Behavioural Circuit Design in Lava

by Matthew Naylor ⟨mfn@cs.york.ac.uk⟩

*The Haskell library **Lava** is great for describing the **structure** of digital circuits: large, regular circuits can be captured by short, clear descriptions. However, structural circuit description alone – and hence Lava – has a somewhat limited application domain. Many circuits are more appropriately expressed in terms of their desired **behaviour**.*

*In this article, I present a Haskell library – called **Recipe** – that sits on top of Lava and provides a set of behavioural programming constructs, including mutable variables, sequential and parallel composition, iteration and choice. Mutable state is provided in the form of rewrite rules – normal Lava functions from state to state – giving the library a powerful blend of the structural and behavioural styles.*

The approach taken here is also applicable to software-based embedded systems programming. Indeed, I have developed a simple C backend for Lava, and my final example – a program that controls a brick-sorter robot – runs on a Lego Mindstorms RCX microcontoller.

Overview

This article is structured in three parts:

- An introduction to structural circuit description in Lava.
- Implementation and discussion of the Recipe library.
- Example applications of the Recipe library, including a sequential multiplier and a controller for a Lego brick-sorter.

Structural Circuit Description in Lava

Lava is a library for hardware design, developed at Chalmers University [1]. In essence, it provides an abstract data type `Bit`, along with a number of common operations over bits. To illustrate, the following Lava function takes a pair of bits and sorts them into ascending order.

```
bitSort         :: (Bit, Bit) -> (Bit, Bit)
bitSort (a, b) =  (a ==> b) ? ((a, b), (b, a))
```

Here, two Lava functions are called: implication (`==>`), which is the standard ordering relation on bits, and choice (`?`), which in hardware terms is a multiplexor, and in software terms is a C-style if-expression.

Lava allows any function over some (nested) structure of bits, be it a tuple, a list, a tree, or whatever, to be:

- simulated by applying it to sample inputs,
- turned into logical formula that can be verified for all inputs of a given, fixed size by a model checker, and
- turned into a VHDL netlist that can be used to configure an FPGA.

For example, to simulate our bit sorter, we can just say:

```
Lava> simulate bitSort (high, low)
(low, high)
```

Looks correct, but to be sure, we can formulate a correctness property:

```
prop_bitSort (a, b) = c ==> d
  where
    (c, d)          = bitSort (a, b)
```

and verify it **for all inputs**:

```
Lava> smv prop_bitSort
Valid
```

In this case we have requested to use the SMV verifier [2]. Now that we are sure our circuit is correct, we can turn it into a VHDL netlist, ready for circuit synthesis:

```
Lava> writeVhdlInputOutput "BitSort" bitSort
        (var "a", var "b") (var "c", var "d")
Writing to file "BitSort.vhd" ... Done.
```

This command generates the file `BitSort.vhd`, and requests that in the VHDL code, the inputs be named `"a"` and `"b"`, and outputs `"c"` and `"d"`.

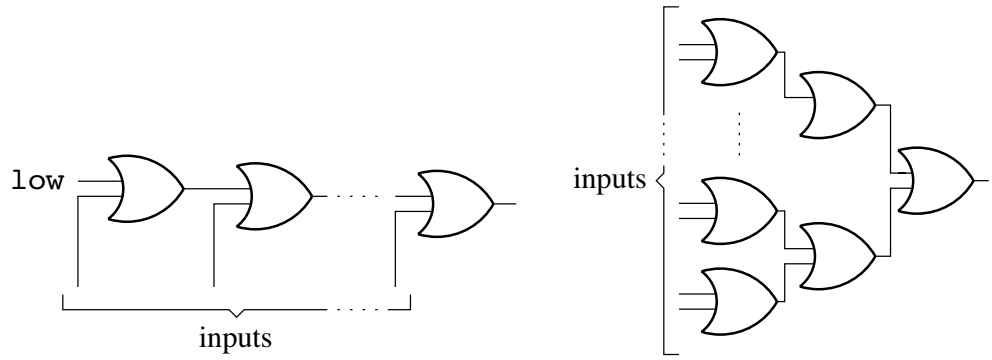

Figure 1: Linear reduction (left) and tree reduction (right).

