

Communicating Haskell Processes

Neil C. C. Brown

University of Kent
neil@twistedsquare.com

Abstract

Interactive programs, such as webservers or GUI-based programs, typically need to be concurrent to be responsive and effective. Concurrency requires extra code to be added to programs to coordinate the interactions between threads, and in standard imperative languages new errors can occur with unintentionally shared mutable data. In this paper we demonstrate the use of the Communicating Haskell Processes concurrent message-passing library for implementing a concurrent game/simulation, and provide a formal operational semantics for the library. We show that the library can support a rich set of message-passing behaviours while remaining concise. These benefits are primarily realised because of the support and features provided by the Haskell language.

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

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Many programs – such as GUIs, networked servers, or games – need to interact with the outside world, dealing with events that may occur at any time, and where multiple events may need to be handled concurrently; for example, a networked server must be able to handle multiple requests from clients simultaneously. Concurrency is a way to write programs as multiple executing entities that can interact with each other (in contrast to parallelism, which focuses on exploiting parallel hardware for performance gains).

Concurrency is traditionally implemented in imperative languages such as C or Java. These languages do not have the type-safety, expressiveness and other benefits of functional languages such as Haskell – and they additionally have problems with race hazards arising from mutation of shared state. Functional programming eliminates this latter worry by eliminating shared mutable state by default. Coupled with its support for imperative programming via monads, this makes Haskell an attractive host language for writing concurrent programs.

Communicating Haskell Processes (CHP) is a software library for Haskell. At its core are basic primitives for concurrent composition and channel communication. It features support for choice be-

tween communications and uses a “poison” mechanism to facilitate concurrent shutdown. Around this it builds a rich set of combinators for facilitating easy composition of processes. An early version of the core of CHP was previously published [1], containing concurrency, channel communications and an early version of barriers and poison but lacking a formal semantics; in this paper:

- We explicate the updated design of CHP, providing arguments for why it is a valid design alternative to some other Haskell concurrency mechanisms.
- We demonstrate how CHP can be used to create a simple concurrent game/simulation, explaining how the design of CHP reduces the possibility for programmer error and shortens the program.
- We formalise the behaviour of CHP by providing an operational semantics, and provide details of its implementation.

In this paper we first set the scene for the example that we will build up during the paper (section 2) before introducing the CHP library (section 3) and introducing its features in turn: simple parallel composition (section 4), channel communication (section 5), choice between actions (section 6), phased barriers (section 7), poison for concurrent termination (section 8), mobile processes (section 9) and advanced process composition (section 10) – building up the example with new processes as we introduce the necessary features. We give details of the operational semantics (section 11) and implementation (section 12) of CHP, before discussing related work (section 13) and offering some conclusions (section 14).

2. Central Example

Throughout this paper, we will build up a concurrent game example. The game mechanics are simple (as they are not the focus of the example): the game has a rectangular grid of tiles. Each tile may contain one item of food and/or one player. Players aim to collect as much food as possible – the game contains one human player and several computer-controlled players. The human player is controlled by keypresses and the state of the game is displayed in a graphics window. Our focus in this paper is on concurrency rather than parallel performance, but for interest we note that the performance is sufficient to run the final example on a typical desktop without speed problems.

Many of the processes in our final system are straightforward: we will have a process responsible for getting events (such as keypresses) and sending them on, and a process responsible for querying the state of the system and drawing this on the screen. We choose to represent the rectangular grid with an active process per tile, connected to its four neighbours by channels. Players are then held by a particular tile, which represents their location. For reference, our final process network is illustrated in figure 1.

3. Communicating Haskell Processes

Communicating Haskell Processes (CHP) is a library for Haskell. In theory, it is an imperative monad-based library that supports concurrent composition of message-passing processes. In practice, it is a thin layer on top of the IO monad, with full access to IO actions from its central CHP monad:

```
data CHP a
```

```
liftIO_CHP :: IO a -> CHP a
```

```
runCHP :: CHP a -> IO a
```

Despite their dual types, runCHP and liftIO_CHP are for different purposes: a CHP program should be enclosed in a single runCHP call at the outermost level (typically in the main function), with liftIO_CHP calls scattered throughout.

