Deriving an Efficient FPGA Implementation of a Low Density Parity Check Forward Error Corrector

Andy Gill      Andrew Farmer
The University of Kansas
{andygill, afarmer}@ittc.ku.edu

Abstract
Creating correct hardware is hard. Though there is much talk of using formal and semi-formal methods to develop designs and implementations, in practice most implementations are written without the support of any formal or semi-formal methodology. Having such a methodology brings many benefits, including improved likelihood of a correct implementation, lowering the cost of design exploration and lowering the cost of certification. In this paper, we introduce a semi-formal methodology for connecting executable specifications written in the functional language Haskell to efficient VHDL implementations. The connection is performed by manual edits, using semi-formal equational reasoning facilitated by the worker/wrapper transformation, and directed using commutable functors. We explain our methodology on a full-scale example, an efficient Low-Density Parity Check forward error correcting code, which has been implemented on a Virtex-5 FPGA.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Applicative (functional) languages

General Terms Design, Performance, Reliability, Verification

Keywords Lava, DSL, Error Correction, LDPC

1. Introduction
Connecting a clear description of an algorithm to an efficient, hardware based implementation brings many benefits. Real implementations contain many compromises, for example in the dimensions of performance, latency, power consumption, or capacity. In this paper, we discuss in detail a full-scale derivation from a specification of the Low Density Parity Check (LDPC) Forward Error Corrector to an efficient and deployable implementation, paying attention to these compromises.

We use the functional language Haskell both as our specification language and our implementation language. Our specification of the LDPC is a short and direct executable description of the algorithm, transliterated from the standard reference on error correcting codes [Moon 2005]. Our implementation is expressed in a Haskell library called Kansas Lava [Gill et al. 2010]. Kansas Lava takes idiomatic Haskell expressions that resemble hardware descriptions of component connectivity, and generates VHDL that can be synthesized using standard tools like Xilinx’s ISE.

This paper presents a case study of taking the Haskell description of the LDPC and refining it into a Kansas Lava implementation. Having both the specification and implementation in the same host language allows a number of well-understood transformations to be used inside the common framework of Haskell refinement. We make extensive use of basic equational reasoning, Burstall-Darlington style fold/unfold [Burstall and Darlington 1977], and most critically, the recently formalized worker/wrapper meta-transformation [Gill and Hutton 2009].

2. Haskell, Architectures and Hardware
Consider a Haskell implementation of the well-known Game of Life, as described in [Hutton 2007]. The game models a trivial cellular system, played on a 2-dimensional board.

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) (b) (c)

Each square in the board (a) either contains a living cell, marked here with a filled-in circle, or is empty. Each step consists of counting your eight neighbors (b). If you are alive and have two or three living neighbors, then you survive to the next generation (c), otherwise you die (c; dotted circles). Empty squares with three living neighbors spawn a new cell (c; darker circles).

Hutton describes the world of the game using a list of coordinates of living cells.

type Board = [(Int,Int)]

Hutton then defines a function that finds survivors and another that finds births, and uses them to define a stepper function, called nextgen.

survivors, births :: Board -> [(Int,Int)]

nextgen :: Board -> Board
nextgen b = survivors b ++ births b

This is a classic example of using big-step computation to cleanly capture the behavior of a system, something Haskell excels at. The pertinent question for this paper is can this Haskell implementation inform the architecture and implementation of a hardware rendition of the Game of Life in a meaningful way?
In making any transition to hardware, decisions about the layout of computation, or architecture, are critical. This architecture takes into consideration what will be computed where, and what components will communicate, without taking into consideration exact timing and clocking issues. For instance, in our Game of Life, we can imagine each box in the 5x5 grid using the values communicated directly from its neighbors to compute its liveness for the next generation.

Assuming we use the common idiom of streams to represent values that change over time, with each time step being a generation, one possible implementation of an architecture takes a list of streams from all the neighbors and returns a stream representing the liveness of the cell over time.

```
box :: Bool -> [Stream Bool] -> Stream Bool
box init nss = res
  where
    res = init : zipWith step (transpose nss) res
step :: [Bool] -> Bool -> Bool
step ns live
  | live && neighbors == 2 = True
  | live && neighbors == 3 = True
  | not live && neighbors == 3 = True
  | otherwise = False
  where
    neighbors = length (filter id ns)
```

Again, Haskell is a useful descriptive tool, this time for architectural concerns. The type of box implies that this architecture takes inputs from its neighbors, using a single boolean result from each one on each conceptual cycle. No commitment has been made about the specific implementation or timing of this communication channel.

Finally, we can consider a hardware description of box. The obvious implementation in this example would mirror the architecture, having the 8 neighborly values communicated directly. For illustration purposes, we encode this using an 8-tuple. So, given a nominal data-type `Signal`, which represents a wire that encodes a clocked signal, the type of one possible implementation uses an initial boolean value and an 8-tuple of neighbor signals to generate a boolean output signal.

```
type Tuple8 a = (a,a,a,a,a,a,a,a)
box_impl :: Bool
  -> Tuple8 (Maybe (Signal Bool))
  -> Signal Bool
```

Of course, for this example there are other alternative implementations, but working with and coding around implementation specific restrictions is completely typical for reaching efficient implementations. This brings out the critical contribution of the step between the architecture and the implementation, namely the elaboration of the decision of how to communicate between computational blocks. Pictorially, the relationship between the arguments of box and box_impl can be representing using:

![Diagram](represent abstract)

From a refinement perspective, there is an interesting relationship between these two types, and encoding this relationship is the central task of any refinement from architecture to implementation.

If we consider all three versions of the Game of Life, there is a pattern of refinement that can be observed. In Hutton’s example, a big-step computation is being performed on the entire state for each new generation. In the architectural description, the computation has been partitioned and state transitions happen locally. In the final example, both computation and communication have been refined into an efficient, hardware-centric implementation.

This paper contains a case study that follows this specification to architecture to implementation pattern. We first introduce the general worker/wrapper transformation, our main refinement weapon in our derivation (§3). We then introduce the notion of commutable functors, which we use to direct our use of worker/wrapper when performing refinements (§4). Next, we introduce Kansas Lava, our Haskell library for expressing hardware (§5), and our target algorithm, the Low Density Parity Check forward error correcting algorithm, and its executable specification in Haskell (§6). We take this specification and refine it into our LDPC architecture by making decisions about computational layout (§7). Finally, we take the architecture and refine it into our LDPC implementation in Kansas Lava, which executes on an FPGA (§8). We close with performance results, discussions of related work, and our conclusion.

In this paper, we make the following contributions.

