Types and Type Families for Hardware Simulation and Synthesis

The Internals and Externals of Kansas Lava

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

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

Abstract. In this paper, we overview the design and implementation of our latest version of Kansas Lava. Driven by needs and experiences of implementing telemetry circuits, we have made a number of recent 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, sequential, and enabled signals. We introduce these abstractions, as well as our new abstractions for memory and memory updates. Internally, we found the need to represent unknown values inside our circuits, so we made aggressive use of type families to lift our values 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.

1 Introduction

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, the given name for a family of Haskell based hardware description libraries, is an idiomatic way of expressing hardware in Haskell which allows for simulation and synthesis to hardware. In this paper, we explore the internal and external representation of a `Signal` in Kansas Lava, and how different representations of signal-like concepts work together in concert.

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 xs = [ length [ () | True <- take n xs ] | n <- [0..]]
```

Of course, this function is not a reasonable implementation. In practice, we could use a function defined in terms of its previous result.

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

Haskell programmers get 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 share common thinking with hardware designers when designing cooperating processes communicating 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 Word32
counter inc = res
  where res = mux2 inc (old + 1,old)
        old = delay 0 res
```

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

At KU, we developed a new version of Lava, which we call Kansas Lava[6], to help generate of a specific set of rather complex circuits that implement high-performance high-rate forward error correction codes. This paper discusses our experiences of attempting to use our new Lava. By way of background, the three main features in this original version of Kansas Lava were:

– Dual-use `Signal`. That is, we can use the same `Signal` 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, called `Matrix` takes two types, a representation of size, and the type of the elements in the matrix itself.
– Use of IO-based reification [5] for graph capture. The loop in the `counter` example above becomes a list of primitives and connections internally, making VHDL generation straightforward.

So what went wrong and what worked well with Kansas Lava? In summary, we found the need to make the following changes:

- We fracture our `Signal` type (as used above) into two types `Seq` and `Comb`, for representing values generated by sequential and combinatorial circuits, and connect the two types with a type class (section 3).
- We then introduce functions for commuting types that contain our `Seq` and `Comb`, giving a representation agility we found necessary (section 4).
- We add a phantom type for clock domains, which allows us to represent circuits with multiple clocks in a way that ensures we have not misappropriated our clocks (section 5).
- We introduce the possibility of unknown values into our simulation embedding (section 6).
- With these building blocks, we provide various simple *protocols* for hardware communications (section 7).

Furthermore, this paper makes the following contributions:

- We document the shortcomings of our original straightforward implementation of the Lava ideas and our new solutions and design decisions.
- Some options we have chosen were not available to the original Lava developers. In particular, type families [3] are a recent innovation. 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.

3 Sequential and Combinatorial Circuits

Haskell is a great host for Domain Specific Languages (DSL), like Kansas Lava. 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 little language, customized to a specific problem domain. There are two flavors of embedded DSLs:

- First, DSLs that use a **shallow embedding**, where values are computed with directly. Most definitions of Monads in Haskell are actually shallow DSLs, for example. The result of a computation in a shallow DSL is a value.
- Second, 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, and this structure can be used to compute a value.

In our first iteration of the embedded DSL Kansas Lava [6], 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 is unusual, in that it contains a shallow and deep embedding at the same time. This is so the same `Signal` can be both executed and reified into VHDL, as directed by the Kansas Lava user. The shallow embedding (direct values) was encoded as an infinite stream of direct values and the deep embedding (abstract syntax tree) was a phantom typed [9] abstract syntax tree. Slightly simplified for clarity, we used:

```
data Signal a = Signal (Stream a) (D a)

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

type D a = AST

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 can write operations that support both the shallow and deep components of `Signal`. We could write functions like `and2`, which acted over `Signal Bool` in this manner.

```
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. For representing a sequential use of `and2`, values that change over time stream in and the result, also a value that changes over time, streams out. However, often we were writing combinatorial circuits, to be later instantiated as components inside a larger sequential circuit. Combinatorial circuits have no state; like a function call they take and return values without history. So the *types* of our functions were saying streams of values (which can have an arbitrary historical memory), but we wanted to think of our circuits as being run many times, independently.