Larger Circuits

The `bitSort` circuit is of fixed size: it always takes two inputs and produces two outputs. However, in Lava, circuit descriptions can be parameterised by the numbers of inputs and outputs. For example, logical disjunction over an arbitrary number of bits can be expressed as:

```
orl        :: [Bit] -> Bit
orl []     =  low
orl (a:as) =  a <|> orl as
```

As physical circuits have a fixed size, the exact number of inputs to a Lava description must be decided when calling a function like `writeVhdl`. This is because Lava, in effect, expands out the recursion in a description. For example, the `orl` description corresponds to the circuit structure shown on the left in Figure 1.

It is clear from Figure 1 that the circuit structure of `orl` has a linear shape: the time taken to compute the output is linear in the number of inputs. With parallelism, and the fact that disjunction is commutative and associative, we can do much better using a tree-shaped structure:

```
tree            :: (a -> a -> a) -> [a] -> a
tree f [a]      =  [a]
tree f (a:b:bs) =  tree f (bs ++ [f a b])
```

The structure of the description `tree (<|>)` is shown on the right in Figure 1. To be sure that it computes the same function as `orl`, we can verify the following correctness property:

```
prop_OrTree = forall (list 8) $ \as ->
                orl as <==> tree (<|>) as
```

Notice that we have only stated that the equivalance holds for size 8 input lists. In fact, this property does not hold in general, since `tree` is undefined on the empty list.

Sequential Circuits

Sequential circuits are circuits that contain clocked memory elements and feedback loops. They can be described in Lava using `delay` elements and value recursion. The `delay` primitive takes a default value and an input. It outputs the default value on the initial clock cycle, and from then on, the value of the input from the previous clock cycle. For example:

```
Lava> simulateSeq (delay low) [high, high, high, low, low]
[low,high,high,high,low]
```

The `delay` element can be thought of as having an internal state, and can therefore remember its input value from the previous clock cycle.

Value recursion, also known as "circular programming", is used to express feedback loops. For example, the following description implements a delay element with an input-enable line. The internal state is only updated when the enable line is active.

```
delayEn init (en, inp) = out
  where
    out                = delay init (en ? (inp, out))
```

The structure of `delayEn` is shown on the left in Figure 2. This function will prove useful later, in the implementation of Recipe.

Another function that will prove useful later is the set-reset latch. It takes two inputs, set and reset. A pulse on the set or reset line causes the internal state of the delay element to go high or low respectively. For our purposes, it is unimportant what happens when the set and reset lines are pulsed at the same time.

```
setReset (s, r) = out
  where
    q           = delay low (and2 (out, inv r))
    out         = or2 (s, q)
```

The structure of `setReset` is shown on the right in Figure 2.

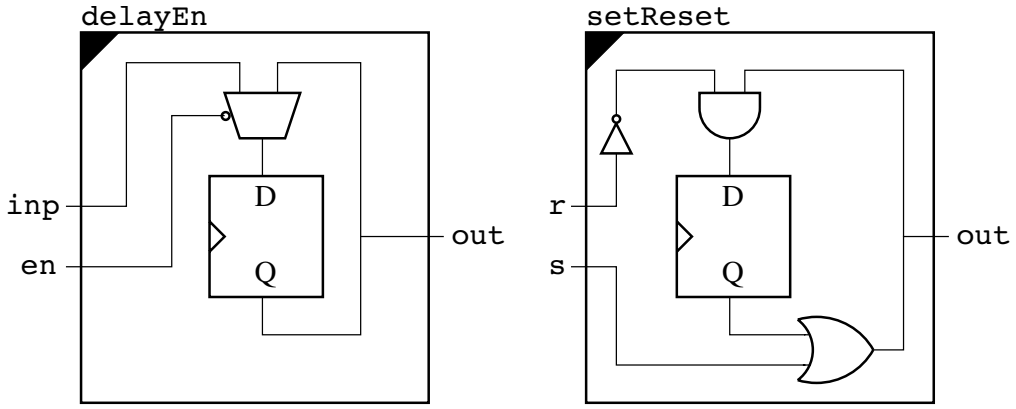

Figure 2: Delay with input enable (left) and a set-reset latch (right).