- This case study is the first example of the new formalization of the worker/wrapper transformation being used in anger on a large example. The case study also demonstrates the viability and usefulness of worker/wrapper as an informal refinement tool, even without a mechanized implementation.
- We give the intermediate steps of our refinement in detail, as well as the challenges faced and decisions made. The intent is that this study be a prelude to adding some automation to our refinement processes, and knowing useful non-trivial examples helps in the design of our (future) refinement engines.
- Finally, we illustrate the utility of using functors to model and implement communication channels, and the refinement possibilities this opens up. Specifically, commutable functors play a pivotal role in allowing us to plot a course through a derivation and show that following this course gives, as a side effect, a list of proof obligations and other useful artifacts.

3. The Worker/Wrapper Transformation

The worker/wrapper transformation (Gill and Hutton 2009; Peyton Jones and Launchbury 1991) changes how a (typically recursive) function operates using only locally verifiable assumptions. The transformation is a general form of data-refinement (Hoare 1972). As a local transformation that acts on a single function, the worker/wrapper transformation can achieve a global influence on a program by pushing specially crafted coercion wrappers in a systematic way from definition site to call sites. Worker/wrapper has been used as a mechanism to implement strictness analysis optimizations (Jones and Partain 1993), argument transposition (Launchbury and Sheard 1993), constructor specialization (Jones 2007), and in the worker/wrapper journal article memoization (Michie 1968) and CPS translation (Appel 1992; Susman and Steele 1975) are also performed. The transformation is straightforward and remarkably general. For these reasons, the worker/wrapper transformation is an ideal candidate for a meta-refinement in our methodology. Specifically, we will use the ability to change the type over which functions operate to perform a number of type-directed hops from our specification to implementation.
3.1 Worker/Wrapper Pre-Conditions

Three possible pre-conditions for worker/wrapper are:

- \( \text{abs} \circ \text{rep} = \text{id}_A \); or
- \( \text{abs} \circ \text{rep} \circ \text{comp} = \text{comp} \); or
- \( \text{fix}(\text{abs} \circ \text{rep} \circ \text{comp}) = \text{fix} \text{comp} \).

All three follow from the formalization, with each pre-condition directly implying the subsequent, weaker pre-condition. The third pre-condition can actually be observed inside Figure 1 where the splicing is a use of this final rule. The intuition behind all the pre-conditions is that they ensure, specifically for the function \( \text{comp} \), that the alternative type \( B \) can safely be used.

3.2 Example of Worker/Wrapper

To give a short example of worker/wrapper in use, we transform the naïve Haskell definition of \( \text{reverse} \) into an efficient implementation.

\[
\text{reverse :: } [a] \to [a] \\
\text{reverse } xs = \text{comp } xs \\
\text{where } \text{comp } [] = [] \\
\text{comp } (x:xs) = \text{reverse } xs ++ [x] \\
\]

- \( \text{reverse} \) is defined as a computation (comp) that takes a list \([a]\) and returns the reverse of the list, also of type \([a]\). As such, the initial type \( A \) is \([a] \to [a]\). We choose the type \([a] \to H a\) for \( B \), where \( H a \) is a data structure that acts like a list, but efficiently implements appending to the end of the list. There are a number of possible such implementations (Hughes 1986; Okasaki 1998).

- Our higher-order functions \( \text{abs} \) and \( \text{rep} \) (not given) contain the functions \( \text{absH} \) and \( \text{repH} \):

  -- converting an \( H \) list into a regular list
  \( \text{absH :: } H a \to [a] \)

  -- converting a regular list into an \( H \) list
  \( \text{repH :: } [a] \to H a \)

  It can be shown that our \( \text{abs} \) and \( \text{rep} \) satisfy \( \text{abs} \circ \text{rep} = \text{id}_A \), meeting one of the worker/wrapper pre-conditions.

Wrap the computation in \( \text{absH} \) and \( \text{repH} \) functions, which form an identity.

\[
\text{reverse :: } [a] \to [a] \\
\text{reverse } xs = \text{absH} \circ \text{repH} \circ \text{comp } xs \\
\text{where } \text{comp } [] = [] \\
\text{comp } (x:xs) = \text{reverse } xs ++ [x] \\
\]

Note that reverse and comp are mutually recursive. Unroll the recursive call to reverse once, so comp calls itself.
Create a new worker function, which is the expression contained in the outer absH, namely: repH (comp xs). After this, reverse is a wrapper\(^9\) that calls worker.

```haskell
reverse :: [a] -> a
reverse xs = absH (worker xs)

worker :: [a] -> H a
worker xs = repH (comp xs)
  where comp [] = []
        comp (x:xs) = absH (worker xs) ++ [x]
```

Now the optimization of the worker can begin in earnest\(^9\). We do not repeat the wrapper reverse again, because it remains unchanged. After some well-understood transformations, we reach:

```haskell
worker :: [a] -> H a
worker [] = repH []
worker (x:xs) = snocH (worker xs) x
```

In this specific case, repH can fuse with absH, and we can reach the optimized solution, reaching:

```haskell
worker :: [a] -> H a
worker [] = repH []
worker (x:xs) = snocH (worker xs) x
```

The whole definition of worker now operates using the new type H a, and we have successfully translated a computation over A into a computation over B\(^10\). Note that the wrapper (here, reverse) is no longer recursive, and its body can be cloned and inlined at all call sites.

This example was presented in a first order, syntactical style, but the transformation itself is higher-order. Exactly the same framework can be used to change the argument types, or even the number of arguments. Indeed, as long as A and B can be used as alternative representations of the same underlying computation, workerwrapper can be applied. Furthermore, the transformation can be repeated, changing to yet another type. We are going to use this chaining ability to help perform the key step of turning an executable specification into an efficient implementation.

### 4. Functors are our Friends

Given the ability to perform type-directed transformation using worker/wrapper, we now present a small set of types that model different components of computation and storage that will be used as our alphabet for both the executable specification and the implementation. These components are all modeled after a mathematical structure called a functor. Formally, a functor is mapping between categories in category theory, and we use the term functor in this paper in the more general sense than the specific Haskell class `Functor`.

