B shows the simulator in action. The simulator can be interacted with (switches toggled; buttons pressed; LEDs lit; LCD utilized), and inside the simulator the external ports (RS232, VGA, RJ-45) are reflected as named UNIX pipes. With careful handling, it is actually possible to telnet into the virtual board, and interactively test how specific circuits might respond.

8 Related work

The ideas behind Lava and embedded hardware description languages are well explored, and there have been many implementations of embedded hardware description languages. There have been three families of such languages that use Haskell: Lava, Hydra and ForSyDe. We now summarize the history of these three family trees.

Lava is the given name to a class of Haskell programs that implement a function-based version of the hardware description language Ruby [21, 24]. Ruby, not to be confused with the modern programming language with the same name, was based on relations, not functions, and was inspired by the seminal work in μFP [36]. The key conundrum in Lava was how to represent loops in circuits. Chalmers Lava [7] chose to use observable sharing [9, 10], while Xilinx Lava [37] choose to use a monad to represent sharing [12, 13]. The Chalmers line of investigation lives on, in the form of Wired [1], and more recently Feldspar [2] which though not a hardware description language uses many of the same concepts and know-how as developed with Lava. Finally, York Lava is a independent effort to use the same ideas to generate VHDL for testing circuit designs and micro-architecture implementation ideas [28].

Hydra is a parallel line of investigation lead by O'Donnell at the University of Glasgow. Hydra pre-dates Haskell itself, with circuits originally being described in Daisy [31], an early lazy functional language. O'Donnell re-implemented Hydra into Haskell [29], and successfully used the developed system as a teaching tool [30]. O'Donnell continued active development on Hydra [33], including the use of Template Haskell for generating unique labels that facilitate reification [32]. Kansas Lava shares many of the concepts and solutions using in Hydra, including the specific ability to distinguish between combinatorial and sequential circuits using types, and using lifted signal values. Kansas Lava implements these concepts using modern Haskell extensions, like type functions and user-level rank-2 polymorphism. Also, while Kansas Lava uses a single `Signal` type inhabited by a phantom type argument that stores both the deep and shallow interpretation simultaneously, Hydra uses the Haskell class system to allow explicit overloading of either interpretation automatically.

ForSyDe [35], is a third system which addresses many of the same concerns as Kansas Lava. Like Kansas Lava, ForSyDe provides both shallow and deep embeddings, though in ForSyDe this is done via two distinct types using the Haskell `import` mechanism. Additionally, ForSyDe provides a rich design methodology on top of the basic language, and supports many "models of computation" [23]. The principal differentiator of Kansas Lava is its use of type families, which allow a *single* executable model to be utilized effectively. Kansas Lava has also pushed further with the family of connected signal types, and has taken a type-based approach to supporting multiple clock domains. A more complete formal comparison of the two systems remains to be done. Like Hydra, ForSyDe's makes use of Template Haskell, but to help with the generation of usable sized types. For example, vectors are statically checked for size using Template Haskell.

Outside of Haskell, there are many other generators for hardware. For example, JHDL [3] is a hardware description language, embedded in Java, which shares many of the same ideas found in Lava. It may be possible to use Java generics to apply the ideas from Kansas Lava to a recent version of Java.

Kansas Lava is a modeling language. It models communicating processes, currently via synchronous signals. There are several other modeling languages that share this computational basis, for example Esterel [4]. There are many other models for communicating processes, and each one has many language-based implementations. The overview paper by Jantsch and Sander [23] gives a good summary of this vast area of research.

Kansas Lava is designed to encourage refinement into efficient circuits, inspired by the outstanding research undertaken by the Computing Laboratory at the University of Oxford. Specifically, the methodologies promoted by [6] and [5], point towards a brighter future for robust software development, and with systems like Kansas Lava, these benefits can include hardware development as well.

9 Conclusions

In this paper we have seen a number of improvements to Kansas Lava guided by the principle of using types to express the nature and limitation of the computation being generated. Kansas Lava required many more changes than anticipated in order to turn it into a useful VHDL generation tool. For now, however, Kansas Lava as a language is somewhat stable.

Each change was initiated because of a specific shortcoming. We used a phantom type to represent clock domains, allowing us to unify our separate signal types and eliminate our explicit lifting combinators. This change also promises a way forward in dealing with circuits with multiple clock domains. We provided generic mechanisms for commuting signals with other compound types, allowing us to write functions that take advantage of Haskell idioms, such as pattern matching, but can still be usefully connected to other circuit components. We explicitly model the semantics of the unknown values in hardware, despite the pervasive consequence of this choice, because our simulations were not matching our experience with generated VHDL. We enumerate a number of basic protocols for intra-circuit communication. These protocols are elegantly expressed by their types. Finally, we have monad that provides an interface for connecting the untyped external inputs and outputs of the FPGA to the typed internal function representing the Kansas Lava circuit.