Recipe – A Library for Behavioural Description

The structural approach is ideal for many kinds of circuit – see the Lava tutorial [1] and Mary Sheeran's web page [3] for more examples. However, when it comes to circuits with multiple feedback loops, structural descriptions can become rather awkward to write and hard to understand. Even the simple feedback loops, like those shown in Figure 2, can take a while to understand.

What we need is a more abstract notion of state and control-flow. One such abstraction, which will be used in Recipe, is Page and Luk's hardware variant of Occam [4]. It provides basic program statements such as skip and assignment, which take one clock cycle to complete, and powerful combining forms including sequential and parallel composition, choice and iteration.

But how can such a behavioural language be integrated with Lava? Koen Claessen and Gordon Pace give us a big hint: they roughly say "just describe the language syntax as a Haskell data type, and write an interpreter for that language in Lava". This approach, which may seem obvious, turns out to be very neat. Indeed, Claessen and Pace have already developed behavioural langauges on top of Lava, but none are quite as general as the one which I present in here.

I will take a slightly different approach to Claessen and Pace here. For a more direct presentation, Recipe will be implemented directly as a set of monadic functions as opposed to a single evaluation function over an abstract syntax. In practice however, an abstract syntax is preferred because optimisations can be applied.

The Recipe Monad

In Page and Luk's Occam variant, each program construct can be thought of as a black box which takes a **start** signal and produces a **finish** signal. So, my Recipe monad is a function that takes a start bit and returns a finish bit:

```
data Recipe a = Recipe { run :: Bit -> Env -> (Bit, Env, a) }
```

You will notice that it also takes an environment (of type `Env`) and produces an environment. For now, the purpose of the environment is unimportant – it will be explained later. The `Recipe` data type can be made a monad as follows:

```
instance Monad Recipe where
  return a = Recipe $ \start env -> (start, env, a)
  m >>= f  = Recipe $ \start env ->
               let (fin0, env0, a) = run m start env
                   (fin1, env1, b) = run (f a) fin0 env0
               in  (fin1, env1, b)
```

This is really no more than a state monad, but for those who are less familiar with monads, here are the important points:

- `return` doesn't read or write any environment information. It just connects the start signal directly to the finish signal.
- `>>=` sequentially composes two recipes. It passes the start signal to the first recipe, which returns a finish signal. This finish signal is then passed as the start signal to the second recipe.
- `>>=` also threads the environment through the two recipes.

This completes the definitions of our first two behavioural constructs – the circuit which does nothing, and sequential composition!

Skip and Wait

The `return` construct is great for doing nothing, but why stop there when we can do nothing **and** take one clock cycle to do it! This is what `skip` does:

```
skip :: Recipe ()
skip =  Recipe $ \start env -> (delay low start, env, ())
```

The finish signal simply becomes the start signal delayed by one clock cycle. Note that the idea of these start and finish signals is that they carry single-cycle pulses to indicate the start or finish event.

The great thing about making Recipe a combinator library is that recipes are just Haskell values. New ones can be defined in terms of existing ones:

```
wait   :: Int -> Recipe ()
wait 0 =  return ()
wait n =  skip >> wait (n-1)
```

Now we can do nothing and use up an arbitrary number of clock cycles to do it. Despite my sarcasm, `skip` and `wait` are actually rather useful, and important.

Parallel Composition

Recipes can be composed in parallel using the `|||` operator, defined as follows:

```
(|||)   :: Recipe a -> Recipe b -> Recipe (a, b)
p ||| q =  Recipe $ \start env ->
             let (fin0, env0, a) = run p start env
                 (fin1, env1, b) = run q start env0
                 fin             = setReset (fin0, fin)
                                       <&> setReset (fin1, fin)
             in  (fin, env1, (a, b))
```

Parallel composition behaves according to a fork/join semantics. This means that an expression of the form `p ||| q` starts `p` and `q` at the same time, and finishes only when **both** `p` and `q` have finished. To implement this blocking behaviour, we feed each of the finish signals of `p` and `q` into the set-line of a set-reset latch, and combine the outputs with an and-gate. This final finish signal is then fed back into the reset-line of each latch, so that the control circuitry is left in a reuseable state, ready for the next time it is executed (for example, if it occurs in the body of a loop).