4. Parallel Composition

A key component of a concurrency library is concurrent composition; the parallel function runs the list of processes in parallel with each other, waits for them all to finish and then returns the list of results in corresponding order:

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

In CHP, processes are not given identifiers, and there is never a need to identify a particular process. As a simple example, the parallel function can be used in combination with liftIO_CHP to read in the contents of two files in parallel (assuming the entire file is read at once, without lazy IO):

```
parallel (map (liftIO_CHP . readWholeFile) [a, b])
```

5. Communication Channels

Concurrently composed processes that do not interact with each other – like those in the previous section – are generally uninteresting. In a concurrent program, processes typically need to interact with each other. In CHP, one of the key ways for processes to interact is via communication channels. These channels are:

- synchronised: there is no buffering on any of the channels (if required, buffering can be implemented using processes);
- anonymous: channels are created, and the ends passed to the relevant processes, but there is no notion of addressing or common mailboxes – processes do not know where the other end of the channel is;
- either point-to-point: these channels are between one writer and one reader – these ends may be shared but they are always used by only one writer and one reader at a time,
- or multi-way: broadcast channels allow multiple readers in a communication, while reduce channels allow multiple writers (we will not discuss these channels in this paper; for a full treatment see Brown [3]).

Rather than having a single Channel a type for channels, we distinguish the reading end Chanin a (from which values can be *input*) and writing end Chanout a (on which values can be *output*) at the type-level. This aids in the readability of a program – making it immediately apparent whether a process will be writing or reading on a channel – and aids safety, by preventing the accidental connection of two processes which both intend to only read from a channel.

The basic primitives for reading and writing on channels are straightforward:

```
writeChannel :: Chanout a -> a -> CHP ()
```

```
readChannel :: Chanin a -> CHP a
```

A channel can be created and its two ends returned using:

```
newChannelRW :: CHP (Chanin a, Chanout a)
```

A simple example that reads one line from stdin in one process, and sends it to another which prints on stdout, is:

```
do (r, w) <- newChannelRW
    parallel [ liftIO_CHP getLine >>= writeChannel w
              , readChannel r >>= liftIO_CHP . putStrLn ]
```

5.1 Distributed Binding

Channels can be thought of as a distributed binding. Consider this code:

```
newChannelRW >>= \ (r, w) ->
parallel [ (do someA
           y <- readChannel r
           process y) , (do x <- someB
                           writeChannel w x
                           someOtherB) ]
```

This runs two processes in parallel, with a channel communication between them on the second line of each. The channel communication makes these processes act as if the binding from the readChannel came directly from the value passed to the corresponding writeChannel call (with the two processes synchronising at that point):

```
parallel [ (do someA
           y <- x
           process y) , (do x <- someB
                           someOtherB) ]
```

5.2 Shared Channels

Shared channels are very simple; they allow one or both ends of a channel to be wrapped with a mutex which must be claimed before the channel can be used. The API is a minimal addition to the channel API:

```
data Shared c a = Shared Mutex (c a)
```

```
claim :: Shared c a -> (c a -> CHP b) -> CHP b
```

The c item is either Chanin or Chanout. The claim function claims the channel end for the duration of the given action, automatically releasing it afterwards (including in the case of exceptions, or poison which is introduced later on).

