Abstractions for Message-Passing

What can be learned from functional programming

Neil C. C. BROWN

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

Abstract

Functional programming languages such as Haskell use type-classes and higher-order functions to capture common coding patterns (e.g. map, monads) and re-use them. This paper applies the same techniques to message-passing programming (within functional languages). Abstractions are introduced to capture common process behaviours and wiring topologies for re-use. These abstractions can make the code built on top of them clearer and more concise.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

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

Languages such as Haskell have seen a proliferation of further abstractions based on type-classes – applicative functors [8], monads [9] and arrows [7] being some of the most popular. These abstractions capture certain patterns of computation, and allow general helper functions (such as mapM) to act on all instances of this pattern.

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). Implementations exist as libraries in several functional languages, e.g. Concurrent ML [10] and Communicating Haskell Processes[3].

This paper contends that there is scope for capturing abstractions for message-passing programming in the same way that has been done for functional programming. This paper will detail several new such abstractions for message-passing systems that have been added to the Communicating Haskell Processes library (which is introduced in section 2):

- higher-level processes, that take functions or processes as arguments or communicate them over channels (section 3);
- wiring functions for connecting common process topologies (section 4), which can be generalised into
- a composition monad for more flexible wiring (section 5); and
- behaviours, for describing different combinations of actions that a process is willing to engage in (section 6).

All of these are library features built using standard Haskell without requiring any new language extensions (merely one or two already commonly in use).

2. Introduction to Communicating Haskell Processes

Communicating Haskell Processes (CHP) is a Haskell library that supports concurrent synchronous message-passing [3], based on the Communicating Sequential Processes calculus [6, 11]. 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.

We refer to something that has type CHP r as being a *complete* CHP process (one that is ready to run). Anything that when given further arguments will be a complete CHP process (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:

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

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

2.1 Parallel Composition

CHP processes can be composed in parallel. Parallel composition runs all the processes in parallel, and waits for them all to terminate before returning a list of their results:

```
runParallel :: [CHP a] -> CHP [a]
```

The type of this process exactly matches that of the standard monadic sequence function specialised to the CHP monad:

```
sequence :: [CHP a] -> CHP [a]
```

The library also features operator versions of parallel composition: one that retains the results and one that discards them (the latter is actually used more often):

```
(<||>) :: CHP a -> CHP b -> CHP (a, b)
(<|*|>) :: CHP a -> CHP b -> CHP ()
```

2.2 Enrolling

As well as channels, CHP also features barriers: synchronisation primitives with persistent enrollment which require all enrolled processes to synchronise together. Barriers feature a standard *scoped* enrollment API:

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

This function takes a barrier, and a function that operates on the enrolled barrier, and enrolls the given function on the barrier for the duration of its execution before resigning; all this is returned as a complete CHP process. Synchronisation is only possible on the `EnrolledBarrier` type, to prohibit attempts to synchronise without first enrolling.

2.3 Choice

CHP supports choice between actions: that is, waiting for the first of several actions to occur, and engaging in exactly one of them. For example, this process reads in an input value, then sends it out on the first channel on which it is able:

```
p :: Chanin a -> Chanout a -> Chanout a -> CHP ()
p input outputA outputB = forever
  (do x <- readChannel input
      writeChannel outputA x <-> writeChannel outputB x)
```

The choice operator, "`<->`", also has a list form named `alt`; their types are:

```
(<->) :: CHP a -> CHP a -> CHP a
alt :: [CHP a] -> CHP a
```

The choice operator allows choice between monadic blocks of code, not just single actions. In these cases, the choice is always between the leading action of each block. Discussion of this feature is reserved for the appendix A at the end of the paper.

3. Processes and Functions

CHP processes can take any type as a parameter; common examples include integers, strings and channel-ends. Processes can also take functions as parameters, which allows powerful processes to be constructed. The most basic, but also one of the most useful processes, is the map process:

```
mapP :: (a -> b) -> Chanin a -> Chanout b -> CHP ()
mapP f input output = forever
  (readChannel input >>= writeChannel output . f)
```

This is very similar to the identity process but it applies a modification function on the value as it is passed through (a strict version that forces evaluation can also be constructed).

It is also possible to construct a filter process[1]:

```
filterP :: (a -> Bool) ->
           Chanin a -> Chanout a -> CHP ()
filterP keep input output
  = forever (do x <- readChannel input
                when (keep x) (writeChannel output x))
```

This process repeatedly reads in values from an input channel, but only sends on those values which meet the given criteria.