Choice

An if-then-else construct over recipes is defined as:

```
cond       :: Bit -> Recipe a -> Recipe b -> Recipe ()
cond c p q =  Recipe $ \start env ->
                let (fin0, env0, _) = run p (start <&> c) env
                    (fin1, env1, _) = run q (start <&> inv c) env0
                in  (fin0 <|> fin1, env1, ())
```

It takes a condition bit `c`, and two recipes, `p` (the then-branch) and `q` (the else-branch). Notice that `c` is of type `Bit`, meaning that any normal Lava function can be used to describe the condition. The conjunction of the start signal and `c` is used to trigger `p`, and the conjunction of the start signal and the inverse of `c` is used to trigger `q`. The the final finish signal is equal to the disjunction of the finish signals of `p` and `q`.

Iteration

A construct that repeatedly runs a recipe while a condition holds is defined as:

```
iter     :: Bit -> Recipe a -> Recipe a
iter c p =  Recipe $ \start env ->
              let (fin, env', b) = run p (c <&> ready) env
                  ready          = start <|> fin
              in  (inv c <&> ready, env', b)
```

The loop body `p` is said to be "ready" when the start signal is active, or its own finish signal is active. However, it is only triggered when it is both ready and the loop-condition holds. If it is ready, and the loop-condition does not hold, then the overall finish signal is triggered.

When using loops, the programmer must take care not to make a loop-body that takes no clock-cycles to complete – this would result in a combinatorial loop in the circuit! If a loop body can take zero clock cycles under certain conditions, then a `skip` should be inserted for safety. This is why I earlier claimed that `skip` is important.

Again, we can define other useful combinators on top of existing ones:

```
forever     :: Recipe a -> Recipe a
forever p   =  iter high p

waitWhile   :: Bit -> Recipe ()
waitWhile a =  iter a skip

waitUntil   :: Bit -> Recipe ()
waitUntil a =  iter (inv a) skip
```

The Environment

We have progressed reasonably far without needing an environment, i.e. any global information about the circuit we are constructing. However, the only remaining

constructs that we wish to define are dependent on each other – constructs to create, read and write mutable variables. To define them separately, we need an environment so that they can share information.

In hardware, a mutable variable corresponds to a delay element with an input-enable (recall `delayEn`). The state of the delay element is updated only when an assignment occurs that activates the enable line. So, an input to a mutable variable is defined as:

```
type Inp = (Bit, Bit)
```

The first element of the pair is the input-enable and the second is the actual input value being assigned to the variable.

The purpose of the environment is to the provide a mapping between variables and their inputs and outputs:

```
type Var = Int

data Env = Env { freshId    :: Var
               , readerInps :: Map Var [Inp]
               , writerInps :: Map Var [Inp]
               , outs       :: Map Var Bit }
```

A variable, which is represented by a unique integer, may have multiple inputs (one for each assignment to that variable in the program), but only one output. Notice that there are **two** mappings between variables and their inputs in the environment. The idea is that the `writerInps` mapping is used like a log, to **record** which variables have which inputs as we go. And the `readerInps` mapping is used to **lookup** the inputs to a variable. Ultimately, when it comes to running the `Recipe` monad (or "following the recipe"), the `writerInps` mapping will be passed circularly as the `readerInps` mapping:

```
follow              :: Bit -> Recipe a -> (Bit, a)
follow start recipe =  (fin, a)
  where
    (fin, env, a)   =  run recipe start initialEnv
    initialEnv      =  Env 0 (writerInps env) empty empty
```

The idea is similar to that of tying the output of a reader-writer monad to its own input, as presented by Russell O'Connor in the last edition of The Monad.Reader[5]. The idea was explained to me by Emil Axelsson, who uses something similar in Wired [6]. Apparently Koen Claessen has also used the idea in an old version of Lava, so it certainly seems quite useful!

It should not be too surprising that the environment is circular as the input to a variable could depend on its own output.