5.3 Wrapping Channels

It is possible to wrap a channel end to apply a pure function on the value before it is sent or after it is received. For Chanin this can be a Functor instance, but Chanout is instead a co-functor (the pure function has reversed types) so in the absence of a popular type-class we have individual functions:

```
instance Functor Chanin
```

```
instance Functor (Shared Chanin)
```

```
mapChanout :: (b -> a) -> Chanout a -> Chanout b
```

```
mapSharedChanout :: (b -> a) -> Shared Chanout a
                  -> Shared Chanout b
```

5.4 Example: Event Process

We can build a simple process that waits for events from an interface and sends them out on a channel. We first posit the existence of these types and functions:

```
data Key
```

```
data Event = UpEvent Key | DownEvent Key | QuitEvent
```

```
getNextEvent :: IO Event
```

The implementation of Key and getNextEvent are not relevant here; they can be built on top of any suitable graphical interface library. With this we can create a process:

```
eventSender :: Chanout Event -> CHP ()
eventSender output
  = forever (liftIO_CHP getNextEvent >>= writeChannel output)
```

This process is very simplistic, but it illustrates the common form of most CHP processes. It takes some channels as its parameters and then has a simple repeated behaviour. We can adapt this to form a more useful process that takes a list of keybindings – a map from keys (on the keyboard) to actions, or specifically in our case, a map from keys to channels. A GADT allows us to be polymorphic over the type being sent on the channels:

```
data KeyBinding where
  Bind :: Key -> KeyUD -> a -> Chanout a -> KeyBinding
```

```
data KeyUD = KeyUp | KeyDown
```

This indicates that whenever the given key enters the given state, the given value (of type a) should be sent on the given channel. Thus we can improve our event process:

```
eventHandler' :: [KeyBinding] -> CHP ()
eventHandler' bindings = forever $
  liftIO_CHP getNextEvent >>= \e -> case e of
    UpEvent k -> sequence_ [writeChannel c x
      | Bind k' KeyUp x c <- bindings, k == k']
    DownEvent k -> sequence_ [writeChannel c x
      | Bind k' KeyDown x c <- bindings, k == k']
  -> return ()
```

We will return to the QuitEvent later on in section 8.3, but for now this process repeatedly waits for the next event, and if it is a key event then the appropriate message is sent out to all the interested listeners.

6. Choice

CHP supports choice between actions using an instance of the standard Alternative type-class:

```
class Alternative f where
  (<|>) :: f a -> f a -> f a
  empty :: f a
```

```
instance Alternative CHP
```

For example, readChannel c <|> writeChannel d waits until it can read from either channel c or write on channel d, and then it performs that *single* action; it will not, in the same choice, perform both communications. One notable feature of choice in CHP is what happens with choice and monadic bind:

```
(readChannel c >>= writeChannel e)
  <|> (readChannel d >>= writeChannel f)
```

This code waits to either read from channel c or channel d. After that choice has been made and the read performed, it continues and writes on channel e or f respectively. That is, it chooses between the left-most/first actions in a monadic block.

The alternative to this API would be to copy the classic CML design and separate events from what comes afterwards, that is to have something like:

```
read :: Chanin a -> Event a
write :: Chanout a -> a -> Event ()
choose :: [Event a] -> Event a
sync :: Event a -> IO a
```

```
instance Functor Event
```

We could then write the previous example as:

```
sync (choose [Left <$> read c, Right <$> read d])
  >>= either (sync . write e) (sync . write f)
```

In this code the overall behaviour is obscured by the addition of the tag types (and it would get more awkward with more than two types!), although the choice is made more explicit. In fact, most CML implementations in Haskell [4, 11] introduce some sort of wrap combinator:

```
wrap :: Event a -> (a -> IO b) -> Event b
```

This incorporates subsequent actions into the event, allowing the user to write:

```
sync $ choose [read c 'wrap' sync . write e
              , read d 'wrap' sync . write f]
```

CHP's design can therefore be seen as a CML variant which dispenses with sync (in favour of effectively using **type** CHP =Event throughout) and replaces wrap with the monadic bind operator (which allows easy use of Haskell monadic **do**-syntax).