The ability to pass functions into concurrent processes allows *communication* behaviour to be captured in a common process, parameterised by the "business logic" that operates on the values being passed around.

3.1 Dynamic Processes

As well as being passed in as parameters, functions can be sent over channels. So we can construct "dynamic" versions of the above processes, where the functions can be changed while the process is running:

```
mapDynP :: (a -> b) -> Chanin (a -> b) ->
           Chanin a -> Chanout b -> CHP ()
mapDynP origF finput input output = mapDynP' origF
  where
    mapDynP' f
      = ((readChannel input >>= writeChannel output . f)
              >> mapDynP' f)
        <-> (readChannel finput >>= mapDynP')
```

This process chooses between receiving a value (and passing it on with the current function applied) or receiving a new function to replace the old one. (See appendix A for more discussion of the choice operator.) In this process the recursion is explicit rather than using the `forever` function, because the behaviour changes according to some persistent state; all the other processes seen so far have had repeating behaviour without any changing state.

A `filterDynP` process can be similarly constructed:

```
filterDynP :: (a -> a) -> Chanin (a -> Bool) ->
              Chanin a -> Chanout a -> CHP ()
filterDynP origF finput input output = filterDynP' origF
  where
    filterDynP' f
      =     (do x <- readChannel input
                when (f x) (writeChannel output x)
                filterDynP' f)
        <-> (readChannel finput >>= filterDynP')
```

This process chooses between reading a value from its input channel, and passing it on if it meets the current filtering criteria (embodied in the function `f`), and reading in a new filtering function. Such a process could be useful, for example, in a firewall system. The process would filter out blocked requests, but could be updated because of changes in the configuration while it is running.

3.2 Deducing Process Behaviour from Types

The free theorems [12] and associated ideas of parametricity allow properties of functions to be deduced solely from their type. For example, given the type `a -> a`, it is apparent that since this function must work for any type `a`, the only possible implementation is to return the argument given. A more complex example is the type

[1] It is possible to construct analogues of most of the standard list processing functions – but that is not the focus of this paper.

`(a -> b) -> [a] -> [b]`. From this time it can be reasoned that any values of type `b` that occur in the result list must be the result of applying the given function to an item in the input list. The result must be some collection of transformed original values, potentially filtered, duplicated or permuted – but done arbitrarily, without access to any information from the values themselves (since this must work for any types `a` and `b` and no functions are given to inspect them).

Similar reasoning can also be used with processes. Given the process with type `Chanin a -> Chanout a ->CHP ()`, we can reason about it similarly to the function with type `[a] -> [a]`. Any items that are sent out of the process must have previously been sent into the process. Items could be dropped, duplicated or permuted, but only arbitrarily.

We can recognise the types of some non-productive processes. A process with type `Chanin a -> Chanout b ->CHP ()` is guaranteed to never produce any output (in a similar way to the function `a -> b` being unimplementable). A process with a type such as `Chanin a -> Chanin b -> Chanout b ->CHP ()` may or may not consume values from its first input channel, and the mere fact of receiving them provides information that may affect the behaviour of the values being passed through on the other channel.

4. Wiring

In message-passing systems with typed channels, composing together processes using channels is a substantial part of the programming model. For example, we may want to compose together the map and filter processes described earlier in the paper 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
       filterP (> 0) input w <|*|> mapP show r output
```

This is shown diagrammatically in figure 1. 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 any knowledge required of its internally concurrent implementation.

4.1 Simple Operator

This composition of 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 = do (mw, mr) <- newChannelWR
                  p r mw <|*|> q mr w
```

This operator works for the example discussed above. However, we do not 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. We need a more general operator than the one above.

4.2 Richer Operator

Figure 2 shows another example of process composition, requiring different connections than figure 1. With the types of the channels needed to compose the processes so apparent, it should be just as easy to compose these processes as those discussed previously.