Before we're ready to implement mutable variables, we need a couple of helper functions over environments. The first of these, `createVar`, obtains a fresh variable from the environment and records a given value as that variable's output:

```
createVar       :: Env -> Bit -> (Var, Env)
createVar env a =  (v, env')
  where
    v           =  freshId env
    env'        =  env { freshId = v + 1
                       , outs    = insert v a (outs env) }
```

The second, `addInps`, takes a list of variable/input pairs and adds them to the `writerInps` mapping.

```
addInps         :: Env -> [(Var, [Inp])] -> Env
addInps env as =  env { writerInps = inps }
  where
    inps       =  unionWith (++) (fromList as) (writerInps env)
```

Mutable Variables

Recall that in hardware, a mutable variable corresponds to a delay element with an input-enable (`delayEn`). As `delayEn` is just a normal function, we first need to know what input to pass it. So we create a fresh variable `v` and then ask the environment for the inputs to `v`. We then combine all these inputs into one as follows:

- The overall input-enable line is equal to the disjunction of the enable lines of all the inputs. We assume that only one input-enable line is active at any one time.
- The overall input value is selected from all the inputs according to which input-enable line is high, using a kind of multiplexor.

Finally, the output from the delay element is recorded in the environment as being the output of `v`. So, the function to create a mutable variable, `newVar`, is defined as:

```
newVar :: Recipe Var
newVar =  Recipe $ \start env ->
            let (v, env') = createVar env out
                inps      = readerInps env ! v
                out       = delayEn low (enable, value)
                enable    = orl (map fst inps)
                value     = orl (map and2 inps)
            in  (start, env', v)
```

Obtaining the value of a variable is just a case of looking it up in the output mapping:

```
readVar   :: Var -> Recipe Bit
readVar v =  Recipe $ \start env -> (start, env, outs env ! v)
```

Writing a value to a variable is achieved by using a more general assignment function called `assign`:

```
writeVar     :: Var -> Bit -> Recipe ()
writeVar v a =  assign [(v, a)]
```

The more general function is capable of performing a number of assignments at the same time. It works by pairing the bit in each assignment with the start signal (which acts as the input-enable), and adding the resulting assignments to the `writerInps` mapping. Notice that like `skip`, assignment takes one clock cycle to complete:

```
assign    :: [(Var, Bit)] -> Recipe ()
assign as =  Recipe $ \start env ->
               let as' = map (\(a, b) -> (a, [(start, b)])) as
               in  (delay low start, addInps env as', ())
```

This interface to mutable state – with `newVar`, `readVar`, and `writeVar` – will be familiar to programmers who have used the `IORef` and `STRef` types in Haskell. However, to completely copy those interfaces, a `modifyVar` function is required:

```
modifyVar     :: (Bit -> Bit) -> Var -> Recipe ()
modifyVar f v =  do a <- readVar v ; writeVar v (f a)
```

A slight generalisation of `modifyVar` is `rewriteVar`, which gives the feeling that functions are being used as rewrite rules:

```
rewriteVar       :: (Bit -> Bit) -> Var -> Var -> Recipe ()
rewriteVar f v w =  do a <- readVar v ; writeVar w (f a)
```

The `rewriteVar` function provides a nice abstraction. It gives mutable state a more consistent and higher-level interface than seperate `readVar` and `writeVar` functions. Unfortunatly, rewriting is limited to "single variables at a time".

Generalised Rewriting

Rewrite rules would be much more powerful if they could be applied between arbitrary structures of variables, e.g. tuples and lists, rather than just single variables. This can achieved quite easily using a type class.

```
class Same a b | a -> b, b -> a where
  smap :: (Var -> Bit) -> a -> b
  szip :: a -> b -> [(Var, Bit)]
```

The idea is that `Same a b` holds between any two types `a` and `b` which have an identical structure – the only difference is that `a` contains variables at its leaves, whereas `b` contains bits. Figure 3 contains example instances of the `same` class.