The analog in functional programming is base values compared to infinite lists or streams of these base values. We wanted a way of taking an operation over a base value, like `Bool`, and lifting it into an operation over streams, where the original operation is applied point-wise. Haskell has many such lift functions: `map`, `zipWith`, `liftM`, `liftA2`, to name a few. With this in mind, we defined the retrospectively obvious relationship.

```
data Comb a = Comb a AST        -- working definition; to be refined
data Seq a = Seq (Stream a) AST -- working definition; to be refined

liftSeq0 :: Comb a -> Seq a
liftSeq1 :: (Comb a -> Comb b) -> Seq a -> Seq b
liftSeq2 :: (Comb a -> Comb b -> Comb c) -> Seq a -> Seq b -> Seq c
```

In this way, we can be completely explicit about what a value means, and how it would be generated. But what type do we give for `and2`? Both a `Seq` and a `Comb` variant are reasonable, for we certainly do not want to use a lift every time we define or use an operation over `Seq`.

It is worth noting at this point the connection with our "Observable" value `O` in ChalkBoard [10], which serves the same purpose as `Comb` here. In ChalkBoard, there is a clear and expressive distinction between using `O` (operating over individually sampled pixels) and `Board` (operating of regions of sampled pixels). The idea of separating circuit definitions into `Comb` and `Seq` came directly from our work on ChalkBoard. However, in circuits, we want some way of writing the same circuit for sequential and combinatorial use. So at this point, we use a classical Haskell type class, `Signal`, to join together `Comb` and `Seq`.

```
class Signal sig where
  liftSig0 :: Comb a -> sig a
  liftSig1 :: (Comb a -> Comb b) -> sig a -> sig b
  liftSig2 :: (Comb a -> Comb b -> Comb c) -> sig a -> sig b -> sig c

instance Signal Comb where { ... }
instance Signal Seq where { ... }
```

We use the name $\texttt{liftSig}_n$ to reflect we are now lifting into `Signal`, and both `Comb` and `Seq` support this overloading. With this new class, we can give a more general type for `and2`:

```
and2 :: (Signal sig) => sig Bool -> sig Bool -> sig Bool
and2 = liftSig2 $ \ (Comb a ae) (Comb b be)
                        -> Comb (a && b) (Entity "and2" [ae,be])
```

By making all primitives that can be both combinatorial and sequential use the type class `Signal`, we can write circuits and give them either a combinatorial (`Comb`) type, a sequential (`Seq`) type, or allow the general overloading. The types are driving the specification of what we can do with the code we have written.

If a specific primitive is used that is sequential, then the entire circuit will correctly inherent this. One combinator that makes this happen is `register`.

```
register :: Comb a -> Seq a -> Seq a -- working definition; to be refined
```

This takes a default combinatorial value, and a sequential signal to be delayed by one (implicit) clock cycle. The types make the shapes of values – how they might be built – explicit.

To summarize, changing `Signal` into a class rather than a data-type and providing overloaded combinatorial and sequential logic gives us three possible ways of typing many circuits. We can use the *class*, and say this is a circuit that can be executed as either combinatorially or sequentially; we can use `Comb` to specify the one shot at a time nature of combinatorial logic; or we can use `Seq` to specify that the circuit is to be only used in a sequential context. Furthermore, we can mix and match these three possible representation specifications, in the way `register` takes both a `Comb` and a `Seq`. For this extra flexibility to work in a pragmatic way, we need some way of normalizing the `Comb`-based function into a suitable type for lifting, which can be done using the type commuting functionality discussed in the next section.

4 Commutable Functors and Signals

Which of these two types for `halfAdder` makes more sense?

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

The first is easier to write, because we can directly address and return the tuple. The second may be used by using our lift functions above. We found ourselves going around in circles between the two approaches. Both could have the same interpretation or meaning, but can we somehow support both inside Kansas Lava without favoring one or the other? To enable this, we invoke type families [3].