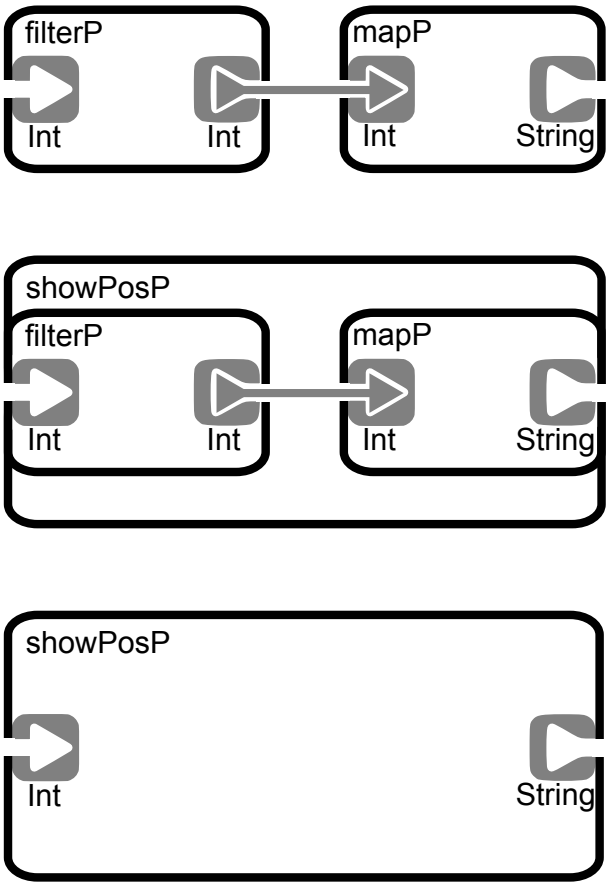

Figure 1. The composition of filter and map, shown explicitly in the top diagram. This composition becomes an opaque box to other components, as shown in the lower diagrams. This component can then be further composed; the programming model used in CHP is compositional in this way, allowing complex networks to be built from joining together different components without regard to their internal implementation.

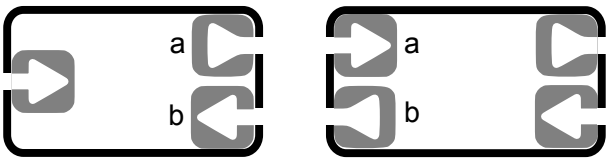

Figure 2. An example of slightly different process composition than figure 1. The letters indicate the types of the channel-ends that each process takes. It is readily apparent both that these processes *can* be composed, and *how* they should be composed (with a pair of channels).

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[2]:

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

instance Connectable (Chanin a) (Chanout a) where
  connect p = (swap <$> newChannelWR) >>= p
    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 more 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 p and q can easily be wired together if their types were:

```
p :: (Chanin Int, EnrolledBarrier ) -> CHP ()
q :: (Chanout Int, EnrolledBarrier ) -> CHP ()
```

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.

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) -> 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). 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 ends of a process pipeline:

[2] The <$> operator is a synonym for fmap and can be thought of for the purposes of this paper as having the type (a -> b) -> CHP a ->CHP b – it applies a pure function on the left-hand side to the return value of a monadic action on the right-hand side.

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

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

One thing to note is that if you have a beginning process, several middle processes and an end process to form a pipeline, it is natural to want to write:

```
begin |<=> middleA <=> middleB <=> middleC <=>| end
```

No associativity/precedence settings on those operators can possibly make the above type-check. The composition of `begin |<=> middleA` is a beginning process that cannot be composed with <=>, nor (if all the middle processes were composed with the tightest binding) with <=>|. Instead, we provide functions similar to those defined in the next section to support such a pattern.

4.3 Capture Common Topologies

We do not always want to simply compose two adjacent processes. Another common case 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:

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

We can easily extend this to a cycle:

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

Both topologies are depicted in figure 3.

This idea of capturing topology extends beyond such one-dimensional structures. A common requirement when building simulations 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 potentially error-prone. Without the connectable interface, it would have to be replicated for each type of channel used, increasing the possibility for error. 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 the type here but omit the lengthy definition[3]:

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

The parameter is a list of rows of processes (which must be rectangular); the result is a 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.