We hope to build up a stronger transformational design methodology around Kansas Lava. Currently we have a number of large circuits that have been translated from high-level models systematically into Lava circuits, where large is defined as generating millions of non-regular discrete logic units. Our largest circuit to date is about 800 lines of Kansas Lava. The commuting of signals has turned out to be extremely useful when deriving our circuits from higher-level specifications, as we discussed in [16]. Writing correct, efficient circuits is hard, and we hope to use some of these circuits as candidates for a more formal use of transformation methodologies in the future.

Acknowledgements We would like to thank the TFP 2010 referees for their useful feedback, both on this article, and on the earlier TFP version of this article. We would also like to thank Neil Sculthorpe for his detailed comments.

Appendix A: VHDL generated for interactive `nand` gate circuit

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.lava.all;
use work.all;
entity FirstExample is
  port(clk : in std_logic;
       SW : in std_logic_vector(3 downto 0);
       LED : out std_logic_vector(7 downto 0));
end entity FirstExample;
architecture str of FirstExample is
  signal sig_2_o0 : std_logic_vector(7 downto 0);
  type sig_3_o0_type is array (7 downto 0) of std_logic_vector(0 downto 0);
  signal sig_3_o0 : sig_3_o0_type;
  signal sig_7_o0 : std_logic;
  signal sig_6_o0 : std_logic;
  signal sig_4_o0 : std_logic;
  type sig_5_o0_type is array (3 downto 0) of std_logic_vector(0 downto 0);
  signal sig_5_o0 : sig_5_o0_type;
begin
  sig_2_o0 <= lava_to_std_logic(sig_3_o0(7)) & lava_to_std_logic(sig_3_o0(6))
```

```
             & lava_to_std_logic(sig_3_o0(5)) & lava_to_std_logic(sig_3_o0(4))
             & lava_to_std_logic(sig_3_o0(3)) & lava_to_std_logic(sig_3_o0(2))
             & lava_to_std_logic(sig_3_o0(1)) & lava_to_std_logic(sig_3_o0(0));
  sig_3_o0(0) <= "" & sig_4_o0;
  sig_3_o0(1) <= "" & sig_6_o0;
  sig_3_o0(2) <= "" & sig_7_o0;
  sig_3_o0(3) <= "" & '0';
  sig_3_o0(4) <= "" & '0';
  sig_3_o0(5) <= "" & '0';
  sig_3_o0(6) <= "" & '0';
  sig_3_o0(7) <= "" & '0';
  sig_7_o0 <= (sig_4_o0 nand sig_6_o0);
  sig_6_o0 <= lava_to_std_logic(sig_5_o0(1));
  sig_4_o0 <= lava_to_std_logic(sig_5_o0(0));
  sig_5_o0(0) <= SW(0 downto 0);
  sig_5_o0(1) <= SW(1 downto 1);
  sig_5_o0(2) <= SW(2 downto 2);
  sig_5_o0(3) <= SW(3 downto 3);
  LED <= sig_2_o0;
end architecture str;
```

Appendix B: Spartan3E starter board simulator running `nand` gate circuit

