Combinators for Message-Passing in Haskell

Neil C. C. BROWN

School of Computing, University of Kent, UK
neil@twistedsquare.com

Abstract. Much code in message-passing programs is tedious, verbose wiring code. This code is error prone and laborious – and tends to be repeated across many programs with only slight variations. By using type-classes, higher-order and monadic functions in Haskell, most of this code can be captured in re-usable high-level combinators that shorten and simplify message-passing programs. We motivate the design and use of these combinators via an example of a concurrent biological simulation, and explain their implementation in the Communicating Haskell Processes library.

1 Introduction

Message-passing programming is a type of imperative concurrent programming that eschews mutable shared state in favour of passing messages between concurrent processes. This paper is particularly concerned with systems featuring synchronous message-passing over point-to-point unbuffered channels (rather than address-based systems such as mailboxes). This style of concurrent programming has recently been successfully applied to biological and complex systems simulation [13], robotics [8], and can achieve good parallel speed-up on multicore machines [16]. Implementations exist as libraries in several functional languages, e.g. Concurrent ML [15] and Communicating Haskell Processes [2].

Message-passing programming supports a compositional model of programming, with processes comprised of sub-networks of communicating processes. However, message-passing languages and libraries do not typically provide easy ways in which to compose processes together, even when the composition is regular (e.g. a pipeline). Process wiring must be done "long hand", declaring channel variables/arrays and passing them to the appropriate process. This style of wiring is tedious, verbose and error-prone.

In contrast, higher-order functional programming allows common coding patterns to be captured and re-used. For example, operations on lists can typically be implemented using some combination of map, filter, or a fold. It is rare to write a function that directly processes a list via pattern-matching, because the operation can often be expressed using one of the aforementioned functions.

Haskell has seen a proliferation of further abstractions based on type-classes, such as applicative functors [10], monads [12] and arrows [7]. These abstractions capture particular patterns of computation, and allow general helper functions

(e.g. mapM) to act on all instances of this pattern; code re-use is supported by parameterising the helper functions with the type-class in question.

This paper contends that patterns in message-passing programming can be captured using functional programming techniques such as higher-order functions and type-classes. This paper's contribution is the introduction of new combinators for message-passing systems which **shorten and simplify code**: **wiring functions** for common process topologies (section 4), which can be generalised into **a composition monad** for more flexible wiring (section 5).

These abstractions are motivated and demonstrated using a central biological simulation example introduced in section 3. All of these new abstractions have been implemented using standard Haskell, and have been added to the Communicating Haskell Processes library, which is introduced in section 2.

2 Background: Communicating Haskell Processes

Communicating Haskell Processes (CHP) is a Haskell library that supports concurrent synchronous message-passing [2], and is based on the Communicating Sequential Processes calculus [6, 17]. As with most imperative Haskell libraries, it provides a monad (named CHP) in which all of its actions take place. Its basic API provides channel creation and communication:

```
newChannelWR :: CHP (Chanout a, Chanin a)
writeChannel :: Chanout a -> a -> CHP ()
readChannel :: Chanin a -> CHP a
```

Note how the channels are used via two ends: the outgoing end (Chanout) on which values are sent, and the incoming end (Chanin) on which values are received. This separation between the two ends at the type level helps prevent mistakes – such as connecting two reading processes together with a channel, resulting in deadlock. It also promotes code clarity: making it clear from the type of a process whether it will send or receive on each channel.

We refer to something that has type CHP r as being a *complete* CHP process (one that is ready to run). Anything that will be a complete CHP process when given further arguments (e.g. Chanin a ->Chanout a ->CHP ()) is referred to simply as a CHP process. An example of a basic CHP process is the identity process that forwards values from one channel to another[1]:

```
idP :: Chanin a -> Chanout a -> CHP ()
idP input output = forever (readChannel input >>= writeChannel output)
```

CHP processes can be composed in parallel using the commutative, associative runParallel function which waits for all the parallel processes to terminate before returning a list of their results:

```
runParallel :: [CHP a] -> CHP [a]; runParallel_ :: [CHP a] -> CHP ()
```

[1] In this paper we suffix these simple processes with "P" to avoid confusion, here with the Haskell identity *function* (id ::a ->a).