Now we can define a generalised read function over structures of variables:

```
read   :: Same a b => a -> Recipe b
read a =  Recipe $ \s env -> (s, env, smap (outs env !) a)
```

And a generalised rewrite function:

```
rewrite       :: (Same a b, Same c d)
              => (b -> d) -> a -> c -> Recipe ()
rewrite f a b =  do x <- read a ; assign (szip b (f x))
```

It is common that the source and destination of a rewrite rule are the same:

```
apply f a = rewrite f a a
```

It is also common to initialise variables with a constant value:

```
set a b = rewrite (\() -> b) () a
```

Futhermore, it is useful to generalise `cond` and `iter` so that their conditions are functions over variables:

```
ifte f a p q = do b <- read a ; cond (f b) p q

while f a p  = do b <- read a ; iter (f b) p
```

The functions `rewrite`, `apply`, `ifte` and `while` provide a neat interface between structural Lava functions and behavioural recipes.

In this section, we have generalised over explicit reading and writing of individual variables, but not creation of variables. This is not a problem – `newVar` is what we want – but it can also be useful to create a list of variables of a given size, called a register:

```
newReg   :: Int -> Recipe [Var]
newReg n =  sequence (replicate n newVar)
```

This completes the definition of the Recipe library.

```
instance Same Var Bit where
  smap f a = f a
  szip a b = [(a, b)]

instance Same a b => Same [a] [b] where
  smap f a           = map (smap f) a
  szip [] bs         = []
  szip (a:as) []     = szip a (smap (const low) a) ++ szip as []
  szip (a:as) (b:bs) = szip a b ++ szip as bs

instance Same () () where
  smap f  () = ()
  szip () () = []

instance (Same a0 b0, Same a1 b1) => Same (a0, a1) (b0, b1) where
  smap f (a, b)          = (smap f a, smap f b)
  szip (a0, a1) (b0, b1) = szip a0 b0 ++ szip a1 b1

instance (Same a0 b0, Same a1 b1, Same a2 b2) =>
           Same (a0, a1, a2) (b0, b1, b2) where
  smap f (a, b, c)  = (smap f a, smap f b, smap f c)
  szip (a0, a1, a2)
       (b0, b1, b2) = szip a0 b0 ++ szip a1 b1 ++ szip a2 b2
```

Figure 3: Example instances of the `Same` class

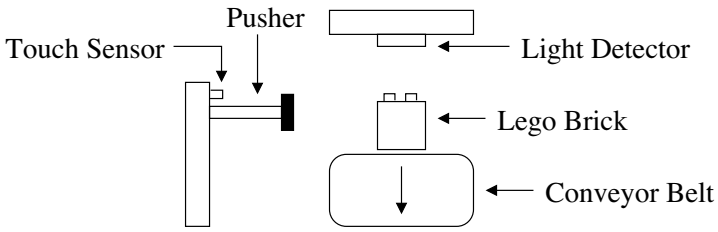

Figure 4: Lego brick-sorter (the brick is moving towards you)

Examples

In this section we use the Recipe library to implement a few short but useful programs: a sequential multipler, and a controller for a Lego brick-sorter.

A Sequential Multiplier

On some FPGA devices, multiplication in a single clock cycle is an expensive operation. It can therefore be useful to use a sequential multipler instead – one that takes several clock cycles to complete, but which has a much shorter critical-path delay, and which uses less FPGA resources. A sequential multiplier is usually implemented using the shift-and-add algorithm.

In Lava, numbers are typically represented as lists of bits, with the least significant bit coming first. In fact, such lists can be defined to be an instance of Haskell's `Num` class. Lava contains most of the required definitions already, in it's `Arithmetic` library.

The other ingredients we need, further to binary addition from the `Num` class, are functions for shifting numbers left and right:

```
shl       :: [Bit] -> Int -> [Bit]
a `shl` n =  drop n a ++ replicate n low

shr       :: [Bit] -> Int -> [Bit]
a `shr` n =  reverse (reverse a `shl` n)
```

Here, for simplicity, we assume the standard binary encoding of non-negative numbers i.e. we're not dealing with two's complement, so we just pad numbers with zeros rather than doing proper sign-extension.

The next step is to define the core of the algorithm as a rewrite-rule. If we're multipling `a` by `b` with an accumulator `acc`, then this step is as follows:

```
step              :: ([Bit], [Bit], [Bit]) -> ([Bit], [Bit], [Bit])
step (a, b, acc) =  (a `shr` 1, b `shl` 1, head b ? (acc+a, acc))
```

Now, to implement the multiplier, we just need to apply the `step` rule while `b` is not equal to zero:

```
mult        :: ([Var], [Var]) -> Recipe [Var]
mult (a, b) =  do acc <- newReg (length a)
                  set acc 0
                  while orl b (apply step (a, b, acc))
                  return acc
```

This recipe takes $n + 1$ clock cycles to complete in the worst case, where n is the bit-length of the multiplicand.