We have a class `Pack`, which signifies that a `Signal` can be used inside or outside a specific structure. The type translation from the packed (structure inside, `Signal` on the outside) to unpacked (structure outside, `Signal` on the inside) is notated using an type family inside the class `Pack`:

```
class (Signal sig) => Pack sig a where
 type Unpacked sig a
 pack :: Unpacked sig a -> sig a
 unpack :: sig a -> Unpacked sig a
```

This class says we can `pack` an `Unpacked` representation, and `unpack` it back again. The reason for two operations is that a packed structure does not need to be isomorphic to its unpacked partner. Though every packed structure with an instance will have one specific unpacked representation to use, two types can share unpacked representations.

Reconsidering the example above, which was over the *structure* two-tuple, we have the instance:

```
instance (Wire a, Wire b, Signal sig) => Pack sig (a,b) where
        type Unpacked sig (a,b) = (sig a, sig b)
        -- types given, not the code
        pack :: (sig a, sig b) => sig (a,b)
        unpack :: 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` flavors given at the start of this section.

```
-- unpacked version
halfAdder :: (Comb Bool,Comb Bool) -> (Comb Bool,Comb Bool)
halfAdder (a,b) = (a ‘xor2‘ b,a ‘and2‘ b)

-- packed version
halfAdder :: Comb (Bool,Bool) -> Comb (Bool,Bool)
halfAdder inp = pack (a ‘xor2‘ b,a ‘and2‘ b)
  where (a,b) = unpack inp
```

As can be seen, there is not a huge difference in clarity, because of the generic nature of the `pack`/`unpack` pair. Both styles can be used depending on context and needs.

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

```
instance (Wire a, Signal sig) => Pack sig (Maybe a) where
        type Unpacked sig (Maybe a) = (sig Bool,sig a)
        -- types given, not the code
        pack :: (sig Bool,sig a) => sig (Maybe a)
        unpack :: sig (Maybe a)  -> (sig Bool,sig a)
```

Consider an alternative type for unpacking a `Signal` of `Maybe`.

```
unpack :: (...) => sig (Maybe a) -> Maybe (sig a) -- WRONG
```

Here, the result is a single result, and can not encode a stream of `Nothing`s and `Just` values. So in general, the unpacked structure *must* be able to notate the complete space of the signal to be packed. Table 1 lists the types `pack` and `unpack` supports.

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, using types in an agile way to represent the intent of 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` [11]. In our case the `pack` and `unpack` are tied to our `Signal` overloading; we are commuting (or moving) the `Signal`.

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 x a)`	`Matrix x (sig a)`
`sig (StdLogicVector x)`	`Matrix x (sig Bool)`

Table 1. Packed and Unpacked Pairs in Kansas Lava

5 Phantom Types for Clock Domains

What does `Seq a` mean? `Seq` is a sequence of values, either separated by `:~` in our shallow embedding, or by sampling values at the rising edge of an implicit clock. Our telemetry circuits have, by design, multiple clock domains. That is, different parts of a circuit beat to different drums. The question is: can we use the Haskell type system to express separation of these clock domains?

We use a phantom type [9] to express the clock domain. Our new `Comb` and `Seq` have the following definitions.

```
data Comb a = Comb a AST            -- working definition; to be refined
data Seq clk a = Seq (Stream a) AST -- working definition; to be refined
```

`Comb` remains unaffected by clocks, and the lack of `clk` in the type reflects this. `Seq` now has a phantom clock value, which is notationally a typed connection to the clock that will be used to interpret the result. How do we use this clocked `Seq`? For basic gates, the same interpretation (and therefore phantom type) is placed on the inputs as the output. Consider `and2`, at the `Seq clk` type:

```
and2 :: Seq clk Bool -> Seq clk Bool -> Seq clk Bool
```

As would be expected, both inputs to `and2`, as well as the output, have the same clock for interpretation of timing.

For latches and registers, we maintain this type annotation of shared interpretation. We can now give our final type for `register`:

```
register :: (...) => Env clk -> Comb a -> Seq clk a -> Seq clk a
```

We can see that `register` combinator takes a combinatorially generated default value and an input which is interpreted using the same clock domain as the output.

The `Env` is a way of passing an environment to the `register` combinator. Specifically, one of the design questions is: do you have an implicit or explicit clock? From experience, we found adding an enable signal to our registers useful, because we could test our FPGA circuits at a much slower speed than real clock speed. Also, we needed to pass in a reset signal. Both clock enable and reset are something that you could imagine wanting to generate from other Kansas Lava circuits under some circumstances. We chose to explicitly have a typed environment, and pass in the clock enable, the reset, and at the same time, an explicit clock.

```
data Env clk = Env { clockEnv  :: Clock clk
                   , resetEnv  :: Seq clk Bool
                   , enableEnv :: Seq clk Bool
                   }
```

`Env` is *not* exported abstractly, and is an explicitly passed value. So the programmer is free to pattern match on `Env` if needed, adding extra logic to the reset and/or enable.

Figure 1 illustrates the timing properties of `register`, and clarifies what `register` actually does in the context of an environment. The `clk`, `rst`, and `en` are the environment. As reflected in the type, there are two inputs, and one output.

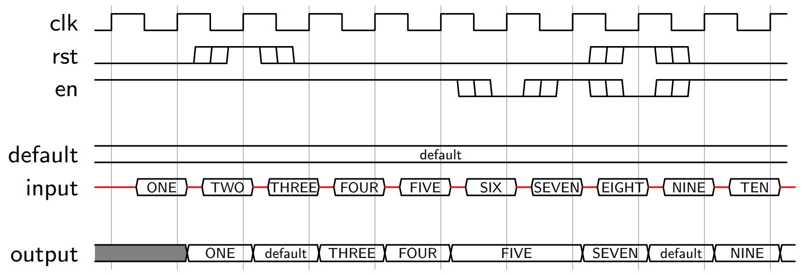


Fig. 1. `register` timing diagram

One remaining question is the representation of the data type `Clock` itself. `Clock` is not a sequence of values interpreted using a clock; it *is* the clock. We considered something like `Seq clk ()`, but this would make it possible to invent nonsense clocks, and leads to convoluted semantics. So we have a new type for `Clock` which, like `Seq`, has a shallow (simulation) and deep (generation) aspect.

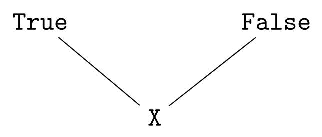
```
data Clock clk = Clock Rational (D clk)
```

The `Rational` is the clock frequency, in Hz, and the `D clk` is the circuit used to generate this clock (typically provided as an input to the whole circuit). Accurate simulation of possible race conditions based on two differently clocked circuits is a hard problem, but we use the `Rational` argument to approximate different clock rates at the interaction boundaries between clock domains.

We provide standard wrappers round FIFO blocks and block RAMs, as recommended by Xilinx, for interfacing at the Kansas Lava level between clock domains. We will see an example of a Kansas Lava multi-clock block RAM in section 7.

6 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 bit in VHDL captures this with an `X` notation. Such unknowns are introduced externally, or from reset time, and represent a value outside the standard values in Haskell. For example, when modeling hardware and transmitting a boolean, we essentially want a lifted domain.



In this form, we have a `Maybe` type. But the situation is more complex for structured types. Consider a `Signal` (`Seq` or `Comb`) 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 again solve this problem using type families. We introduce a new class, `Wire`, which captures the possibility of unknowns and other issues concerning the representation of the values in wires.

```
class Wire w where
        -- A way of adding unknown inputs to this wire.
        type X w
        -- check for bad things
        unX :: X w -> Maybe w
        -- and, put the good or bad things back.
        optX :: Maybe w -> X w

pureX :: (Wire w) => w -> X w   -- the pure of the X type
pureX = optX . Just
```

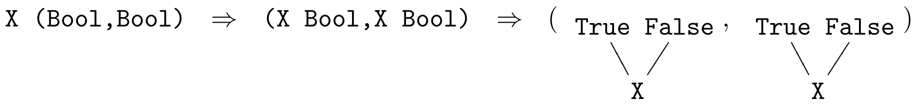
The type family `X` means lifted value, and there is a way of extracting the value from the `X`, using `unX`, and a way of injecting a value into an `X`, using `optX`. In this way, `X` of a specific type represents a type that can admit unknown value(s). For example, our instance for `X Bool` uses `Maybe`.