The version with an underscore suffix discards the results of the parallel computations. The type of these functions exactly matches that of the standard monadic sequence functions, specialised to the CHP monad:

```
sequence :: [CHP a] -> CHP [a]; sequence_ :: [CHP a] -> CHP ()
```

2.1 Barriers and Enrolling

As well as channels, CHP also features barriers. A barrier is a synchronisation primitive that can only be used by processes enrolled on (i.e. members of) the barrier. When an enrolled process wishes to synchronise on the barrier, it must wait for all other enrolled processes to also do so. Barriers are created with an enrollment count of zero, using one of the functions:

```
newBarrier :: CHP Barrier; newBarrierPri :: Int -> CHP Barrier
```

The latter function features priority: the default is 0, and larger numbers indicate higher priority. When a process can choose between completing two barriers, the higher priority barrier will be chosen. Barriers feature a "scoped" API for enrolling, that eschews explicit enroll and resign (de-enroll) calls in favour of taking as an argument the block of code to execute while enrolled:

```
enroll :: Barrier -> (EnrolledBarrier -> CHP a) -> CHP a
```

The enroll function takes a barrier and a CHP process that operates on the enrolled barrier. The returned completed CHP process enrolls the given process on the barrier for the duration of its execution and resigns afterwards. To prohibit attempts to synchronise without first enrolling, synchronisation is only possible on the EnrolledBarrier type:

```
syncBarrier :: EnrolledBarrier -> CHP ()
```

As an example, the following code enrolls twice on a barrier, then runs two corresponding processes in parallel that repeatedly synchronise on the barrier:

```
do bar <- newBarrier
   enroll bar (\eb0 -> enroll bar (\eb1 ->
     runParallel [replicateM_ 100 $ syncBarrier eb0, replicateM_ 100 $ syncBarrier eb1]))
```

Note that it is crucial that both enrollments happen before the parallel composition (rather than in each parallel branch). Consider the alternative code:

```
do bar <- newBarrier
   runParallel [enroll bar (replicateM_ 100 . syncBarrier)
               ,enroll bar (replicateM_ 100 . syncBarrier)]
```

The barrier begins with an enrollment count of zero. When, in the above code, the first parallel branch runs, it will enroll, increasing the enrollment count to one. When it then tries to synchronise on the barrier, it may do so by itself. Thus one branch can enroll and (potentially) perform all 100 synchronisations before the other branch starts to run (and do the same). Thus, for the branches

to synchronise together, the enrollment of both processes must occur before the parallel composition begins.

CHP already features two helper functions for enrolling, which hint at the combinator-based approach seen later in the paper. The enrollList function enrolls a single process on a whole list of barriers (nesting the process inside all the enrollments), while the enrollAll function enrolls each of a list of processes on a single barrier:

```
enrollList :: [Barrier] -> ([EnrolledBarrier] -> CHP a) -> CHP a
enrollList [] f = f []
enrollList (b:bs) f = enroll b (\eb -> enrollList bs (\ebs -> f (eb:ebs)))

enrollAll :: Barrier -> [EnrolledBarrier -> CHP a] -> CHP [a]
enrollAll b ps = enrollList (replicate (length ps) b) (runParallel . zipWith ($) ps)
```

The previous example can thus also be written as:

```
newBarrier >>= flip enrollAll [replicateM_ 100 . syncBarrier, replicateM_ 100 . syncBarrier]
```

3 Motivating Example: Blood Clotting Simulation

Section 2 introduced the existing Communicating Haskell Processes library. This section provides a motivating example for the design and inclusion of the new features in the library introduced in future sections. The example is a concurrent simulation of blood clotting, with "sticky" platelets moving down a one-dimensional pipeline of site processes. It is inspired by the example presented by Schneider et al. [19], and has been converted to CHP to use some advanced concurrency features such as conjunction [3].

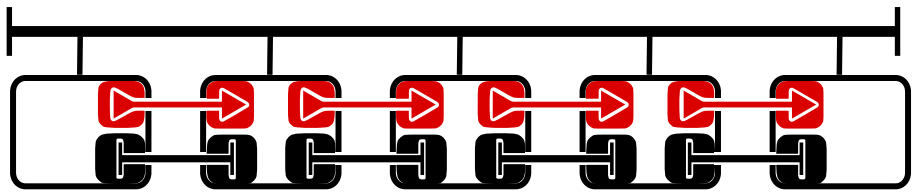

Fig. 1. Illustration of how the blood example is connected together. The first (left-most) process is a platelet generator, and the last (right-most) is a platelet consumer. The three processes in the centre are examples of the site processes (there are 100 in the real model). The processes are connected to their neighbours with a channel (the arrows) carrying platelets, and a barrier (drawn as a line with perpendicular ends). All the processes also enroll on a shared "tick" barrier (shown above the processes).