A Lego Brick-Sorter

As part this year's "Reactive Systems Design" module at the University of York, Jan Tobias Mühlberg designed a Lego brick-sorter that students would build and program during the practical sessions. As a demonstrator on this module, I couldn't resist developing a Haskell solution to the problem!

The basic structure of Tobias's brick-sorter is shown in Figure 4. It is intended to operate as follows:

- Initialise: the pusher-arm is moved into its resting position – the position in which it is touching the touch-sensor. Note that the pusher arm is either "moving" or "not moving" – there is no notion of direction. Mechanically, the pusher will automatically flip direction when it reaches as far as it can strecth, or when it reaches the resting position.

- Sort: when the light-detector dectects a reflective, light-coloured brick then the pusher should push the brick off the conveyor belt, and return to the resting position.

A program to control the brick-sorter according to this behaviour is shown in Figure 5. Since the Lego Brick-Sorter is controlled using the Mindstorms RCX micro-controller, I needed to develop a C backend for Lava [7] before this program could actually be used. Although Lava is more suited to describing large parallel circuit structures, Recipe programs are suitable for sequential, CPU-based archiectures too. The C code generated by Lava for a Recipe program will typically be fairly efficient.

Concluding Discussion

I have shown how a subset of Page and Luk's variant of Occam can be defined as a monadic library in Lava. In particular, I provided a combinator for applying normal Lava functions as rewrite rules over arbitrary structures of mutable variables. This results in a powerful combination of the structural and behavioural styles.

There is one issue in Page and Luk's Occam that leaves me slightly unsatisfied: when a variable is assigned two different values in parallel (in the same clock cycle), the behaviour of the circuit is undefined. In Recipe, I would like to investigate the use of **atomic** rewrite rules to obtain a more useful behaviour in such situations. This would potentially allow constructs such as communication channels to be defined within the language. The language BlueSpec already provides atomic actions, but it is quite different to Occam.

```
sorter                 :: (Bit, Bit) -> Recipe (Var, Var)
sorter (touch, light) =  do belt <- newVar
                            push <- newVar

                            set push high
                            waitUntil touch
                            set push low

                            set belt high
                              ||| forever (do waitUntil light
                                              set push high
                                              waitWhile touch
                                              waitUntil touch
                                              set push low)

                            return (belt, push)
```

Figure 5: Recipe to control brick-sorter

Acknowledgements

Finally, thanks to Jan Tobias Mühlberg for helping me get my Lava-generated C programs running on the Lego Mindstorms RCX. Without this goal, I may not have been encouraged to polish the Recipe library, which I initially wrote over two years ago! Thanks also to Emil Axelsson, Neil Mitchell, Tobias (again), and Tom Shackell for their comments on the first draft of this article.

References

[1] Koen Claessen and Mary Sheeran. A Tutorial on Lava. `http://www.cs.chalmers.se/~koen/Lava/tutorial.ps`.

[2] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131 (1992). `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[3] Mary Sheeran. Mary Sheeran's Web Page. `http://www.cs.chalmers.se/~ms/`.

[4] Ian Page and Wayne Luk. Compiling occam into field-programmable gate arrays. In W. Moore and W. Luk (editors), **FPGAs, Oxford Workshop on Field Programmable Logic and Applications**, pages 271–283. Abingdon EE&CS Books, 15 Harcourt Way, Abingdon OX14 1NV, UK (1991).

[5] Russell O'Connor. Assembly: Circular Programming with Recursive do. `http://www.haskell.org/sitewiki/images/1/14/TMR-Issue6.pdf`.

[6] Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In **Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)**, volume 3725 of **Lecture Notes in Computer Science**. Springer Verlag (October 2005).

[7] Jan Tobias Muehlberg and Matthew Naylor. FUN with Lego Mindstorms. `http://www.cs.york.ac.uk/plasma/talkrelated/318.pdf`.