6.1 Example: Input State Process

We can use choice to implement a process akin to a shared-state process or overwriting buffer. The process reads in key events that come from the eventHandler' process (defined in section 5.4), and uses the events to keep track of which direction the player should currently be heading (if any). We first define a useful helper function that is a slight modification of the standard forever function:

```
foreverFeed :: Monad m => (a -> m a) -> a -> m b
foreverFeed f x = f x >>= foreverFeed f
```

This repeatedly executes the monadic function, feeding the results into a further execution of the function. This should not be mistaken with mfix :: Monad m => (a -> m a) -> a, which executes the function only once and ties a recursive knot. With this we can define our process:

```
data Direction = DirUp | DirDown | DirLeft | DirRight
  deriving Eq
```

```
inputState' :: Chanin (Direction, KeyUD)
             -> Chanout (Maybe Direction) -> CHP ()
```

```
inputState' input output = foreverFeed go Nothing
  where
    go x = (do e <- readChannel input
             case snd e of
               KeyUp -> return $
                 if Just (fst e) == x then Nothing else x
               KeyDown -> return $ Just $ fst e
            ) <|> (writeChannel output x >> return x)
```

The central behaviour of this process is the inner go function. This chooses between reading from its input channel and writing to its output channel. In the latter case, it writes the current state, and then retains the same state for the next execution of the process. If it reads from its input channel, it checks to see if the key corresponding to the current direction has been released. If so, the new state is to have no heading (Nothing). If instead a key has been pressed down, that becomes the new heading (overwriting a previous heading in the case where the user is holding down two keys).

The output channel of this process can thus be read from at any time to find out which direction the player currently wants to head, without exposing any details of the keys or key-bindings. This modularity is good software engineering practice.

7. Phased Barriers

Communication channels allow a process to send information to another process. In a sense, two pieces of information are shared between the processes: the first is the value that is explicitly sent on the channel, and the second is the implicit knowledge that the processes are now at a particular stage in their execution.

Barriers are a synchronisation primitive that do not involve sending values between processes. They instead embody the synchronisation part of channels, allowing processes to determine that they are all at a common stage in their execution. For example, a common pattern in simulations is to have a discovery phase, in which agents may query each other for information, and a movement phase, in which agents may alter their state (e.g. move around in a simulated world). It is typically important for the consistency of the simulation that the agents all see a consistent world state in the discovery phase – this requires making sure that all the queries in this phase occur before the state-changes in the movement phase.

A barrier can be used to separate these two phases. In CHP we define a barrier as follows; barriers:

- have a notion of enrollment (or membership): a barrier has an associated count of processes enrolled on it;
- allow dynamic enrollment and resignation: processes may enroll on the barrier (increasing its membership count) or resign (decreasing its membership count) at any time – a barrier does *not* have a fixed membership count for its lifetime;
- are synchronous – when a process wants to synchronise on a barrier, it must wait for all processes (as many as are currently enrolled) to synchronise;
- are re-useable – barriers may be used repeatedly (in some synchronisation libraries, barriers are single-use);
- permit choice between barrier synchronisations (and between channel communications).

The idea behind phased barriers is that barriers:

- have the notion of a phase: a phase is a shared piece of information that deterministically alters with each successive synchronisation on a barrier.

The core of the phased barrier API is as follows:

```
newPhasedBarrier' :: [phase]
                -> CHP (Unenrolled PhasedBarrier phase)
syncBarrier    :: PhasedBarrier phase -> CHP phase
enroll        :: Unenrolled PhasedBarrier phase
                -> (PhasedBarrier phase -> CHP a) -> CHP a
```

The `syncBarrier` call synchronises on a barrier and returns the new current phase of the barrier. The `newPhasedBarrier'` call takes a list of phases, which are cycled into an infinite list of phases: the head of the infinite list is the first phase, and after the first synchronisation the phase is the second item in the list and so on. The barrier creation call returns an unenrolled barrier; processes must then `enroll` on the barrier. The `enroll` function takes an unenrolled barrier (first parameter), an action to execute while enrolled on the barrier (second parameter), and then: enrolls the process, then performs the action and finally resigns, returning the result of the inner action. Using this style of API, rather than an `enroll` and `resign` call, prevents the programmer mistakenly forgetting the `resign` call.