In Haskell the class `Functor` provides an implementation for a function `fmap`, a mnemonic for functor-map, which obeys two properties.

```haskell
class Functor f where
  -- law: fmap id = id
  -- law: fmap f . fmap g = fmap (f . g)
  fmap :: (a -> b) -> f a -> f b
```

There are many Haskell data-structures that are functors, for example Haskell lists and `Maybe`. Take as a concrete example streams. A `Stream` and its functor instance can be defined as:

```haskell
data Stream a = Cons a (Stream a)

instance Functor Stream where
  fmap (Cons x xs) = Cons (f x) (fmap f xs)
```

It is straightforward to show that this implementation satisfies the two laws.

```haskell
{(arch) (List (Stream Bool))
  {1} (List (Stream (Maybe Bool)))
  {2} (List (Maybe (Stream Bool)))
  {3} (Tuple8 (Maybe (Stream Bool)))
  {4} (Tuple8 (Maybe (Signal Bool)))
}
```

**Figure 2.** Refining the Game of Life: Architecture to Implementation

#### 4.1 The Big Idea

A functor often represents a container, like a list, that contains zero or more items of a given type. Functors can also be used to represent computation models, like an infinite stream of data. Our thesis is that functor-like structures can be effectively used to represent structures that describe real-world communication inside architectural descriptions and hardware solutions. As such, these functor chains become guideposts to performing larger refinement.

Reconsidering the example from Section 2, we refine from an architecture of a list of streams of booleans, and reach a fixed-sized tuple of optional signals of boolean. Figure 2 illustrates the steps taken in this example, from architecture to implementation. For symmetry, we will write the list type using `List`, and not “[]”, for the remainder of the paper. In the example, we are performing three possible translations: adding a functor (step 1); commuting functors (step 2); replacing one functor with another (steps 3 and 4). Each translation has its respective proof obligation. This type-level explanation can be used to explain the stepwise thinking of the implementation refinement.

#### Lifting the Bool

We choose to implement the Game of Life where the edges do not wrap back on each other. As such, neighbors that would be outside the boundary wall are always zero. We want to optimize the code generated for the edges. Though in this example this optimization makes little difference, we intend to implement operations acting on a sparse matrix in our large case study, and in that case, the ability to generate no code for zeros is critical to reaching an efficient implementation. The `Maybe` functor is used to represent the case of being always zero because the neighbor is out-of-bounds.

Rather than jump straight to the the type `Maybe (Signal Bool)`, we lift the type of `Bool` to `Maybe Bool`. The reason for this intermediate step is to be explicit about what `Maybe` actually represents. By first using `Nothing` to represent always zero on operations that use the innermost type (`Bool`), we separate the optimization from the meaning of zero in this circuit. Using the `abs` and `rep` terminology, we have the following implementation of the coercions.

```haskell
rep1 True = Just True
rep1 False = Just False
abs1 Nothing = False
abs1 (Just True) = True
abs1 (Just False) = False
```

Worker/wrapper can therefore be used to translate box into a worker with the type following the first step in Figure 2.
Commute Maybe and Stream

We now want to commute the Functors Stream and Maybe. There is a significant difference between a stream of optional values, and an optional stream of values. If we try define an abs and rep, these differences are reflected in the preconditions.

\[
\begin{align*}
\text{abs} & \quad : \quad (\text{Stream} \quad (\text{Maybe} \quad a)) \quad \rightarrow \quad \text{Stream} \quad (\text{Maybe} \quad a) \\
\text{abs} & \quad \text{Nothing} \quad = \quad \text{repeat} \quad \text{Nothing} \\
\text{abs} & \quad (\text{Just} \quad x) \quad = \quad \text{fmap} \quad \text{Just} \quad x \\
\text{rep} & \quad : \quad \text{Stream} \quad (\text{Maybe} \quad a) \quad \rightarrow \quad \text{Maybe} \quad (\text{Stream} \quad a) \\
\text{rep} & \quad = \quad ????? \\
\end{align*}
\]

\text{abs} is straightforward to write, but \text{rep} cannot be written; there are too many possible input shapes to represent. However, if we restrict the use of the Stream (Maybe a) space to streams where all the values are Nothing or all the values are just Just, then we can write \text{rep}.

\[
\begin{align*}
\text{-- Pre-condition: Stream in canonical form} \\
\text{rep} & \quad (\text{Cons} \quad \text{Nothing} \quad _) \quad = \quad \text{Nothing} \\
\text{rep} & \quad \text{ss} \quad = \quad \text{Just} \quad (\text{fmap} \quad \text{fromJust} \quad \text{ss}) \\
\end{align*}
\]

The external condition, that the Stream (Maybe a) be one of two specific canonical forms, could be captured formally as a context-sensitive extension of the worker/wrapper transformation. We remain however intentionally informal here; this is exactly the type of refactoring refinement that programmers routinely do without appealing to semi-formal reasoning, and we want to support this while capturing proof obligations that could be filled in later. A reasonable question to ask is if by allowing this informality, we have watered down the methodology too far from its formal roots. We do not believe so, because it is the use of chains of functors that provides the major engineering value, allowing documentation and design exploration that traditional refactoring techniques find challenging. We will, however, revisit this specific point when we have better tool support for these ideas and refinements.

Commuting Stream and Maybe is the step that yields our key optimization. By taking out-of-bounds streams that were always Nothing values and eliminating them altogether, we avoid generating unnecessary hardware. This concept is key to our later LDPC example. More generally, this is an example of a commutative functor pair. Under certain well-defined conditions, specific functors can be safely transposed, and when used with the worker/wrapper transformation, this transposition gives a new program that is one step closer to an implementation.

Replacing List with Tuple8

The next step is to replace the list (which we knew was always 8 elements long) with an 8-tuple. There are a number of pragmatic reasons why we want to generate hardware components with fixed sized inputs. First, this makes reification into hardware possible (Gill 2009). Second, using a fixed shape can encourage sharing of components when generating FPGA bitstreams, potentially speeding up bitstream generation. The abs and rep are trivial.

\[
\begin{align*}
\text{abs} & \quad : \quad \text{Tuple8} \quad a \quad \rightarrow \quad \text{List} \quad a \\
\text{abs} & \quad (a,b,c,d,e,f,g,h) \quad = \quad [a,b,c,d,e,f,g,h] \\
\text{rep} & \quad : \quad \text{List} \quad a \quad \rightarrow \quad \text{Tuple8} \quad a \\
\text{rep} & \quad [a,b,c,d,e,f,g,h] \quad = \quad (a,b,c,d,e,f,g,h) \\
\end{align*}
\]

Here, the pre-condition is that the code that uses box only generates lists with eight elements. This assumption would be easy to verify, once again using a context-aware pre-condition.

Replacing Stream with Signal

The final step in this example is replacing Stream with Signal. In this case, the two are used with a trivial isomorphism, where both are clocked using an implicit clock at the same speed, resulting in a one to one mapping. This mapping is not always this trivial. For example, a stream could be sampled using control logic when being translated into a signal. Given the trivial translation however, we can use coercion functions to and from Signal.

\[
\begin{align*}
\text{abs} & \quad : \quad \text{Signal} \quad a \quad \rightarrow \quad \text{Stream} \quad a \\
\text{abs} & \quad = \quad \text{fromSignal} \\
\text{rep} & \quad : \quad \text{Stream} \quad a \quad \rightarrow \quad \text{Signal} \quad a \\
\text{rep} & \quad = \quad \text{toSignal} \\
\end{align*}
\]

4.2 Discussion

We have (informally) translated an architectural description into an efficient implementation, using functors as our guides. Each step involved demonstrating, in some sense, that the two types represent the same communication within the problem domain.