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

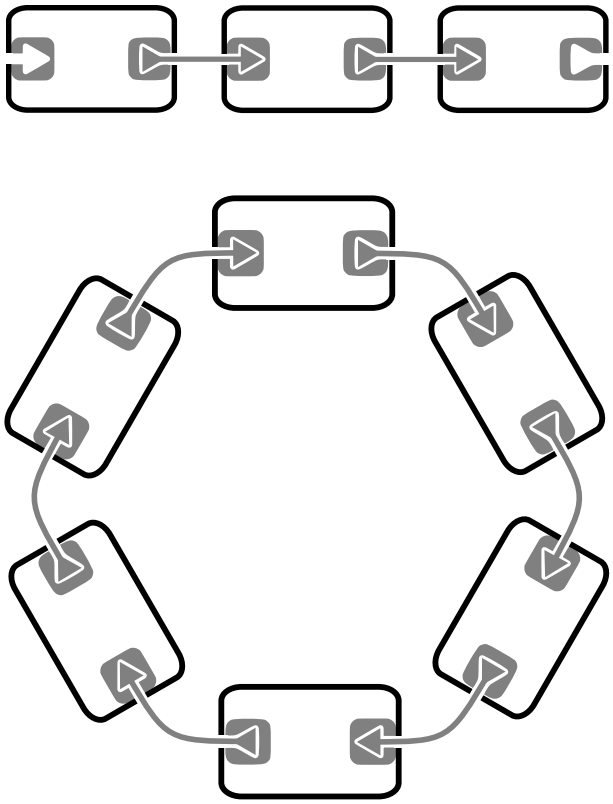

Figure 3. The pipeline topology (top) and cycle topology (bottom). It can be seen that a cycle can be formed simply by connecting the two end points of a pipeline together. The processes are illustrated here by connecting them with a single channel, but any regular interface could be connected together using the Connectable class described in this paper.

5. Compositional Wiring

The previous section 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 pipeline of processes may all be connected to their neighbours with a channel – but they may also all be enrolled together on a barrier.

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

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

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

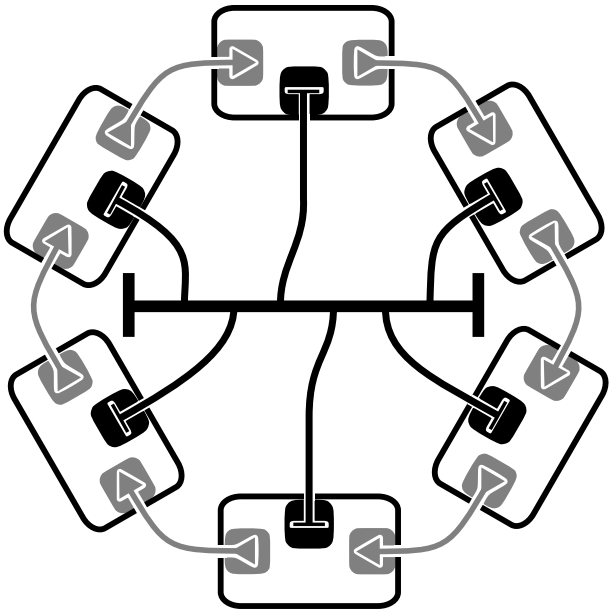

Figure 4. A ring of processes that are also all enrolled on the same central barrier.

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 CHP functions. We create functions like this (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]
```

Given a list of `processes :: [ EnrolledBarrier -> Chanin a -> Chanout a -> CHP ()`, we can compose them, as depicted in figure 4, 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.

The type of the Composed monad is:

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

This type is not without precedence as a monad; it is equivalent to `forall b. ContT b CHP a`, the continuation-passing monad transformer on top of CHP, 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
```

The output of any Composed block is almost always such a list of complete CHP processes ready to be run in parallel.

The monad instance for Composed is as follows:

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

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

The latter can easily be expanded into an enrollAllR function:

```
enrollAllR :: Enrollable b p => b p -> [Enrolled b p -> a]
                -> Composed [a]
enrollAllR b ps = mapM (enrollR b) ps
```

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 the paper:

```
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 definition trivial based on the cycleR function, with the help of the standard list function transpose that swaps rows for columns in a list of lists. (The <=< function composes two monadic functions; its type is Monad m =>(b ->m c) -> (a -> m b) -> a -> m c)

It is possible for users to define their own wiring functions using this monad. For example, a user may find that frequently in their program they have a list of processes where they wish to enroll all the processes at odd positions in the list on one barrier, and all the processes at even positions on another barrier. They could write a function to do this, and use it 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.

6. Behaviours

A common pattern in simulations built with CHP is the tick pattern. Each process enrolls on a barrier named tick that has a low priority. The tick barrier divides time into timesteps (tick occurs on the boundary between two timesteps). Each process offers a choice between performing various actions with other processes, and performing the tick event. Since the tick event has the lowest priority it will only performed when no other choices by any process can be.

Consider a process representing a physical site in a simulation. It may be currently holding an agent. It will allow another agent to enter the space (a maximum of one per time-step) and it will independently allow its current agent to move on. So it has three actions; moveIn (which may happen once or not at all), moveOut (which may happen once or not at all) and tick (which will happen once, and will end this current behaviour). Afterwards we want a list of the agents now in the site.

We must implement this functionality as follows:

```
start cur = (moveIn >>= movedIn cur)
          <-> (moveOut >> movedOut)
          <-> (tick >> return [cur])

movedIn old new = (moveOut >> tick >> return [new])
              <-> (tick >> return [old, new])

movedOut = (moveIn >>= \new -> tick >> return [new])
         <-> (tick >> return [])
```

There are 5 different paths through this functionality; each move may or may not occur (and if they both occur, they may occur in either order), followed by tick. If we added a third movement option, there would be 16 different paths and the number would grow exponentially.

It is a poor abstraction to program them using the above mechanism. We instead want an API that will allow us to clearly and concisely express our intention: we wish to perform two things once or not at all, ended at any time by a third thing, and we want to know afterwards what happened.

6.1 Behaviour API

Our solution to this is called behaviours. A behaviour is an item of type CHPBehaviour a[4], and represents some action (potentially repeated) that will result in a value of type a. Such a behaviour can be executed using the offer function:

```
offer :: CHPBehaviour a -> CHP a
```

We allow combination of behaviours (iterated choice) using the alongside combinator:

```
alongside :: CHPBehaviour a ->
             CHPBehaviour b ->
             CHPBehaviour (a, b)
```

Its type aside, alongside is semantically associative and commutative. We define two fundamental behaviour types; those which end the current offer, and those which do not. The former uses endWhen, the latter encompasses all the other functions:

[4] CHPBehaviour has a Functor instance for modifying the return value of the item, but is not a monad, nor an applicative functor.

```
endWhen :: CHP a -> CHPBehaviour (Maybe a)

once :: CHP a -> CHPBehaviour (Maybe a)

upTo :: Int -> CHP a -> CHPBehaviour [a]

repeatedly :: CHP a -> CHPBehaviour [a]

repeatedlyRecurse :: (a -> CHP (b, a)) -> a ->
                     CHPBehaviour [b]
```

The Maybe return of endWhen indicates that it may or may not have occurred. This may seem odd, but it is possible to have two or more endWhen functions combined with alongside, and then precisely one will have occurred (with a Just return) and the others will not have (a Nothing return).

The latter four functions can all be thought of as special cases of one another; once is equivalent to listToMaybe . upTo 1, while repeatedly is upTo without an upper bound, and repeatedlyRecurse offers persistence of state through repeated executions of the action.

Some laws for simple behaviours naturally follow from the definitions of these behaviours:

```
offer (repeatedly p) = forever p

offer (once p) = p >> stop -- i.e. it does not finish

offer (endWhen q) = Just <$> q

offer (endWhen p `alongside` endWhen q)
  = Just <$> (p <-> q)

offer (once p `alongside` endWhen q)
  = (p >>= \x -> q >> return x)
    <-> (q >> return Nothing)
```

Most further examples have no such simple rearrangement, which is of course the incentive to express them as behaviours.

Our earlier example can now be expressed with behaviours:

```
do ((old, new),_) <- offer ((once moveIn
                               `alongside` once moveOut)
                              `alongside` tick)
```

6.2 Implementation

The CHPBehaviour type is made up of a value that would be returned if the offer were terminated now, and a possible choice to offer as part of the behaviour; if the latter part is Nothing, this indicates that offer should terminate now:

```
data CHPBehaviour a
  = CHPBehaviour a (Maybe (CHP (CHPBehaviour a)))

instance Functor CHPBehaviour where
  fmap f (CHPBehaviour x m)
    = CHPBehaviour (f x) (fmap (fmap (fmap f))) m)
```

The alongside combinator is implemented using CHP's choice operator, and whichever option occurs, it is joined together with the other half of the alongside call again for the next choice:

```
alongside oa@(CHPBehaviour a (Just fa))
          ob@(CHPBehaviour b (Just fb))
  = CHPBehaviour (a, b)
      (Just $ (flip alongside ob <$> fa)
              <-> (alongside oa <$> fb))

alongside (CHPBehaviour a _) (CHPBehaviour b _)
  = CHPBehaviour (a, b) Nothing
```

The repeatedly combinator accumulates a list (adding at the head for efficiency):