We can design a helper function on top of `newPhasedBarrier'` that uses the Haskell type-classes `Enum` and `Bounded` to automatically generate the list of phases:

```
newPhasedBarrier :: (Bounded phase, Enum phase) -> phase
                -> CHP (Unenrolled PhasedBarrier phase)
newPhasedBarrier start = newPhasedBarrier'
  ([start .. maxBound] ++ cycle [minBound .. maxBound])
```

This starts at the given phase and then cycles through all phases in order. One can easily create a barrier with the aforementioned discovery and movement phases:

```
data Phase = Discovery | Movement deriving (Enum, Bounded)
```

```
newPhasedBarrier Discovery
```

If a program needs a barrier without phases, it can use the unit type `()` for phases, which is an element of both `Enum` and `Bounded`:

```
newBarrier = newPhasedBarrier ()
```

Sometimes there are processes which are only interested in acting on a particular phase of the barrier. For this situation we provide a helper function that synchronises repeatedly until the particular phase is reached:

```
syncAndWaitForPhase :: Eq phase => phase
                  -> PhasedBarrier phase -> CHP ()
syncAndWaitForPhase target bar
  = do ph <- syncBarrier bar
      when (ph /= target) $ syncAndWaitForPhase target bar
```

7.1 Example: Frame Barrier

In our game example, we will use a barrier to synchronise between consecutive frames, and also to separate the player movement phase from the screen update phase. This means our drawing process waits for the screen update phase, then reads in updates from all the tile processes who have changed their state, draws this on the screen and repeats.

This description belies some complexity: the drawing process does not know how many processes will have changed their state, so it is not clear how it can know when it has received all the updates. We use some simple logic: all the tile processes are enrolled on the central barrier, and all will send their update before offering to synchronise. Therefore the drawing process can choose between synchronising on the barrier (which can only complete when all the tiles have finished sending their updates) or receiving another update. For the drawing process we assume that the following functions and types exist:

```
data TileDraw = Draw { drawFood :: Bool, drawPlayer :: Bool }
drawOnScreen :: Array (Int, Int) TileDraw -> IO ()
```

We can then define our drawing process:

```
draw' :: Chanin (Pos, TileDraw) -> FrameBarrier -> CHP ()
draw' input frame = foreverFeed go $
  listArray ((0,0),(size-1,size-1)) (repeat (Draw False False))
  where
    go s = do syncAndWaitForPhase UpdatePhase frame
              s' <- (s //) <$> getUpdates
              liftIO_CHP $ drawOnScreen s'
              return s'

    getUpdates = ( syncBarrier frame >> return [] ) <|>
                ((:) <$> readChannel input <*> getUpdates)
```

The `getUpdates` function implements the previously described behaviour, reading from the input channel and adding to a list of updates until the barrier synchronisation completes. The updates are then applied to the state (which is held in an array) which is drawn on screen.

All the processes that we have seen so far execute as fast as they can; each frame will be processed as fast as possible, which may prove too fast (or too uneven) for a human player. Delaying frames so that they occur at roughly regular intervals is a simple matter of adding another process to the barrier synchronisation that only synchronises at a given rate. This automatically slows down all the other processes to the right rate, without ever altering any of their code. The delaying process is very simple:

```
frameLimiter :: Eq phase => Int -> phase
              -> PhasedBarrier phase -> CHP ()
frameLimiter fps ph bar = forever $ do
  waitFor (1000000 `div` fps)
  syncAndWaitForPhase ph bar
```

The `waitFor` function is part of the CHP library, it takes a delay in microseconds and waits for that length of time to elapse.