There are many alternative ways of doing this derivation, even using the same methodology. For example, a more direct approach to optimizing the walls would have been to explicitly build in support for lists up to 8 long, with the walls not contributing elements to the argument list. Another approach would be to just generate a constant \text{False} for the walls. The overarching point is that tradeoffs in the implementation are reflected in the choice of functors and how these functors are translated.

Finally, the box function is not recursive. The worker/wrapper transformation provides a useful framework for semi-formal reasoning, even in this case. We will return to using worker/wrapper on a recursively defined architectural description shortly.

5. Kansas Lava

We target Kansas Lava because it is a subset of Haskell from which we can generate synthesizable VHDL. We will take executable specifications in Haskell, which use Haskell functors, and transform them via worker/wrapper into hardware implementations in Kansas Lava, which use Kansas Lava functors. Targeting Kansas Lava in this way dramatically narrows the gap between Haskell specification and hardware realization.

Kansas Lava is a Haskell-hosted Embedded Domain Specific Language (EDSL) for hardware description and synthesis. Based on the idea of directly specifying circuits as functions, as in the original Lava (Bjesse et al. 1998), Kansas Lava seeks to be a modern and scalable implementation of the Lava paradigm. Kansas Lava is based around two functor-like structures: \text{Comb}, for combinationally generated values, and \text{Seq}, for sequentially generated values. \text{Comb}-centric circuits can be lifted into sequential circuits, using a small set of built-in lift functions:

\[
\begin{align*}
\text{liftS0} & \quad : \quad \text{Comb} \quad a \quad \rightarrow \quad \text{Seq} \quad a \\
\text{liftS1} & \quad : \quad (\text{Comb} \quad a \quad \rightarrow \quad \text{Comb} \quad b) \quad \rightarrow \quad \text{Seq} \quad a \quad \rightarrow \quad \text{Seq} \quad b \\
\text{liftS2} & \quad : \quad (\text{Comb} \quad a \quad \rightarrow \quad \text{Comb} \quad b \quad \rightarrow \quad \text{Comb} \quad c) \quad \\
& \quad \quad \rightarrow \quad \text{Seq} \quad a \quad \rightarrow \quad \text{Seq} \quad b \quad \rightarrow \quad \text{Seq} \quad c \\
\end{align*}
\]

Kansas Lava simultaneously maintains both a shallow and a deep embedding of a circuit, so a Lava circuit can be executed directly (the shallow embedding), or can be reified into VHDL for use on an FPGA (using the deep embedding). The ability to execute a partially refined Lava circuit allows for faster design iteration.
On top of Comb and Seq, Kansas Lava makes use of other structures to express common idioms. Two important structures we use are Enabled and Write.

```haskell
funMap :: (Rep a, Rep b) => (a -> b) -> Seq a -> Seq b
```

Enabled represents an optional value being passed over a wire, typically using an extra boolean signal to express validity. Write represents an optional index-value pair, and is used to express element-wise updates to mutable state, as well as multiplex larger values over time.

Kansas Lava also has a complete implementation of sized types, rendered using associated type functions (Chakravarty et al. 2005), and uses them to implement sized matrices, and sized signed and unsigned bit vectors. Our size types, by notation, start with $U$, $n$ represents the unsigned $n$-bit number.

```haskell
data X256 = ... unsigned number of width x ...
```

We define Matrix x as an instance of Functor. Matrix X8, for example, replaces the use of Tuple8. In addition, we have a number of useful synonyms built on these sized types. One we will be using is $U_3$, an unsigned $n$-bit number.

```haskell
type U3 = Unsigned X3
```

Apart from these idioms, we also support fmap style lifting over Seq (and Comb, using overloading). The key lift function works by enumerating all inputs and recording all outputs, which are stored on a ROM, which is a Seq to Seq function.

```haskell
funMap :: (Rep a, Rep b) => (a -> b) -> Seq a -> Seq b

funMap = (Rep a, Rep b) => (a -> b) -> Seq a -> Seq b
```

Rep is a class that encodes construction and deconstruction of the bitwise representation, allowing enumeration of all inputs.

In addition to the signal types, Kansas Lava makes extensive use of associated type families to permit the flexible handling of groups of signals. This is implemented via a type class and a type function Unpacked. (Both Comb and Seq are instances of Signal.)

```haskell
class (Signal sig) => Pack sig a where
  type Enabled a = Maybe a
  type Write a b = Enabled (a,b)

  pack :: Unpacked sig a -> sig a
  unpack :: sig a -> Unpacked sig a
```

pack and unpack allow us to effortlessly commute between packed and unpacked signals. This commuting groups of signals with pack and unpack turns out to be exactly the same shape as the data transformations we are performing in this paper, and have turned out to be Really Useful in practice.

To give an example of an actual Kansas Lava program, Figure 3 lists the VHDL generated for the top-left hand box of the data transformations we are performing in this paper, and have just introduced. The definition of n encodes the decision that missing streams (corresponding to out-of-bounds neighbors) will be represented by a stream of zeros. We also convert each boolean signal into a signal of three-bit unsigned numbers, and add them together, giving the number of live neighbors. It should be observed that this is a regular foldr over the Matrix content.

We then use fmap to lift the regular Haskell function life over a stream containing the number of live neighbors and this cell’s status. Since the domain of life is limited by the size type, every possible input can be enumerated and a ROM can be generated to implement this function as a lookup.