```
instance Wire Bool where
        type X Bool   = Maybe Bool
        optX (Just b) = return b
        optX Nothing  = Nothing
        unX (Just v)  = return v
        unX (Nothing) = Nothing
```

Our instance for tuples uses tuples of `X`.

```
instance (Wire a, Wire b) => Wire (a,b) where
        type X (a,b)      = (X a, X b)
        optX (Just (a,b)) = (pureX a, pureX b)
        optX Nothing      = ( optX (Nothing :: Maybe a)
                            , optX (Nothing :: Maybe b)
                            )
        unX (a,b) = do x <- unX a
                       y <- unX b
                       return (x,y)
```

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


X (Bool,Bool) ⇒ (X Bool,X Bool) ⇒ (
True False , True False)
X
X

So there are 9 possible values for the pairing of the two boolean signals.

We can now give our complete types for `Comb` and `Seq`:

```
data Comb a    = Comb (X a) (D a)
data Seq clk a = Seq (Stream (X a)) (D a)
```

One complication is that all function primitives now need to be written over our `X` type. However, only the primitives will know how they want to handle unknowns anyway, and we use the built-in support for the `Maybe` monad where possible. So what happens 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:

```
and2 = liftS2 $ \ (Comb a ae) (Comb b be) ->
                Comb (liftM2 (&&) a b)
                     (Entity "and2" [ae,be])
```

The tricky part comes when we work with any type of polymorphism, including containers or selectors. Consider the type for `mux2`:

```
mux2 :: (Signal sig, Wire a) => sig Bool -> (sig a,sig a) -> sig a
```

We can see from the type that we have two arguments, a signal of `Bool`, which is our conditional control, and a pair of signals. Semantically, `mux2` dynamically chooses one of the tupled signals depending on the `Bool` signal. We want to capture this behavior in our shallow embedding.

Our implementation (remember this is *internal* code, not what the Kansas Lava user would see or need to write) for `mux2` is:

```
mux2 :: forall sig a . (Signal sig, Wire a)
     => sig Bool -> (sig a,sig a) -> sig a
mux2 i ~(t,e)
        = liftSig3 (\ ~(Comb i ei)
                      ~(Comb t et)
                      ~(Comb e ee)
                        -> Comb (mux2shallow (witness :: a) i t e)
                                (Entity "mux2" [ei,et,ee])
                 ) i t e

mux2shallow :: forall a . (Wire a) => a -> X Bool -> X a -> X a -> X a

witness = error "witness"
```

At first glance, this is similar in flavor to `and2`. The main new trick here is the creation of a type witness to pass to `mux2shallow`, using a scoped type variable [8], and the explicit use of `forall` to force the scoping. Without the type witness, `mux2` will never type-check the call to `mux2shallow`. The problem is that without the witness, there is no way to unify the other arguments of `mux2shallow` to their expected types. Type families, like `X`, are not injective, so `X a ~ X b` (`X a` unifies with `X b`) does not imply `a ~ b`, unlike (most) traditional type constructors. We use this trick for passing type witnesses all over our implementation.

The implementation of `mux2shallow` can now focus on the problem at hand, choosing a signal.