8. Poison: Concurrent Termination

One issue with concurrent programs is that of termination. With many concurrent processes running it is not clear how to coordinate the graceful termination of the process network. Sending messages is very difficult to get right without provoking a deadlock [13], and is not sufficient – for example, we could not terminate our earlier `eventHandler` process that only has an output channel.

There are several termination methods already available to Haskell programmers. The `killThread` function allows an identified thread to be killed via an asynchronous exception [7]. Asynchronous exceptions require reasoning about which operations are interruptible, and when threads are and are not masked from exceptions – it is not immediately clear when a thread may or may not receive an asynchronous exception. Additionally, CHP does not support the identification of threads¹ as it does not fit with the programming model of anonymous processes.

Another mechanism for terminating concurrent Haskell programs are indefinitely-blocked exceptions. Intuitively, if a thread is blocked on a synchronisation primitive (such as an `MVar` or `TVar`) and that thread holds the only reference to that primitive, it can never be woken up. Therefore, if during the garbage collection such a situation is discovered, the thread is woken up with an indefinitely-blocked exception. This mechanism is clearer in one sense than the asynchronous exceptions, because the points at which an exception can be thrown (only `MVar` and `TVar` operations) are clear. However, this mechanism is fragile: if another thread holds a reference to the `MVar/TVar` but will not use it to communicate then the exception will not be thrown. Additionally, it cannot be used to detect partial deadlock, wherein some processes have terminated but others can continue. For example, a merging process that chooses between reading from several `TVars` will not be given an indefinitely-blocked exception as long as at least one of the `TVars` it is reading from is still referred to by another thread.

For terminating the process network, CHP introduces the idea of poison. Channels and barriers can be put into a poisoned state. All processes waiting on a channel or barrier when it is poisoned by another party are immediately woken up and have a poison exception thrown (this is not an exception thrown with `throw` in IO, but conceptually it is an exception). All future attempts to use the channel or barrier will result in a poison exception being thrown. The key in using poison is that once a process encounters poison it should poison all the channels and barriers that it is using, then tidy up any open resources and terminate. This means that the neighbouring processes (in the process network) will encounter poison and do the same (repeated poisonings of a channel or barrier have no further effect). If the process graph is fully connected (as it usually should be in a CHP program) then poison will spread throughout and the process network will shut down.

The API for poison is as follows:

```
throwPoison :: CHP a
onPoisonRethrow :: CHP a -> CHP b -> CHP a
```

```
class Poisonable c where
  poison :: c -> CHP ()
```

```
instance Poisonable (Chanin a)
instance Poisonable (Chanout a)
instance Poisonable (PhasedBarrier phase)
```

The full semantics are given in section 11.1, but we give some informal laws that provide the intuition:

```
throwPoison >>= m = throwPoison
throwPoison 'onPoisonRethrow' m = m >> throwPoison
readChannel c = throwPoison -- iff c is poisoned
```

¹ Although it can be done, using: `liftIO_CHP myThreadId`.

The semantics of parallel composition and poison are: the processes are run in parallel until all have completed (either by throwing poison or completing normally). After they have all completed, if any of the processes completed by throwing poison, the parent `parallel` call also throws poison. Thus poison is propagated up the hierarchy of processes to the topmost level.

8.1 Using Poison

In general, the form for using poison is as follows. We start off with the simple identity process that forwards values from one channel to another:

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

To add support for poison, this should become:

```
idProcess :: Chanin a -> Chanout a -> CHP ()
idProcess input output
  = forever (readChannel input >>= writeChannel output)
    'onPoisonRethrow' (poison input >> poison output)
```

Adding support for poison is very formulaic: a process should handle poison at its outermost level by poisoning all its channel and barrier parameters and then rethrowing the poison.