Figure 4 lists the VHDL generated for the top-left hand box only. There are three inputs (neighbors), and one output, as well as clocking and control signals. We can see, for example, the two-bit unsigned extend in the signal assignment to sig_7_o0.

```vhdl
entity box is
  port(rst : in std_logic; i0 : in std_logic; i1 : in std_logic; i2 : in std_logic; clk_en : in std_logic; clk : in std_logic; clk_en : in std_logic; i0 : in std_logic; i1 : in std_logic; i2 : in std_logic; o0 : out std_logic);
end entity box;
architecture str of box is
begin
  -- Signals
  signal sig_1_o0 : sig_1_o0_memory_type:=
  signal sig_2_o0 : std_logic := '1';
  signal sig_3_o0 : std_logic;
  signal sig_4_o0 : std_logic;
  signal sig_5_o0 : std_logic;
  signal sig_6_o0 : std_logic_vector (15 downto 0);
  signal sig_7_o0 : std_logic;
  signal sig_8_o0 : std_logic;

  -- Matrix
  type sig_4_o0_memory_type is array (15 downto 0)
  of std_logic_vector(0 downto 0);
  signal sig_4_o0_memory_type :=
  ("0","0","0","0","1","1","0","0","0","0","0","0","0","0","0");

  begin
    sig_6_o0 <= std_logic_vector((unsigned(sig_7_o0) + unsigned(sig_8_o0)));
    sig_7_o0 <= "00" & i0;
  end architecture box;
```

```
box_impl :: Bool -> Matrix X8 (Maybe (Seq Bool)) -> Seq Bool

box_impl init ns = res
  where
    n :: Seq U3
    n = foldr (\ a b ->
      case a of
        Just v => unsigned v + b
        Nothing -> b)
      0
    $ M.toList ns
    res :: Seq Bool
    res = register init
      $ funMap life (pack (n,res))

life :: (U3,Bool) -> Maybe Bool

-- stay alive if 2 or 3 neighbors are alive
life (2,True) = return True
life (3,True) = return True

-- spawn if 3 neighbors are alive
life (3,False) = return True

-- otherwise die
life _ = return False
```
6. LDPC

A Low Density Parity Check (LDPC) code is a Forward Error Correcting (FEC) code that uses parity bits to support the recovery of data transmitted over a noisy link (Moon 2005). The LDPC algorithm uses Belief Propagation to attempt to find the original message inside a received signal, and can be constructed in a way that is arbitrarily close to the channel capacity, or the Shannon Limit. Though invented in the 60’s by Gallager (1962), it remained until the mid-90’s a theoretical curiosity due to the computational cost of any implementation. Advances in LDPC code construction have resulted in codes that can be decoded in real time (Andrews et al. 2007). Our lab was tasked with implementing one such code, on an FPGA fabric, and investigating the properties of the final implementation.

The decode specification is concise, and can be given by three equations. Using the notation from the standard reference on error correcting codes (Moon 2005), and starting with the matrix $\eta$, we have:

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

$$\eta_{m,n} = -2 \tanh^{-1} \left( \prod_{j \in A_{m,n}} \tanh \left( \frac{1}{2} (\eta_{m,j}^{[l-1]} - \lambda_j^{[l-1]}) \right) \right)$$

For each $n$:

$$\lambda_n = \lambda_n^{[0]} + \sum_{m \in A_{n,m}} \eta_{m,n}^{[l]}$$

For each $n$:

$$c_{n}^{[l]} = 1, \text{if} \lambda_n^{[l]} > 0, \text{otherwise} = 0.$$  

For each $m$, $c_n^{[l]}$ is computed, starting from iteration 1. This algorithm terminates with the result $c$ if parity is achieved ($A(c) = 0$), or alternatively an iteration threshold is reached, in which case, then $c_{n}^{[l]}$ is returned. The input, $\lambda^{[0]}$ is a soft input, being a representation of likelihood of a symbol, with zero representing unknown. Finally, $A_{n,m}$ represents $\{ m : A_{m,n} = 1 \}$, and $A_{m,n}$ represents $\{ n : A_{m,n} = 1 \}$.

A is a sparse binary matrix that forms a parity checking property for messages that have parity added using the encoding matrix. The details of how to invent these two matrices, jointly called a code, are not important here; in our case we were required to use a specific code invented by JPL (Consultative Committee for Space Data Systems (CCSDS) 2007). The JPL code we are using is a 4K 2/3-rate code, which takes 4096 bits, and transmits 2048 additional parity bits. The decoder actually operates on 7K bits, the last 1K being extra untransmitted parity bits.

Each row of the sparse matrix $A$ represents an individual parity check between a small number of bits. When multiplying $A$ with a symbolized received packet (normalized to the binary symbols 0 and 1), if every value in the result is zero, then the received packet is, with extremely high probability, without error. The challenge of LDPC is finding a variant of the received noisy signal that achieves parity. The algorithm works by having each (non-zero) element of $\eta$ contain a small delta that attempts to allow its specific row to reach parity.

Performing tanh and $\text{tanh}^{-1}$ is pervasive in the algorithm, but expensive in hardware, so the following compromise is often taken:

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

$$\eta_{m,n} = -\frac{3}{4} \text{fold min} \left[ \eta_{m,j}^{[l]} - \lambda_j^{[l]} \right]_{j \in A_{m,n}}$$

$$\text{min}(x, y) = \text{sign}(x) \cdot \text{sign}(y) + \text{min}(|x|, |y|)$$

The intuition behind this modification of the algorithm is that the Belief Propagation was dominated by the smallest value, because of the use of the product of the log likelihoods. This compromise results in a slight loss of decoding quality, around 0.2dB, in return for an algorithm that is much easier to map onto FPGAs. In the FPGA literature (Andrews et al. 2007) this function is a known simplified variant of the min function. From an algebraic point of view, we observe that $\text{min}_1$ is associative and commutative, so any fold (left, right or balanced) can be used. We also observe that $\text{min}_1$ is also bit-accurately associative and commutative, even when used on quantized fixed-precision values.

It is straightforward to transliterate this mathematical description into a Haskell-based executable specification, as given in Figure 6. This is actually an optimized Haskell program, using three different representations of matrices, that ran fast enough to use on real test vectors. (In this paper, as much as possible, we present original code, but for presentation purposes and due to space considerations we have performed some minor $\alpha$-conversions, and other small changes, like omitting the debugging parameterization. None of the changes affect the thesis of this paper.)

```
loop :: forall x y a
  . (Fractional a, Ord a, Size x, Size y)
  => Options
  => Int
  => A y x
  => SM.Matrix (y,x) a
  => Matrix x a
  => Matrix x a
  => IO (Maybe (Matrix x U1,Int))
loop options n a@(A a_rref aRows aCols) ne lam orig_lam
  | cardinality ans == 0 = return $ Just (c_hat,n)
  | n > iterations options = return Nothing
  | otherwise = loop options (n+1) a ne' lam' orig_lam
  where
    c_hat :: Matrix x U1
    c_hat = fmap \(c -> if c > 0 then 1 else 0) lam
    ans :: BitMatrix (y, X1)
    ans = a_rref 'mult' fromMatrix (unitColumn c_hat)
    ne' :: SM.Matrix (y,x) a
    ne' = SM.fromAssocList 0
    \((m,n), -0.75 + (foldr1 metric
    [ (ne SM.! (m,j)) - (lam ! j) ]
    \( j <- toList (aRows ! m), j /= n )
    ))
    \( (m,n) <- toList (toList a_rows a_ref )

    lam' :: Matrix x a
    lam' = forall $ \n -> (orig_lam \ n)
              + sum [ ne SM.! (m,n) \ n <- toList (aCols ! n) ]
```

Figure 6. Haskell Description of LDPC
7. From Specification to Architecture

We start our implementation with a requirement to handle a 4 Mbps stream of data, transmitted over a 70MHz carrier frequency. This is inside the performance range of existing implementations on Virtex-4 technologies (Andrews et al. 2007), but still requires careful performance engineering. The first stage of our derivation is to choose our architecture. That is, a Haskell program that represents the way we choose to lay out computation on a hardware fabric, without being specific about details like clocking or control. Most existing hardware implementations of LDPC use a crossbar architecture to communicate between different segments of existing hardware implementations of LDPC, without being specific about details like clocking or control.