Platelets move (in a consistent direction) along a one dimensional pipeline. On each time-step a platelet may move or not move, with the following rule: if there are platelets immediately before or immediately after it in the pipeline, a

platelet will only move forwards if they do so too. Each platelet may refuse to move on a given time-step with probability 5%. We model the sites (locations which can either hold a single platelet, or be empty) as active processes, and the platelets as passive data that passes between the sites. An illustration of their connectivity is given in figure 1.

The new features introduced later in this paper will demonstrate the power of a functional combinator-based approach. To provide a contrast to the existing methods that must be used in other imperative languages, such as occam or libraries for Java, we first present the example in figure 6 using idioms from imperative languages, such as numeric indexing. The exact definitions of the processes are not relevant in this paper and are thus omitted for brevity.

4 Wiring: Process Composition

In message-passing systems with typed channels, a substantial part of the programming model is the composition of processes using channels. For example, we may want to compose together the mapP and filterP processes (analogues of the standard list-processing functions) into a process that filters out negative numbers and then turns the remaining positive numbers into strings:

```
showPosP :: Chanin Int -> Chanout String -> CHP ()
showPosP input output = do (w, r) <- newChannelWR
                           runParallel_ [filterP (> 0) input w, mapP show r output]
```

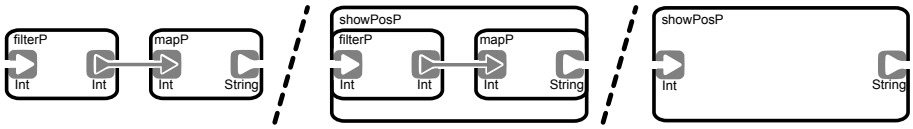

Fig. 2. The composition of filterP and mapP, as shown in the left-hand diagram. This composition becomes an opaque box to other components, as shown progressively in the middle and right diagrams. This component can then be further composed in a similar manner. The programming model used in CHP is thus compositional, allowing complex networks to be built from joining together different components without regard to their internal implementation.

This is shown diagrammatically in figure 2. It is instructive to note that the composition of two such processes with a single input channel and single output channel is itself a process with a single input channel and a single output channel. This component can then be re-used without requiring any knowledge of its internally concurrent implementation.

4.1 Simple Composition Operator

This composition of two single-input, single-output processes is so common that it is worth capturing in an associative operator:

```
(==>) :: (Chanin a -> Chanout b -> CHP ()) -> (Chanin b -> Chanout c -> CHP ())
      -> (Chanin a -> Chanout c -> CHP ())
(==>) p q r w = newChannelWR >>= \(mw, mr) -> runParallel_ [p r mw, q mr w]
```

The previous showPosP process can be written using this operator as follows:

```
showPosP = filterP (> 0) ==> mapP show
```

This point-free style is clearer and more elegant. By not introducing extra variable names we eliminate potential mistakes (mis-wiring). It can be seen that this process composition operator is an analogue of function composition.

We do not, however, always want to connect processes merely with a single unidirectional channel. We may want to connect processes with a pair of channels (one in each direction) or three channels, or a channel and a barrier, etc., as for example in the main function in our blood clotting example in figure 6 – which means that we need a more general operator than the one above.

4.2 Richer Composition Operator

Figure 3 shows another example of process composition, requiring different connections than figure 2. The types and directions of the channels needed to compose the processes are readily apparent – it should be just as easy to join these processes with two channels as it was to join filterP and mapP with one.

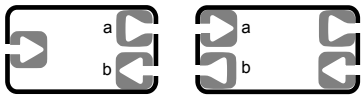

Fig. 3. An example of slightly different process composition than figure 2. The letters indicate the types of the channel-ends that each process takes. It is readily apparent, both that these processes *can* be composed, and *how* they should be composed: with a pair of channels.