8.2 Automatic Poison

Since adding support poison is very formulaic, it can be automated. We do this by wrapping a process with a call to an `autoPoison` function, so that the previous example can be written as:

```
idProcess, idProcess' :: Chanin a -> Chanout a -> CHP ()
idProcess = autoPoison idProcess'
idProcess' input output
  = forever (readChannel input >>= writeChannel output)
```

This is a simple idiom that moves all the poison code into a single simple line. The `autoPoison` function must be able to wrap functions with any number of arguments; this can be done using a type-class trick that is almost identical to the way that `printf` is implemented in Haskell. We will not comment on it in detail, but we give the code here as a reference:

```
class AutoPoison p where
  autoPoison' :: [CHP ()] -> p -> p
```

```
instance AutoPoison (CHP a) where
  autoPoison' pois m = m 'onPoisonRethrow' sequence_ pois
```

```
instance (AutoPoisonParam a, AutoPoison p) =>
  AutoPoison (a -> p) where
  autoPoison' pois p x = autoPoison' (aPoison x : pois) (p x)
```

```
class AutoPoisonParam a where aPoison :: a -> CHP ()
```

```
instance AutoPoisonParam (Chanin a) where aPoison = poison
instance AutoPoisonParam (Chanout a) where aPoison = poison
instance AutoPoisonParam (PhasedBarrier phase) where
  aPoison = poison
```

```
instance AutoPoisonParam Int where aPoison _ = return ()
instance AutoPoisonParam Char where aPoison _ = return ()
```

```
autoPoison :: AutoPoison p => p -> p
autoPoison = autoPoison' []
```

There are also appropriate instances for the basic aggregate types: lists, Maybe, Either and tuples.

8.3 Example: Adding Poison

Adding poison to our `inputState'`, `draw'` and `frameLimiter'` processes is a simple matter of using the `autoPoison` function:

```
inputState = autoPoison inputState'
draw = autoPoison draw'
frameLimiter = autoPoison frameLimiter'
```

Our `eventHandler` process requires a little more code because the `Chanout` item is “hidden” in a data-type that we created. There are two options; one is to write out the poison long-hand:

```
eventHandler bindings
= eventHandler' bindings 'onPoisonRethrow'
sequence_ [poison c | Bind _ _ c <- bindings]
```

The other way is to use `autoPoison` by adding an instance for our data type:

```
instance AutoPoisonParam KeyBinding where
  aPoison (KeyBinding _ _ c) = poison c
```

```
eventHandler = autoPoison eventHandler'
```

These two methods are equivalent in their behaviour.

We have shown how to propagate poison around our process network, but we have not yet shown how we introduce it in our example. We recall our `eventHandler'` process, which was previously ignoring any `QuitEvents`. We change it to handle the event by simply throwing poison (which in turn will cause all its output channels to be poisoned):

```
eventHandler' bindings = forever $ do
  e <- liftIO_CHP getNextEvent
  case e of
    UpEvent k -> sequence_ [writeChannel c x
                            | Bind k' KeyUp x c <- bindings, k == k']
    DownEvent k -> sequence_ [writeChannel c x
                               | Bind k' KeyDown x c <- bindings, k == k']
    QuitEvent -> throwPoison
```

Growing the Example

We can now show an implementation for our tile process, that incorporates all the previous concepts into one process. We start by introducing some utility structures from the `CHP` support library:

```
data ChannelPair a = ChannelPair
  {inputPair :: Chanin a, outputPair :: Chanout a}

data FourWay above below left right = FourWay
  {above :: above, below :: below, left :: left, right :: right}
```

We can use these for connecting together our tiles; each tile will have a channel pair (i.e. input and output connection) in four directions to its neighbouring tiles. Across these channels we will transmit players as they move around – either human players or AI players, with accompanying state:

```
data PlayerState
= Human {timeToEat :: Int, chan :: Chanin (Maybe Direction)}
| DumbAI {aiDirection :: Direction}
```

```
instance AutoPoisonParam PlayerState where
  aPoison (Human _ c) = poison c
  aPoison _ = return ()
```

```
type TileCP = ChannelPair PlayerState
type TileChansP = FourWay TileCP TileCP TileCP TileCP
```