```
mux2shallow :: forall a . (Wire a) => a -> X Bool -> X a -> X a -> X a
mux2shallow _ i t e =
   case unX i :: Maybe Bool of
       Nothing -> optX (Nothing :: Maybe a)
       Just True -> t
       Just False -> e
```

We extract the value from the `X Bool`, which we consider abstract here, and if it is `True` or `False`, we choose a specific signal, or generate an unknown value if `X Bool` is the unknown. Again, we need to use scoped type variables, though the behavior of `mux2shallow` should be clear from its definition. This way of thinking about supporting unknowns, where we extract values from `X`, handle unknowns algorithmically, and repackage things using `X` again is a common pattern. We have the flexibility to include a basic hardware style thinking about unknown values, and make extensive and pervasive use of use of it to provide hardware style semantics to Kansas Lava users.

7 Protocols for Signals

On top of the generality of the above types, we have constructed a number of type idioms that make building circuits easier. We have three idioms, validity (`Enabled`), memory (`Memory`), and updates to memory (`Pipe`).

Validity of a value on a wire is a general concept. Often, this is done using an *enable* boolean signal that accompanies another value signal. When the enable signal is `True`, the value signal is present, and should be consumed. When the enable signal is `False`, then the value signal is arbitrary, and should be ignored. In Kansas Lava, we define this enable concept using:

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

We could have used a literal pairing of `(Bool,a)`, but we found this to be cumbersome in practice. Instead, we let the *representation* be the pair, and the *value* be lifted using a `Maybe`.

```
instance (Wire a) => Wire (Maybe a) where
        type X (Maybe a) = (X Bool, X 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, and our implementations have a data-flow feel where the enable is the token flowing through the circuit. The concept of `Enabled`, however, is distinct from our concept of unknown. `Enabled` is a user *observable* phenomenon, and user-level decisions can be make based on the validity bit; while unknown values are a lower-level shallow embedding implementation trick to allow our combinatorial circuits to behave more like hardware.

Reading memory is a sequential function, from addresses to contents:

```
type Memory clk a d = Seq clk a -> Seq clk d
```

Given a `Memory` we can send in addresses using function application, and expect back values, perhaps after a short discrete number of clock cycles. We defer the actual *creation* of the `Memory` for a moment, but observe that the read requests and the values being read share the same clock.

Writing memory is done using a sequence of address-datam write request pairs. Specifically:

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

The pipe idiom gives an optional write command, saying write this datam (`d`) to this address (`a`), if enabled. The name `Pipe` is used as a mnemonic for pushing something small though a pipe, one piece at a time. Of course, `Pipe` is a general concept, and could be used as a simple protocol in its own right.

Returning to the question of how we actually construct a `Memory`, we make two observations. Our first observation is that there is an interesting symmetry between `Memory`, which has a datum answer for every address, and `Pipe`, for which, if we look backwards in time, we can also observe the relevant address write. Given access to history, both represent the same thing: access to values larger than those which a single signal can encode, though the expected implementation for both is completely different. This symmetry gives our design of memory generation. Specifically, we have a function that takes a `Seq` of `Pipe` and returns a `Memory`:

```
-- working definition; to be refined
pipeToMemory :: (...) => Env clk -> Seq clk (Pipe a d) -> Memory clk a d
```

Our second observation is that the clocking choices from the `Pipe` input can be completely independent to the clocking choices for the reading of the memory. There is no reason in hardware, other than the complexity of implementation, that a memory *must* read and write based on the same clock. We can reflect this possibility in our type. So, given a stream of such `Pipe`-based write requests, we can construct a `Memory`:

```
pipeToMemory :: (...)
             => Env clk1
             -> Env clk2
             -> Seq clk1 (Pipe a d)
             -> Memory clk2 a d
```

Here, `pipeToMemory` has two clock domains, `clk1` and `clk2`. If the clocks are actually the same clock, then Kansas Lava can generate simpler hardware that can rely on this, but the full generality is to available to the user.

Given a sequence of enabled addresses, we can also turn a `Memory` back into a `Pipe`:

```
memoryToPipe :: (...)
             => Env clk
             -> Memory clk a d
             -> Seq clk (Enabled a)
             -> Seq clk (Pipe a d)
```

This time, because we are *reading* memory, we are in the same clock domain for all arguments to `memoryToPipe`. The types force this, and make this design choice clear. A dual clock domain version of `memoryToPipe` is possible, but the semantics do not generalize into the same single clock domain version, because an extra latch would be required to handle the clock domain impedance.