To generalise the variety of composition possible, we use Haskell's type-class mechanism. We define a two parameter type-class, Connectable, an instance of which indicates that the two parameters can be wired together in some fashion, and provide a function that must be implemented to do so:

```
class Connectable l r where
  connect :: ((l, r) -> CHP a) -> CHP a
```

Instances for channels (in both directions) are trivial:

```
instance Connectable (Chanout a) (Chanin a) where
  connect p = newChannelWR >>= p

instance Connectable (Chanin a) (Chanout a) where
  connect p = newChannelWR >>= (p . swap)
    where swap (x, y) = (y, x)
```

We choose this style of function to compose the processes, rather than say connect ::CHP (l, r), because we may need to enroll the processes on the synchronisation object for the duration of their execution. Our chosen style of function allows us to do just that for an instance involving barriers:

```
instance Connectable EnrolledBarrier EnrolledBarrier where
  connect p = do b <- newBarrier
                 enroll b (\b0 -> enroll b (\b1 -> p (b0, b1)))
```

The instance that grants much greater power to the Connectable interface is the one that works for any pair of Connectable items:

```
instance (Connectable lA rA,Connectable lB rB) => Connectable (lA, lB) (rA, rB) where
  connect p = connect (\(ax, ay) -> connect (\(bx, by) -> p ((ax, bx), (ay, by))))
```

This instance means that two processes can easily be wired together if they need to be connected by a channel *and* a barrier, for example. Similar instances can also be constructed for triples and so on. Programmers may also create their own instances (as with any Haskell type-class) for synchronisation primitives not known to the library, or for compound data structures that feature several synchronisation primitives that need to be wired together differently.

A particularly powerful way to enhance this operator would be to use session types on CHP channels. Session types generalise from carrying a particular type on a one-way channel (as CHP currently does) to specifying the series of communications that can take place in both directions between two participants, encapsulating the entire protocol between two parties in the channel type. It has been shown that session types can be embedded well in a Haskell setting [14].

The Connectable interface is a suitable basic API, but it is too unwieldy to compose processes together. We can use it to define a more general version of the composition operator seen earlier:

```
(<=>) :: Connectable l r =>
    (a -> l -> CHP ()) -> (r -> b -> CHP ()) -> (a -> b -> CHP ())
(<=>) p q x y = connect (\(l, r) -> runParallel_ [p x l, q r y])
```

The type of this operator is very general. No restrictions are placed on the "outer" types a and b (which may be channels, but are not required to be so). This operator composes together any pair of two-argument processes where the second argument of the first process can be connected to the first argument of the second process. We can also trivially define other operators that are useful at the start and end of a process pipeline, respectively, and that compose just a start and end process:

```
(|<=>) :: Connectable l r => (l -> CHP ()) -> (r -> b -> CHP ()) -> (b -> CHP ())
(<=>|) :: Connectable l r => (a -> l -> CHP ()) -> (r -> CHP ()) -> (a -> CHP ())
(|<=>|) :: Connectable l r => (l -> CHP ()) -> (r -> CHP ()) -> CHP ()
```

We also provide a pipelineComplete function in the next section to support combining one start process and one end process with multiple middle processes.

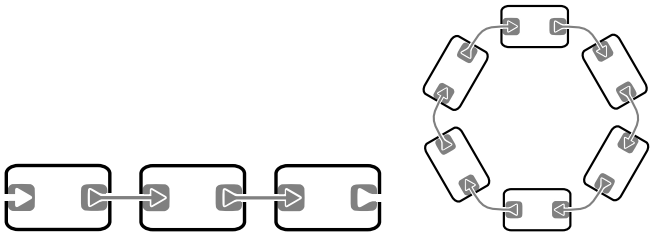

Fig. 4. The pipeline topology (left) and cycle topology (right). It can be seen that a cycle can be formed simply by connecting the two end points of a pipeline together. The processes are illustrated here by connecting them with a single channel, but any regular interface could be connected together using the Connectable class.

4.3 Capturing Common Topologies

We do not always want to simply compose two adjacent processes. Another common requirement is to wire together a pipeline of processes. We can do this by building on top of our connectable operator, meaning that the helper function is parameterised by the type of connection between processes, but fixes the topology – we can then easily extend this to a cycle (also known as a ring):

```
pipeline :: Connectable r l => [l -> r -> CHP ()] -> l -> r -> CHP ()
pipeline = foldr1 (<=>)

cycle :: Connectable r l => [l -> r -> CHP ()] -> CHP ()
cycle ps = connect (\(l, r) -> pipeline ps l r)
```