Most existing hardware implementations of LDPC use a crossbar architecture to communicate between different segments of Λ, via intermediate nodes that represent individual rows of η. Values are passed from Λ nodes to η nodes, and back, through this crossbar, and this communication dominates the cost of computing each iteration. Much of the recent research into LDPC implementation and this communication dominates the cost of computing each iteration. Given this, we can rework our row-based computations.

Figure 7 shows the conceptual breakup into “cells” of computation. Dividing the fabric horizontally is trivial. The same Λ is passed in both the top and bottom, and the Λ deltas can be computed for each column using a simple addition. The parity is delivered as two separate streams of data, transmitted over a 70MHz carrier frequency. This is inside the performance range of existing implementations on Virtex-4 technologies (Andrews et al. 2007), but still requires careful performance engineering. The first stage of our derivation is to choose our architecture. That is, a Haskell program that represents the way we choose to lay out computation on a hardware fabric, without being specific about details like clocking or control. Most existing hardware implementations of LDPC use a crossbar architecture to communicate between different segments of Λ, via intermediate nodes that represent individual rows of η. Values are passed from Λ nodes to η nodes, and back, through this crossbar, and this communication dominates the cost of computing each iteration. Much of the recent research into LDPC implementation and this communication dominates the cost of computing each iteration. Given this, we can rework our row-based computations.

Figure 7 shows the conceptual breakup into “cells” of computation. Dividing the fabric horizontally is trivial. The same Λ is passed in both the top and bottom, and the Λ deltas can be computed for each column using a simple addition. The parity is delivered as two separate streams of data, transmitted over a 70MHz carrier frequency. This is inside the performance range of existing implementations on Virtex-4 technologies (Andrews et al. 2007), but still requires careful performance engineering. The first stage of our derivation is to choose our architecture. That is, a Haskell program that represents the way we choose to lay out computation on a hardware fabric, without being specific about details like clocking or control. Most existing hardware implementations of LDPC use a crossbar architecture to communicate between different segments of Λ, via intermediate nodes that represent individual rows of η. Values are passed from Λ nodes to η nodes, and back, through this crossbar, and this communication dominates the cost of computing each iteration. Much of the recent research into LDPC implementation and this communication dominates the cost of computing each iteration. Given this, we can rework our row-based computations.

Dividing the fabric vertically, as shown in Figure 7(c), is more challenging. This is because, assuming an identity element for min† to which unlocks our architectural choice. This is because, assuming an identity element for min†, each non-zero element in η being computed using

\[ \text{fold} \min^† \theta \left[ (\eta_{m,j}^{l-1} - \lambda_j^{l-1}) \right] \mid j \in \mathbb{N}_{m,n} \]

that is, every η value of a row depends on every other (non-zero) η value in the same row. This means that splitting the fabric vertically unfortunately results in a communication overhead that is no longer proportional to the perimeter of the fabric and instead depends on the number of contributing values in each row of η.

We can use the associativity and commutativity of min† to minimize communication costs. Specially, it is straightforward to demonstrate that, for any associative and commutative f that has an identity element z, and any total predicate p, the following holds.

\[ \forall x. \text{fold} f \in x z \equiv f(\text{fold} f \in x \in x, p x) \]

Given this, we can rework our row-based computations.

For each row m, and partition predicate p, where p n is true:

\[ \text{fold} \min^† \theta \left[ (\eta_{m,j}^{l-1} - \lambda_j^{l-1}) \right] \mid j \in \mathbb{N}_{m,n} \]

{Expand out explicitly the \( \neq n, \mathbb{N}_m \) represents \{n : A_{m,n} = 1\} }

\[ \equiv \text{fold} \min^† \theta \left[ (\eta_{m,j}^{l-1} - \lambda_j^{l-1}) \right] \mid j \in \mathbb{N}_{m,n}, j \neq n \]

{For \min†, we can factor our \text{fold} into two \text{folds} }

\[ \equiv \min^†(\text{fold} \min^† \theta \left[ (\eta_{m,j}^{l-1} - \lambda_j^{l-1}) \right] \mid j \in \mathbb{N}_{m,n}, j \neq n, p j) \]

\[ \equiv \min^†(\text{fold} \min^† \theta \left[ (\eta_{m,j}^{l-1} - \lambda_j^{l-1}) \right] \mid j \in \mathbb{N}_{m,n}, j \neq n, \neg p j) \]

\[ \equiv \min^†(\text{fold} \min^† \theta \left[ (\eta_{m,j}^{l-1} - \lambda_j^{l-1}) \right] \mid j \in \mathbb{N}_{m,n}, j \neq n, p j) \]

This derivation means that we can combine all the values outside of our (arbitrary) partitioning once, and use this external result as part of the answer for each sub-computation, inside the partition. This trick of splitting folded values is used at the (arbitrary) vertical boundary of our computation fabric. Figure 7(d) illustrates this formulation. The left and right sides both use each others’ row summary as well as any external row summary, and the resultant row summary for the larger box is the \min† of the two row summaries.

Armed with the ability to split the computation fabric in two different directions, we need to choose a suitable sized “cell”. Reducing to single 1x1 cells results in optimal parallelism, but unmanageable communication, and would not fit on any current FPGA. When we examine the JPL LDPC decode matrix \( A \), we find a by-construction curiosity that unlocks our architectural choice. Specifically, the decode matrix is constructed entirely of fixed-sized...
cells of either an empty matrix, an identity matrix, or of a cyclically shifted identity matrix. This property holds by design for the JPL codes, and in the case of the 4K code, we have a matrix of 28 columns by 12 rows, containing cells of 256x256.

Assuming we render each of these 256x256 cells as individual hardware components in our architecture, we can (1) invent the type for our cells, (2) define a mechanism for routing data between cells, and (3) refine our specification into a Haskell program that implements (1) and (2). The aspect of these cells that draws us to this design is that each cell contains only 256 individual intermediate \( \eta \) values, but implements a sub-matrix of the \( \eta \) matrix sized 256x256, a wonderful property!