```
repeatedly m = reverse <$> repeatedly' []
  where
    repeatedly' xs
      = CHPBehaviour xs
          (Just ((repeatedly' . (: xs)) <$> m))
```

The once combinator will execute its action once, and afterwards will only offer stop, the choice that can never be taken, thus preventing it occurring a second time, without terminating the call to offer:

```
once = CHPBehaviour Nothing . Just .
         fmap (flip CHPBehaviour (Just stop) . Just)
```

The endWhen combinator will become Nothing after its execution, thus terminating the call to offer:

```
endWhen = CHPBehaviour Nothing . Just .
            fmap (flip CHPBehaviour Nothing . Just)
```

Finally, the offer call itself simply repeats the behaviour until a Nothing value is found for the action:

```
offer (CHPBehaviour _ (Just m)) = m >>= offer
offer (CHPBehaviour x Nothing) = return x
```

6.3 Relation to Grammars

The way that we originally explored programming the behaviour of our system at the start of section 6 is a context-free grammar (CFG), and the program code is similar to how such a system might be encoded in a form such as Backus-Naur Form (BNF). The grammar is a grammar over the sequence of actions that the particular process might take – but it is more like a generative grammar than a parser.

CFGs are generally used for certain sequences of actions, e.g. one a followed by many bs followed by a c. What we require here, and what CFGs and notations such as BNF are ill-suited to capture, is patterns such as: many bs with at most one a amongst them. This must be captured using a BNF rule with two states, one where a has not happened yet and one where a has. Trying to make this more complex, e.g. many bs with at most two a and three c anywhere among them, does not scale when using this finite-state automata-like approach.

Our Haskell solution to the problem does not correspond to a CFG – its use of higher-level combinators takes it further than a CFG. There do exist grammar systems that can capture the same as our Haskell solution – particularly two-level grammars [5]. Intuitively, a two-level grammar is a grammar that generates a grammar; a higher-order grammar, if you will.

We begin representing our system in a two-level grammar by defining hyper-rules that capture the behaviour of our combinators – and like our combinators, they can remain constant for all the uses of behaviours (they only need be defined once):

```
g: g, END.
g: EMPTY; SEQ check.

OPTIONAL SEQ check: OPTIONAL symbol, SEQ check,
                    OPTIONAL notin SEQ.
REPEATABLE SEQ check: REPEATABLE symbol, SEQ check.
EVENT check: EVENT symbol.
```

In these hyper-rules, colon introduces a definition, comma is sequence and semi-colon is choice or branching. The first grammar, g, is the grammar g' followed by the END terminal. The grammar g' can either be empty or the meta-production SEQ followed by

`check` – the latter item is a post-fix tag used to match with the latter three hyper-rules (in effect, it invokes these hyper-rules by needing to be parsed).

The first hyper-rule matches an `OPTIONAL` meta-production at the front of a sequence, with a final `check` tag. This must be an `OPTIONAL` meta-production followed by a further sequence (also with a post-fix `check` tag), where the optional item is not in that sequence. That is, this rule matches an optional item before a sequence and ensures that the optional item does not occur again in the following sequence. The `notin` notation is itself a rule that can be constructed that only parses if the left-hand side does not appear in the right-hand side [5].

The second hyper-rule matches a `REPEATABLE` meta-production at the front of a sequence. It is simply a repeatable symbol followed by a further sequence with a `check` tag.

The third hyper-rule is for a sequence with a single event (the base case, in effect) that matches just that event.

These hyper-rules use the following constant meta-productions (like the hyper-rules, they only need be defined once):

```
EVENT :: END; OPTIONAL; REPEATABLE.
EMPTY :: .
SEQ :: EVENT ; SEQ EVENT.
```

`EMPTY` is the empty sequence. `SEQ` is any sequence of `EVENT`. `EVENT` is any event of the three classes of events:

- `END`, the events that end the behaviour,
- `OPTIONAL`, the events that can happen at most once, and
- `REPEATABLE`, those events which may occur any number of times.

These latter three sets of events are the only things that need be altered to adapt this two-level grammar system to the particular behaviour at hand, e.g.:

```
END :: tick.
REPEATABLE :: .
OPTIONAL :: moveIn; moveOut.
```

7. Related Work

Several other message-passing libaries exist in functional programming languages besides CHP. CML is the most obvious precursor [10], and it has since been converted to Haskell, too [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 CML.

Erlang is a functional programming language with a strong message-passing component [1]. However, Erlang uses addressed asynchronous messages (sent to a particular address) rather than channel-based synchronous messages. 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 described in this paper are not common in Erlang.

Lava is a hardware design domain-specific language embedded in Haskell [2]. Lava featured operators to compose together digital circuit components into new 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.

8. Conclusions

The Communicating Haskell Processes library is an imperative message-passing library built in a functional programming language with a clean, simple API. 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.

The mapP and filterP processes transfer long-standing functional programming ideas directly into message-passing programming. The dynamic versions of these processes demonstrate how processes can be dynamic and with changing behaviour even when all data is immutable.

CHP programs are made up of many components composed together concurrently. The long-hand way of composing these processes, by manually declaring channels and passing the ends to the right processes, is tedious and error-prone. The combinators discussed in this paper allow for a more point-free style, composing processes together without ever naming the channels.

The Connectable type-class allows the wiring functions to abstract away from the mechanisms used to compose adjacent 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.

The work on behaviours shows that the most straightforward way to program some processes using just the imperative monad can end up intricate and unclear. But the combinator-based higher-order approach can again be used to provide a simpler API to capture these repeated behaviours more clearly.

8.1 Language Extensions

This work does not introduce any new language extensions for Haskell, but it does use a few pre-existing extensions to the Haskell 98 and Haskell 2010 standards (as much modern Haskell development does). These are:

- Multi-parameter type-classes, which are required for the Connectable class – however, further contentious extensions such as functional dependencies and/or type families are *not* required;
- Flexible instances, which are required for the Connectable instances; and
- Rank-2 types, which are required for the definition of the Composed monad.

A. Choice of Leading Actions

Choice in CHP is between leading actions of a given code block. So for example, in this code:

```
(readChannel inputA >>= writeChannel outputA)
<-> (readChannel inputB >>= writeChannel outputB)
```

The process waits until it is able to input from channel inputA or channel inputB. Once this choice has been made and the input successfully made, it goes on to write the value to channel outputA or outputB respectively. So although the choice operator takes both blocks as an argument, it only chooses between the leading actions.

If this was not permitted, and choice was only allowed between single actions, we would have to code the process more awkwardly. The Either sum type in Haskell provides a convenient solution for choices between two items:

```
((Left  <$> readChannel inputA)
  <-> (Right <$> readChannel inputB)) >>=
  either  (writeChannel outputA) (writeChannel outputB)
```

However, for items with three choices the approach begins to get more awkward, requiring nested sum types or unnecessary definitions of extra types:

```
data ChoiceResult a b c = A a | B b | C c

do x <- alt  [A <$> readChannel inputA
             ,B <$> readChannel inputB
             ,C <$> readChannel inputC])
   case x of
     A y -> writeChannel ouputA y
     B y -> writeChannel ouputB y
     C y -> writeChannel ouputC y
```

Compare the latter to:

```
alt  [readChannel inputA >>= writeChannel outputA
     ,readChannel inputB >>= writeChannel outputB
     ,readChannel inputC >>= writeChannel outputC]
```

The intention and behaviour of the code is actually clearer in the latter case by using this leading action rule. This design choice is also entirely consistent with the Communicating Sequential Processes calculus [6, 11] on which the library is based.

References

[1] J. Armstrong, R. Virding, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1993.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 174–184, New York, NY, USA, 1998. ACM. ISBN 1-58113-024-4. doi: http://doi.acm.org/10.1145/289423.289440.

[3] N. C. C. Brown. Communicating Haskell Processes: Composable explicit concurrency using monads. In *Communicating Process Architectures 2008*, pages 67–83, Sept. 2008.

[4] A. Chaudhuri. A concurrent ML library in concurrent Haskell. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 269–280, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: http://doi.acm.org/10.1145/1596550.1596589.

[5] J. C. Cleaveland and R. C. Uzgalis. *Grammars for Programming Languages*. Elsevier, 1977.

[6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. URL `http://www.usingcsp.com/`.

[7] J. Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37 (1-3):67–111, 2000. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/S0167-6423(99)00023-4.

[8] C. Mcbride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796807006326.

[9] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7. doi: http://doi.acm.org/10.1145/158511.158524.

[10] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[11] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997. URL `http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf`.

[12] P. Wadler. Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0. doi: http://doi.acm.org/10.1145/99370.99404.