Both topologies are depicted in figure 4. We can also define a function for connecting a complete pipeline, as discussed at the end of the previous section:

```
pipelineComplete :: Connectable l r =>
  (l -> CHP ()) -> [r -> l -> CHP ()] -> (r -> CHP ()) -> CHP ()
pipelineComplete begin middle end = (begin |<=> pipeline middle) |<=>| end
```

This idea of capturing topology extends beyond such one-dimensional structures. A common requirement when building concurrent simulations with the CHP library is to form a regular two-dimensional (or three-dimensional) grid, either with or without diagonal connections. Producing such wiring, especially with diagonal connections, is verbose and error prone. Without the Connectable interface, it would have to be replicated for each type of channel used, increasing the possibility for error (this was originally the case in the CHP library [2]). But we can now write the function once, test it to show its correctness once, and re-use it repeatedly in different programs. We show an example type here but omit the lengthy definition[2]:

```
grid4way :: (Connectable right left, Connectable bottom top) =>
  [[above -> below -> left -> right -> CHP r]] -> CHP [[r]]
```

[2] It can be found in the library at `http://hackage.haskell.org/package/chp-plus`; an alternate short implementation is given in section 5.2 of this paper.

The parameter is a list of rows of processes (which must be rectangular); the result is a corresponding list of rows of results. The processes are wired together into a regular grid where the far right edge also connects to the far left edge, and the bottom edge to the top: this forms a torus shape.

Any topology (especially regular topologies) can be captured in helper functions like those given above, and re-used regardless of the channel types required to connect the processes.

4.4 Improved Process Wiring: Blood Clotting Example

The blood clotting example shown in figure 6 wired up its pipeline of processes by creating a list of channels and a list of barriers. List indexing was used to access the corresponding channels and barriers for each process. The connectable operators and functions introduced in the previous sections allow the processes to be wired together using a couple of the new operators and the pipeline combinator. This combinator is a list fold which replaces imperative-style list indexing.

The main feature of programming with CHP that enables the process wiring operators is the use of first-class processes[3]. In other languages where processes cannot be passed around, a function such as pipeline would not be possible to define. For example, the occam language does not have first-class processes. The C++CSP concurrent programming library allows complete processes (instances of classes that inherit from a CSProcess class) to be passed around, but processes still requiring channels is neither a straightforward nor natural idiom to support.

The revised version of the main process of the blood clotting example using the connectable operators where possible is shown in figure 7 and can be contrasted to figure 6. The new code using the connectable operators is much shorter. It is also instructive to note that there is no longer a call to the runParallel_ function in the main wiring function. The concurrency, which is a central primitive of CHP, has been captured in the pipelineComplete wiring function. This is indicative of the higher-level nature of the new process wiring, which abstracts away the details of the parallelism (and removes the channel declarations) in favour of operators that capture the connectivity pattern being used to join together the processes.

5 Compositional Wiring

Section 4 outlined ways to compose processes into a complete whole. We often have situations where a process needs not just one set of connections, but also some other cross-cutting connection. For example, a cycle of processes may all be connected to their neighbours with a channel – but they may also all be enrolled together on a barrier (as illustrated in figure 5). We have a similar situation in our blood clotting example, depicted in figure 1.

[3] Since a CHP process is a function/monadic action, these being first-class in Haskell means that CHP has first-class processes.

Consider how to implement such an arrangement with the combinators that we have introduced thus far; we have (with specialised types for illustration):

```
enrollAll :: Barrier -> [EnrolledBarrier -> CHP a] -> CHP [a]
pipeline :: [Chanin a -> Chanout a -> CHP ()] -> Chanin a -> Chanout a -> CHP ()
```

Both processes expect a list of processes that take exactly the required arguments (a barrier or a channel pair, respectively) and return a CHP process. Neither supports partial application that would return a process ready to be wired up by the other function: in short, these combinators do not compose.

We cannot simply create a function without the CHP monad, such as:

```
pipeline' :: [Chanin a -> Chanout a -> b] -> Chanin a -> Chanout a -> [b]
```

We require access to the CHP monad in order to run the processes in parallel, and to create the channels used to connect them together. This means that we need a different strategy in order to support composing these combinators in a useful way. To that end, we introduce a Composed monad.