Together `Enabled`, `Memory` and `Pipe` form a small and powerful algebraic framework over `Seq`, able to express many forms of sequential communications. We intend to exploit this in the future, using it for guiding Kansas Lava program derivations.

8 Related Work

The ideas behind Lava, or in general, programs that describes and generates hardware circuits, are well explored. The original ideas for Lava itself can be traced back to the hardware description language μFP [13]. A pattern in the research that followed was the common thinking between functional programming and hardware descriptions. A summary of the principles behind Lava specifically can be found in [2] and [4].

The ForSyDe system [12], which is also embedded in Haskell, addresses many of the same concerns as Kansas Lava. Like Kansas Lava, ForSyDe provides both a shallow and deep embedding, though in ForSyDe this is done via two distinct types and using the Haskell `import` mechanism. Additionally ForSyDe provides a rich design methodology on top of the basic language, and supports many "models of computation" [7]. The principal differentiator of Kansas Lava is its use of type familes, 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.

Kansas Lava is a modeling language. It models communicating processes, currently via synchronous signals. There are several other modeling languages that share this basic computational basis, for example Esterel [1]. There are many other models for communicating processes, and each model family has many language-based implementations. The overview paper written by Jantsch, et. al. [7] gives a good summary of this vast area of research.

9 Conclusions

In this paper, we have seen a number of improvements to Kansas Lava, unified by a simple principle: how can we use types to express the nature and limitation of the computation being generated by this Kansas Lava expression? We have made many more changes than we anticipated to Kansas Lava to turn it into a useful VHDL generation tool. For now, however, Kansas Lava as a language has somewhat stabilized.

Each change was initiated because of a specific shortcoming. We separated the types of combinatorial and sequential circuits because we were writing combinatorial circuits and imprecisely using the universal signal type, which was sequential. We provided generic mechanisms for commuting signals, so that we could have our cake (write functions in the style we find clearest) and eat it too (lift these functions if necessary). We used phantom types for clock domains, because we do not trust ourselves to properly render circuits in with implicit nature of interpretation of signals under multiple clocks. Finally, we allow the representation of the unknown in our simulations despite the pervasive consequence of the choice, because our simulations were not matching our experience with generated VHDL.

We hope that we can build up a stronger transformationally based design methodology round Kansas Lava. Currently we have a number of large circuits where we have translated 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 Haskell. The commuting of signals has turned out to be extremely useful when deriving our circuits from higher-level specifications. Writing correct, efficient circuits is hard, and we hope to address at least some of these circuits as candidates for our methodologies.

References

1. G. Berry. The constructive semantics of pure Esterel. http://www-sop.inria.fr/esterel.org/files/, 1999.
2. Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.
3. Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, New York, NY, USA, 2005. ACM.
4. Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Dept. of Computer Science and Engineering, Chalmers University of Technology, April 2001.
5. Andy Gill. Type-safe observable sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, Sep 2009.
6. Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing Kansas Lava. In *Proceedings of the Symposium on Implementation and Application of Functional Languages*, Sep 2009.
7. Axel Jantsch and Ingo Sander. Models of computation and languages for embedded system design. *IEE Proceedings on Computers and Digital Techniques*, 152(2):114–129, March 2005. Special issue on Embedded Microelectronic Systems.
8. Simon Peyton Jones and Mark Shields. Lexically scoped type variables. http://research.microsoft.com/en-us/um/people/simonpj/papers/scoped-tyvars/.
9. Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, October 1999.
10. Kevin Matlage and Andy Gill. ChalkBoard: Mapping functions to polygons. In *Proceedings of the Symposium on Implementation and Application of Functional Languages*, Sep 2009.
11. Conor McBride and Ross Patterson. Applicative programing with effects. *Journal of Functional Programming*, 16(6), 2006.
12. Ingo Sander. *System Modeling and Design Refinement in ForSyDe*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, April 2003.
13. Mary Sheeran. mufp, a language for vlsi design. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 104–112, New York, NY, USA, 1984. ACM.