```
  _||_____|VGA|_____|X|__|232 DCE|__|232 DTE|__
 |o||                                          |_
 |                                             | |
 |                     +-----+                 | |
----+         DIGILENT |FPGA|    clk: 460      | |
RJ45|   ##             |    |    SPARTAN-3E    | |
----+   ##             +-----+    \      /     | |
_|_                                \    / ()   | |
USB|     +--+                       \  /       |_|
---'     |##|         +-----+      FPGA        |_
 |+--+   +--+         |####|         .....@@.  | |
 ||##|           +----------------+  76543210  |_|
 |+--+  (e)      |                |            |_
 |  (a) (|) (g)  |                |   : : k :  | |
 |      (x)      +----------------+   h j : l  |_|
 +---------------------------------------------+
```

References

1. Axelsson, E.: Functional programming enabling flexible hardware design at low levels of abstraction. Ph.D. thesis, Department of Computer Science and Engineering Chalmers University of Technology and University of Gothenburg (2008)
2. Axelsson, E., Claessen, K., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajdax, A.: Feldspar: a domain specific language for digital signal processing algorithms. In: MEMOCODE'10, pp. 169–178 (2010)
3. Bellows, P., Hutchings, B.: JHDL—an HDL for reconfigurable systems. In: Annual IEEE Symposium on Field-Programmable Custom Computing Machines (1998)
4. Berry, G.: The constructive semantics of pure Esterel (1999). http://www-sop.inria.fr/esterel.org/files/
5. Bird, R.: Pearls of Functional Algorithm Design. Cambridge University Press, Cambridge (2010)

6. Bird, R., de Moor, O.: Algebra of Programming. International Series in Computing Science, vol. 100. Prentice Hall, New York (1997)
7. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: hardware design in Haskell. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, pp. 174–184 (1998)
8. Chakravarty, M.M.T., Keller, G., Peyton Jones, S.: Associated type synonyms. In: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming, pp. 241–253. ACM, New York (2005)
9. Claessen, K.: Embedded languages for describing and verifying hardware. Ph.D. thesis, Dept. of Computer Science and Engineering, Chalmers University of Technology (2001)
10. Claessen, K., Sands, D.: Observable sharing for functional circuit description. In: Proc. of Asian Computer Science Conference (ASIAN). Lecture Notes in Computer Science. Springer, Berlin (1999)
11. Elliott, C., Finne, S., de Moor, O.: Compiling embedded languages. J. Funct. Program. **13**(2), 9–27 (2003)
12. Erkök, L.: Value recursion in monadic computations. Ph.D. thesis, OGI School of Science and Engineering, OHSU, Portland, Oregon (2002)
13. Erkök, L., Launchbury, J.: A recursive do for Haskell. In: Haskell Workshop'02, Pittsburgh, Pennsylvania, USA, pp. 29–37. ACM, New York (2002)
14. Gill, A.: Type-safe observable sharing in Haskell. In: Proceedings of the Second ACM SIGPLAN Haskell Symposium, Haskell '09, pp. 117–128. ACM, New York (2009)
15. Gill, A.: Declarative FPGA circuit synthesis using Kansas Lava. In: The International Conference on Engineering of Reconfigurable Systems and Algorithms (2011)
16. Gill, A., Farmer, A.: Deriving an efficient FPGA implementation of a low density parity check forward error corrector. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP '11, pp. 209–220. ACM, New York (2011)
17. Gill, A., Neuenschwander, B.: Handshaking in Kansas Lava using patch logic. In: Practical Aspects of Declarative Languages. LNCS, vol. 7149. Springer, Berlin (2012)
18. Gill, A., Bull, T., Kimmell, G., Perrins, E., Komp, E., Werling, B.: Introducing Kansas Lava. In: Proceedings of the Symposium on Implementation and Application of Functional Languages. LNCS, vol. 6041. Springer, Berlin (2009)
19. Gill, A., Bull, T., Farmer, A., Kimmell, G., Komp, E.: Types and type families for hardware simulation and synthesis: the internals and externals of Kansas Lava. In: Proceedings of Trends in Functional Programming. LNCS, vol. 6546. Springer, Berlin (2010)
20. Gill, A., Bull, T., DePardo, D., Farmer, A., Komp, E., Perrins, E.: Using functional programming to generate an LDPC forward error corrector. In: Proceedings of the IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM '11, pp. 133–140. IEEE Comput. Soc., Los Alamitos (2011)
21. Hutton, G.: The Ruby Interpreter. Research Report 72, Chalmers University of Technology (1993)
22. IEEE Std 1164-1993, IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164) (1993). doi:10.1109/IEEESTD.1993.115571
23. Jantsch, A., Sander, I.: Models of computation and languages for embedded system design. IEE Proc., Comput. Digit. Tech. **152**(2), 114–129 (2005). Special issue on Embedded Microelectronic Systems
24. Jones, G., Sheeran, M.: Circuit design in ruby. In: Staunstrup, J. (ed.) Formal Methods for VLSI Design, pp. 13–70. Elsevier, Amsterdam (1990)
25. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: 2nd USENIX Conference on Domain Specific Languages (DSL'99), Austin, Texas, pp. 109–122 (1999)
26. McBride, C., Patterson, R.: Applicative programing with effects. J. Funct. Program. **18**(1), 1–13 (2008)
27. Moon, T.K.: Error Correction Coding: Mathematical Methods and Algorithms. Wiley-Interscience, Hoboken (2005)
28. Naylor, M., Runciman, C.: The reduceron: widening the von Neumann bottleneck for graph reduction using an FPGA. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) Implementation and Application of Functional Languages. Lecture Notes in Computer Science, vol. 5083, pp. 129–146. Springer, Berlin (2008)
29. O'Donnell, J.: Generating netlists from executable circuit specifications in a pure functional language. In: Functional Programming, Glasgow, 1992. Workshops in Computing, pp. 178–194. Springer, Berlin (1992)
30. O'Donnell, J.: From transistors to computer architecture: teaching functional circuit specification in hydra. In: Hartel, P., Plasmeijer, R. (eds.) Functional Programming Languages in Education. Lecture Notes in Computer Science, vol. 1022, pp. 195–214. Springer, Berlin (1995)
31. O'Donnell, J.T.: Hardware description with recursion equations. In: Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and Their Applications, pp. 363–382 (1987)

32. O'Donnell, J.T.: Embedding a hardware description language in template Haskell. In: Domain-Specific Program Generation. Lecture Notes in Computer Science, vol. 3016, pp. 143–164. Springer, Berlin (2004)
33. O'Donnell, J.T.: Overview of Hydra: a concurrent language for synchronous digital circuit design. Information **9**(2), 249–264 (2006)
34. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries—The Revised Report. Cambridge University Press, Cambridge (2003)
35. Sander, I.: System modeling and design refinement in ForSyDe. Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden (2003)
36. Sheeran, M.: muFP, a language for VLSI design. In: LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, pp. 104–112. ACM, New York (1984)
37. Singh, S.: Designing reconfigurable systems in Lava. In: International Conference on VLSI Design, p. 299 (2004)