5.1 The Composed Monad

We need to abstract over the return types of the processes being composed together while still allowing access to the functionality in the CHP monad. We therefore create functions such as (again with types specialised for illustration):

```
enrollAllR :: Barrier -> [EnrolledBarrier -> a] -> Composed [a]
pipelineR :: [Chanin a -> Chanout a -> b] -> Chanin a -> Chanout a -> Composed [b]
cycleR :: [Chanin a -> Chanout a -> b] -> Composed [b]
```

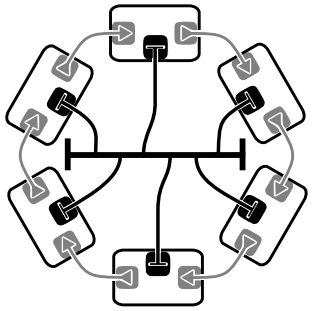

Fig. 5. A ring of processes connected to their neighbours with a single channel, and also all enrolled together on the same central barrier.

Given a list of `processes ::[EnrolledBarrier ->Chanin a ->Chanout a ->CHP ()]`, we can compose them, as depicted in figure 5, simply using:

```
enrollAllR b processes >>= cycleR
```

The meaning of composition in this monad is not intuitively the sequencing of actions as is often the case for monads (in fact, the monad is conceptually commutative in many cases). It is instead a form of nesting – the code above enrolls the processes on the barrier, and inside the scope of that enrollment it wires them together in a cycle. From a user's perspective the monad can be thought of as a series of wiring instructions. Each command composes the processes further until finally the complete processes are returned: the output of any Composed block is almost always such a list of complete CHP processes ready to be run in parallel. The type of the Composed monad is:

```
newtype Composed a = Composed { runWith :: forall b. (a -> CHP b) -> CHP b }

instance Monad Composed where
  return x = Composed (\r -> r x)
  (>>=) m f = Composed (\r -> m `runWith` ((`runWith` r) . f))
```

This type is not without precedence as a monad; it is equivalent to the continuation-passing monad transformer on top of CHP, `forall b. ContT b CHP a`, and is technically the codensity monad of CHP. The monad is not used to pass continuations, however. The intuition is that any type wrapped in Composed needs to be told how it can be turned into a CHP action, and then it becomes that CHP action. At the outer-level this is accomplished with runParallel:

```
run :: Composed [CHP a] -> CHP [a]
run ps = ps `runWith` runParallel
```

5.2 Composed Wiring Functions

We can re-define all the wiring functions seen earlier in the new Composed monad. The most basic are the connectR and enrollR functions:

```
connectR :: Connectable l r => ((l, r) -> a) -> Composed a
connectR p = Composed (\r -> connect (r . p))

enrollR :: Barrier -> (EnrolledBarrier -> a) -> Composed a
enrollR b p = Composed (\r -> enroll b (r . p))
```

The latter can easily be expanded into an enrollAllR function:

```
enrollAllR :: Barrier -> [EnrolledBarrier -> a] -> Composed [a]
enrollAllR b ps = mapM (enrollR b) ps
```

The enrollAllR function enrolls a list of processes on the given barrier. Without the Composed monad it is an intricate recursive function, but with the Composed monad it is a non-recursive and straightforward mapM call.

We can define the pipelineR function as follows:

```
pipelineR :: Connectable l r => [r -> l -> a] -> Composed (r -> l -> [a])
pipelineR [] = return (\_ _ -> [])
pipelineR (firstP:restP) = foldM adj (\x y -> [firstP x y]) restP
 where adj p q = connectR (\(l, r) x y -> (p x l) ++ [q r y])
```

As before, the cycleR function is a small addition to the pipelineR function:

```
cycleR :: Connectable l r => [r -> l -> a] -> Composed [a]
cycleR [] = return []
cycleR ps = pipelineR ps >>= connectR . uncurry . flip
```

With these composition operators we can now easily define the 4-way grid composition discussed earlier in section 4.3:

```
grid4wayR :: (Connectable below above, Connectable right left) =>
  [[above -> below -> left -> right -> a]] -> Composed [[a]]
grid4wayR = (mapM cycleR . transpose) <=< (mapM cycleR . transpose)
```

The inherent symmetry, and regularity, of the combinator is exposed, and its cycleR-based definition trivial with the help of the standard list function transpose that swaps rows for columns in a list of lists and the (<=<) :: Monad m =>(b ->m c) ->(a ->m b) ->a ->m c function that composes two monadic functions.)

It is possible for users to define their own wiring functions using this monad. For example, a user may have a repeated pattern in their program, such as a list of processes where they wish to enroll all the processes at odd positions in the list on one barrier, but all the processes at even positions in the list on another barrier. They could write a function to do this, and use it in different situations in combination with other functions – for example, one such list may further be wired into a pipeline, while another may be wired into a star topology.