We can now target a specific abstracted architecture that reflects our choices. Figure 8 gives the type and pseudo-code for our main architecture function. Rather than write this by hand, we used by-hand refinement from the specification. We set up adjacent Haskell modules with the same API, and imported them qualified, allowing the main driver to pick and choose of set of possible implementations to execute using tests vectors, and compare results. In this way, each new refinement step could be tested against its previous step. The steps taken were as follows.

- A purely functional `solve` function is extracted from the `loop` function, which explicitly takes and returns the two pieces of state (the \( \eta \) and \( \lambda \)).

```haskell
solve :: forall x y a . (Size x, Size y, Fractional a, Ord a)
  => A y x
  -> (SM.Matrix (y,x) a, Matrix x a)
  -> (SM.Matrix (y,x) a, Matrix x a)
SM . Matrix is our sparse representation of a matrix.
```

- From this solve function, a first version of `operate` is extracted, which takes and returns unsized Haskell arrays.

```haskell
operate0 :: (Ord b, Fractional b, Num b, x ~ Int, y ~ Int)
  => Array (x,y) U1
  -- A
  -> Array y b
  -- lambda
  -> Array x (Maybe b)
  -- Global partial
  -> ( Array (x,y) b
        , Array x (Maybe b)
        , Array y b )
  -- new eta
  -- new lambda
  -> ( SM.Matrix (y,x) a, Matrix x a )
```

- The `operate` function is factored to act using `Stream`, rather than take and return state.

```haskell
operate1 :: (Ord b, Fractional b, Num b, x ~ Int, y ~ Int)
  => Array (x,y) U1
  -- A
  -> Stream (Array y b)
  -- lambda
  -> Stream (Array x (Maybe b))
  -- Global partial
  -> Stream (Array x (Maybe b))
  -- Local partial
  -- operate1 is extracted, as well as spotting zero matrices. This makes no difference to the type, but relies on the splitting function actually implementing the LDPC by correctly chaining together vertical and horizontal components.

- We finally added the ability to represent the optimized zeros, giving the type in Figure 8. This final step was actually performed using a by-hand execution of the worker/wrapper transformation.

8. From Architecture to Implementation

We now have our working executable architecture model of LDPC, expressed in terms of arguments with chains of functors. We now apply the same approach as in Section 4.1 to get from our architecture to our implementation. Of course, this real implementation example is more involved. The steps we took to translation from architecture to (efficient) implementation were as follows.

- We switched to using a generic representation of the large sparse array \( A \) using worker/wrapper. This was more an artifact of the way we were using the model, rather than strictly necessary.

- Next, we use worker/wrapper to change `Stream` of `Array` into `List` of `Stream` of (fixed-size) `Matrix`. This step also committed to the fixed cell size \( S \).

```haskell
operate3 :: forall b x y
  . (Ord b, Fractional b, Num b, Real b, x ~ Int, y ~ Int)
  => A (x,y)
  -- A
  -> List (Stream (Matrix S2 b))
  -> List (Stream (Matrix S2 (Maybe b)))
  -> List (Stream (Matrix S2 b))
  -> List (Stream (Matrix S2 (Maybe b)))
```

- We then commuted the `Maybe` functor outwards, giving:

```haskell
operate4 :: forall b x y
  . (Ord b, Fractional b, Num b, Real b, x ~ Int, y ~ Int)
  => A (x,y)
  -> List (Stream (Matrix S2 b))
  -> List (Stream (Matrix S2 (Maybe b)))
  -> List (Stream (Matrix S2 b))
  -> List (Stream (Matrix S2 (Maybe b)))
```

This is an extension of the transformation performed when commuting `Maybe` and `Stream`, in § 4.1.
9. Onwards to Hardware

The final step is taking our working and cycle-accurate VHDL generated by Kansas Lava, and executing it on our hardware. Techniques for programming an FPGA board are well documented, and there are many online examples and other resources to draw from. Our first place-and-routed version of LDPC came back with unreasonable path delays. The circuit clocked at around 14Mhz, which is unfortunately too slow to run on our board, which requires at least 40MHz.

In Kansas Lava, the internal graph structure of all nodes is a first class data-structure. It was straightforward to add an option for annotating code with path length comments, making finding critical paths at the Kansas Lava-level straightforward. The depth (distance from a register output, excluding trivial wiring) also appeared as comments in our generated VHDL. The ability to start within the high level description language yet also achieve instant cycle accurate simulation allows for quick exploration of designs.

Using the depth data from these comments, we again refine the type of operate. One place there was a depth issue was the tree of min and, caused by the recursive code-generating solution. The obvious solution was to place a register after every min, but there is no guarantee in this design that all sub-trees of min have the same depth. This would result in the stream of values becoming out of sync, and the wrong values being operated on.

Our generative code approach offers another alternative. The type we are communicating with internally is:

\[
\text{... (Maybe (Seq (Write SZ FLOAT))) \ldots}
\]

taken from the second-to-last step in Section 8. We refine this (using worker/wrapper) into:

\[
\text{... (Maybe (Delay (Seq (Write SZ FLOAT)))) \ldots}
\]

\[
\text{-- definition of Delay type Delay a = (Int,a)}
\]

The \text{Int} in \text{Delay} represents the number of cycles the index-value \text{Write} datum has been statically delayed, using the \text{Delay} functor definition.

We then lifted \text{min} function, over the delay.

\[
\text{liftDelay :: (Rep a, Rep b, Rep c)}
\]

\[
\Rightarrow (Comb a \rightarrow Comb b \rightarrow Comb c)
\]

\[
\Rightarrow \text{Delay (Seq a)}
\]

\[
\Rightarrow \text{Delay (Seq b)}
\]

\[
\Rightarrow \text{Delay (Seq c)}
\]

\[
\text{liftDelay f (n,a) (m,b) } = (\text{max n m + 1, liftS2 f a' b'} )
\]

where

\[
a' = \text{delays (max n m - n)} a \\
b' = \text{delays (max n m - m)} b
\]

This lifting, and an extra pause of a few cycles added to the state “Post Share” in Figure 6 allowed for the min function to be automatically retimed with the correctly aligned data arriving for merging in the same cycle. This automated the addition of a set of strategic registers between components. After making the necessary changes to the control logic to accommodate this, we reached a more respectable clock rate of 92.5Mhz and more scope for optimization remains. Table 1 gives a summary of the current device utilization, including the communication driver, for the 8-bit quantized version of LDPC.

<table>
<thead>
<tr>
<th>Device Utilization on the XC5VLX110T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of RAMB18X2s</td>
</tr>
<tr>
<td>Number of Slice Registers</td>
</tr>
<tr>
<td>Number of Slice LUTS</td>
</tr>
<tr>
<td>Number of Slice LUT-Flip Flop pairs</td>
</tr>
</tbody>
</table>
10. Results on FPGA

Figure 11 shows the result of running the 8-bit quantized LDPC circuit, on the FPGA, performing forward error correction.

- BPSK is the trivial decode with no parity checking, and is our baseline.
- LDPC is our forward error correction decoder in action.

LDPC requires extra parity bits to be transmitted, and therefore the LDPC results can be worse than BPSK, when low signal to noise causes LDPC to fail frequently. However, under normal operating conditions, the LDPC is known to allow for around 8dB of forward error correction, allowing for lower power transmitters and/or operation in noisier environments.

The next steps in the project include:

- Testing our circuit with real noise over an air gap, rather than additive white Gaussian noise. This will involve connecting the LDPC to our existing implementation of a demodulator, and building a transmitter. Both are project requirements.
- Accurately verifying the maximum possible rate from this circuit.
- Experimenting with new control logic to transfer multiple FLOAT values per cycle, increasing the throughput further. We think that the use of functors for refinement will make this possible to explore.
- By using the min, we took a small compromise over an implementation based on tan, and we would like to attempt to build a tanh version of the LDPC for FPGAs.

How competitive is our implementation of LDPC? Our current implementation is well within data rate for the original specification (4 Mbps). Figure 11 shows LDPC running a maximum of 200 iterations of Belief Propagation on each 6K frame. Under normal conditions an average of 25 iterations are required. Given our current timing of approximately 800 cycles per iteration, we can theoretically handle a data rate around 16 Mbps. This appears comparable with other implementations (Andrews et al., 2007), though accurate comparisons are challenging, because published results for LDPC are often given in terms of best case, rather than actual throughput speed. As already mentioned, we have not by any means exhausted our optimization opportunities, and we have a realtime decoder. Of course, if we want a faster circuit, we chain up our functors, and start refining again.

11. Related Work

Functional programming has a long and rich history of program refinement. It is taught in undergraduate textbooks (Hutton 2007), and some members of the functional programming community have made this into an art form (Bird 2010). What was attempted in this paper was of a slightly different nature; the refinement was a guide for a semi-formal method, but as when refactoring programs in the wild, we were prepared to take leaps of faith with correctness. The difference here is that, as a consequence of the use of worker/wrapper in many places, we have an audit trail of (currently unsatisfied) proof obligations.

The worker/wrapper transformation was initially used to exploit strictness information to unbox integer computations in the Glasgow Haskell Compiler (Jones and Partain 1993) and was more recently formalized (Gill and Hutton 2009). We believe there is tremendous untapped potential for systematic refinement, and the work described in this paper is a prelude to future refinement and automation; we now know where the shoe pinches!

We make critical use of the ability to generate VHDL provided by Kansas Lava, and the Xilinx XST toolkit to generate bit files for FPGA reprogramming. Gill et al. (2011) presents the FPGA-specific details of deploying our generated LDPC implementation, including board-specific challenges and data transportation issues. Given the common LDPC artifact, there is overlap between Gill et al. (2011) and this paper. Specifically, the generic description of the LDPC algorithm in Section 6 and the high-level description of our fabric partitioning aspect at the start of Section 7 were both adapted from the descriptions in the earlier paper. The earlier paper alluded to but is not specific about the existence of a refinement methodology (Section 5, final paragraph). Apart from this reference, the earlier published work assumes that we implement the LDPC algorithm in Kansas Lava directly to facilitate generation of the VHDL LDPC; this paper discusses in detail how we instead reached our implementation using refinement techniques.

The original ideas for Lava (Bjesse et al. 1998) can be traced back to the Ruby hardware description language (Jones and Sheeran 1990) and prior to that, µFP (Sheeran 1984). Kansas Lava, like other Lavas, is a modeling language. Kansas Lava models communicating processes, via synchronous or hand-shaken signals. There are several other modeling languages that share a similar basic computational basis, for example Esterel (Berry 1999). Finally, 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. (Jantsch and Sander 2005) gives a good summary.

12. Conclusion

We have developed a toolkit called Kansas Lava for expressing FPGA-level concerns, following in the tradition of previous Lava implementations while also leveraging modern features like associated type families. On top of this toolkit, we have a function-based semi-formal methodology for deriving circuits from executable specifications. We have completed one full-scale derivation by hand, and are preparing to undertake more derivations and automate some of the mundane details.

While Kansas Lava is a subset of Haskell, there are important restrictions on the types available, as they must be representable in hardware if we are to realize a VHDL or FPGA-based implementation. As such, there is a gap between the executable specification in Haskell and the version that operates only on the available Kansas Lava types. We extensively use the worker/wrapper meta-transformation to bridge this gap. With worker/wrapper, we systematically transform the type of our computation to achieve a viable implementation from an executable specification.
Functors play a crucial role in this implementation derivation. The type of our architectural description is given as a chain of functors, which we transform by adding, commuting, or replacing functors in the chain until the type is fully realizable in hardware. The fact that Kansas Lava signal types are functors within the restrictions of the preconditions makes this transformation possible.

By expressing intermediate computations using chains of functors, and performing refinement transformations by hand, we have developed an efficient LDPC decoder than runs in real time. This implementation of LDPC meets all the project’s performance, correctness, and resource utilization goals.

The major weakness of commutatable functors and worker/wrapper for deriving hardware circuits, as used in this paper, is that everything is done by hand. This leads to proof obligations being notes in a derivation chain, not formally captured, and means that architecture exploration is challenging, especially if an earlier stage needs revisited. At one point in the derivation, we backtracked to change a representation, resulting in changing several stages of implementations of LDPC, not just one, which was burdensome.

The next steps are two-fold. The derivation was a free-for-all, with approximate directions being provided by the architecture, and the human deliver figuring out how to twist the functors to hit the target. We want to develop a more rigorous methodology, based on functors and using worker/wrapper, that provides more guidance. Simultaneously, we want to start automating many of the transformations we are using.

Acknowledgments

The authors would like to thank the Test Resource Management Center (TRMCT) Program and Evaluation/Science and Technology (T&E/S&T) Program for their support. This work was funded in part by the T&E/S&T Program through the Army PEO STRICCT Contracting Office, contract number W900KK-09-C-0018 for High-Rate High-Speed Forward Error Correction Architectures for Aeronautical Telemetry (HFEC). The authors would also like to thank Xilinx University Program for their donation of FPGA equipment. Finally, the authors would like to thank the members of the CSDL lab at ITTC, and the EECS 800 Advanced Functional Programming class, for many fruitful discussions about using functional languages for hardware description and other applications.

References