The use of wiring combinators avoids explicitly declaring and naming the channels and barriers required to construct the process network. This makes the code shorter, and prevents errors (such as passing the wrong channel-end to the wrong process, which will compile if they have the same type). It also means that common topologies (such as pipelineR) can be recognised by name when reading code – it is not straightforward to recognise wiring patterns when they are written out "long-hand" with individual named channels.

5.3 Further Improved Process Wiring: Blood Clotting Example

The motivation behind the Composed monad was that our original combinators did not easily compose. Certain combinators, such as enrollAll and pipeline, cannot easily be used together. For this reason our previous simplification of the blood platelets' wiring in figure 7 used enrollList instead. Often, nesting the combinators like this can lead to code nested many levels deep that is hard to follow, with many extra named parameters that are hard to track.

Our new Composed monad allows us to simplify the wiring in our blood clotting example even further by using two combinators: see figure 8 for the result. It can be seen that the only communication primitive that is named is the tick barrier. There are no manipulations involving list indexing as before. All of the creation of channels and barriers (except for tick) and all of the concurrency is hidden in the combinators for the Composed monad; pipelineCompleteR and enrollAllR create the channels and barriers, while the run function runs all the resulting processes concurrently.

```
plateletGenerator :: (Chanout Platelet, EnrolledBarrier) -> EnrolledBarrier -> CHP ()

plateletConsumer :: (Chanin Platelet, EnrolledBarrier) -> EnrolledBarrier -> CHP ()

site :: (Chanin Platelet, EnrolledBarrier) -> (Chanout Platelet, EnrolledBarrier)
     -> EnrolledBarrier -> CHP ()

numSites = 100

main :: IO ()
main = runCHP_ $ do
  (writers, readers) <- unzip <$> replicateM (numSites + 1) newChannelWR
  bars <- replicateM (numSites + 1) newBarrier
  tick <- newBarrierPri (-1)
  enrollList (replicate (numSites + 2) tick) $ \ticks ->
    enrollList bars $ \ebars -> runParallel $
      [site (readers !! i, ebars !! i)
            (writers !! succ i, ebars !! succ i)
            (ticks !! i) | i <- [0..numSites-1]] ++
      [plateletGenerator (writers !! 0, ebars !! 0) (ticks !! numSites)
      ,plateletConsumer (readers !! numSites, ebars !! numSites)
                        (ticks !! succ numSites)]
```

Fig. 6. An example version of the blood clotting example that uses array-like indexing idioms for wiring. The internal definition of the processes being wired together is not relevant (their types are given here to aid understanding), and the network is depicted in figure 1.

```
main = runCHP_ $ do
  tick <- newBarrierPri (-1)
  enrollList (replicate (numSites + 2) tick) $ \ticks ->
    pipelineComplete (flip plateletGenerator (ticks !! numSites))
                     (map (flip2 site) (take numSites ticks))
                     (flip plateletPrinter (ticks !! succ numSites))
  where flip2 f c a b = f a b c
```

Fig. 7. A revised version of the wiring code originally shown in figure 6, which uses the pipeline combinator and other new operators to simplify the wiring of the process network.

```
main = runCHP_ $ newBarrierPri (-1) >>= \tick -> run $
  pipelineCompleteR plateletGenerator (replicate numSites site) plateletConsumer
    >>= enrollAllR tick
```

Fig. 8. A further revised version of the wiring code shown originally in figure 6 (and previously revised in figure 7). This time the Composed monad is used to reduce the complete wiring code to just a few lines.

6 Related Work

Several other message-passing libraries exist in functional programming languages. Concurrent ML is the most obvious precursor [15], and it has since been converted to Haskell, too [18, 4]. Given support for type-classes or a comparable mechanism, there is no reason why the programming patterns captured in this paper could not also be captured in Concurrent ML.

Erlang is a functional programming language with a strong message-passing component. However, Erlang uses asynchronous messages sent to a particular process address, rather than channels. This difference is vital with respect to the work described in this paper; the process composition described here does not apply to Erlang, and the styles of process that are composed in this paper are not common in Erlang. Additionally, Erlang is dynamically typed, which precludes the type-based connectable operators seen in this paper.

Lava is a hardware design domain-specific language embedded in Haskell [1]. Lava featured operators to compose together digital circuit components. This is an analogue of the Connectable operators seen in this paper – although Lava featured different combinators depending on data-flow direction, whereas the Connectable class abstracts away details such as directionality and types.

At an abstract level, CHP can be thought of as a way to represent interactive computations. Another way to do so is Functional Reactive Programming (FRP) [11]. There are various implementations of FRP [11, 5, 9], but broadly they represent interaction as a function from timed observations/inputs to timed outputs. This neatly removes explicit state and imperative constructs, but can cause problems with causality (where future events can affect past behaviour).

7 Conclusions

The Communicating Haskell Processes library is an imperative message-passing library built in a functional programming language. This paper has shown how the ideas of higher-order functions, type-class-based abstractions and re-usable combinators can be taken from functional programming and applied to message-passing programming, with all of the same benefits.

CHP programs are made up of many components composed together concurrently, and connected by channels and barriers. The "long-hand" way of composing these processes – manually declaring channels and passing the ends to the right processes – is tedious, verbose and error prone. The combinators discussed in this paper allow for an elegant and concise point-free style, composing processes together without ever naming the primitives that connect the processes.

The Connectable type-class allows the wiring functions to abstract away from the primitives used to compose processes and to instead focus on capturing topology. This allows complicated functions (such as two-dimensional grids with diagonal connections) to be written once and re-used. The Composed monad takes this further and allows complicated composition with several cross-cutting concerns to be done easily and compositionally, which makes for completely flexible

wiring of processes. Both of these mechanisms could generalise to composing processes with any Haskell communication primitive such as MVar or TChan.

All of this work is only possible because functions and processes are first-class in CHP, and can thus be passed as arguments. Implementing these combinators in message-passing frameworks in other languages would either be overly verbose and awkward (e.g. using interfaces and classes in Java) or simply not possible (e.g. in the language occam, where higher-order programming is not possible).

References

1. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: hardware design in Haskell. In: ICFP '98. pp. 174–184. ACM (1998)
2. Brown, N.C.C.: Communicating Haskell Processes: Composable explicit concurrency using monads. In: Communicating Process Architectures 2008. pp. 67–83 (Sep 2008)
3. Brown, N.C.C.: Conjoined Events. In: Advances in Message Passing, 2010. ACM (Jun 2010)
4. Chaudhuri, A.: A concurrent ML library in concurrent Haskell. In: ICFP '09. pp. 269–280. ACM (2009)
5. Elliott, C.M.: Push-pull functional reactive programming. In: Haskell '09. pp. 25–36. ACM (2009)
6. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
7. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. 37(1-3), 67–111 (2000)
8. Jadud, M., Jacobsen, C.L., Simpson, J., Ritson, C.G.: Safe parallelism for behavioral control. In: 2008 IEEE Conference on Technologies for Practical Robot Applications. pp. 137–142. IEEE (November 2008)
9. Liu, H., Cheng, E., Hudak, P.: Causal commutative arrows and their optimization. In: ICFP '09. pp. 35–46. ACM (2009)
10. McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. 18(1), 1–13 (2008)
11. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Haskell '02. pp. 51–64. ACM (2002)
12. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: POPL '93. pp. 71–84. ACM (1993)
13. Polack, F.A., Andrews, P.S., Sampson, A.T.: The engineering of concurrent simulations of complex systems. In: 2009 IEEE Congress on Evolutionary Computation (CEC 2009). pp. 217–224. IEEE Press (May 2009)
14. Pucella, R., Tov, J.A.: Haskell session types with (almost) no class. In: Haskell '08. pp. 25–36. ACM (2008)
15. Reppy, J.H.: Concurrent Programming in ML. Cambridge University Press, Cambridge, England (1999)
16. Ritson, C.G., Sampson, A.T., Barnes, F.R.M.: Multicore Scheduling for Lightweight Communicating Processes. In: COORDINATION 2009. Lecture Notes in Computer Science, vol. 5521, pp. 163–183. Springer (June 2009)
17. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall (1997)
18. Russell, G.: Events in haskell, and how to implement them. In: ICFP '01. pp. 157–168. ACM (2001)
19. Schneider, S., Cavalcanti, A., Treharne, H., Woodcock, J.: A Layered Behavioural Model of Platelets. In: ICECCS-2006. pp. 98–106. IEEE (Aug 2006)