We will also define some helper functions and types to be used internally:

```
dirToChan :: Direction -> FourWay a a a a -> a
dirToChan DirLeft = left
dirToChan DirRight = right
dirToChan DirUp = above
dirToChan DirDown = below
```

```
data TileState p = Tile {tileFood :: Bool, tilePlayer :: Maybe p}
```

```
sameState :: TileState p -> TileState p -> Bool
sameState a b = tileFood a == tileFood b &&
  isJust (tilePlayer a) == isJust (tilePlayer b)
```

Now we can define our tile process:

```
tileP , tileP' :: Maybe PlayerState -> Shared Chanout TileDraw
-> TileChansP -> FrameBarrier -> CHP ()
tileP = autoPoison tileP'
tileP' initial output neighbours frame
= goPlayer (Tile False Nothing) (Tile False initial)
where
  goPlayer prev s = do
    PlayerPhase <- syncBarrier frame
    case s of
      Tile food (Just p) -> do
        (mdir, food', p') <- case p of
          Human tte dirChan -> do
            mdir <- readChannel dirChan
            return $ if tte == 0 && food
              then (mdir, False, Human 3 dirChan)
              else (mdir, food, Human (max 0 (tte-1)) dirChan)
          DumbAI dir -> return (Just dir, False, p)
        case outputPair . flip dirToChan neighbours <$> mdir of
          Nothing -> goUpdate prev (Tile food' (Just p'))
          Just out -> (do writeChannel out p'
                        goUpdate prev (Tile food' Nothing)
                        ) <|> goUpdate prev (Tile food' (Just p'))
      Tile b Nothing ->
        (do p <- alt (map readChannel neighbourInputs)
              goUpdate prev $ Tile b (Just p)
        ) <|> goUpdate prev s

  neighbourInputs = [ inputPair $ f neighbours
                    | f <- [left, right, above, below]]
```

```
goUpdate prev s = do
  UpdatePhase <- syncBarrier frame
  s' <- if tileFood s then return s
        else do r <- liftIO_CHP $ randomRIO (0 :: Int, 9999)
              return $ s {tileFood = r < 4}
  unless (sameState prev s') $
    claim output $ flip writeChannel $
      Draw (tileFood s') (isJust $ tilePlayer s)
  goPlayer s' s'
```

This process has two internal behaviours: `goPlayer` for the player phase, and `goUpdate` for the tile phase, which both begin with the appropriate synchronisation on the barrier. Both behaviours take two parameters of type `TileState`; the first is the state from the last frame, the second is the current state. We will begin by explaining the update phase. If the tile does not currently contain any food, there is a random chance (0.004%) that food will be created. Then, if the state has changed, an update is sent on the shared channel to the drawing process. After this, we move into the player phase.

In the player phase, the behaviour depends on whether the cell currently contains a player. If there is a player, the tile behaves accordingly. Human players read their direction from a channel, and only eat if they have not eaten in the last three frames. AI players can always eat, and they always move in the same direction. If the player is not moving, we proceed immediately to the update phase (we do not try to read in a player from our neighbours – only one player can occupy the tile at once).

If the player is moving, we attempt to send the player to the appropriate neighbour. Note that we do not simply commit to sending the player; if the tile's neighbour is full, this would provoke deadlock. Instead we use a common idiom: we choose between sending the player (followed by moving to the update phase) and directly moving to the update phase. We make the barrier low-priority, so the move will always be preferred if it is possible.

If the tile is empty, we use the same idiom where we offer to read a player from the tile's neighbours (followed by moving to the update phase) or directly moving to the update phase, again relying on the lower priority barrier to make sure the move is preferred. The reading from the neighbours is a good example where exclusive choice is a necessity; we do not want to accidentally read two players from different directions as this would violate our design that a tile can only hold one player.

9. Mobile Processes

The `tileP` process shown in the previous section has an undesirable feature; the range of different players is cemented into the `PlayerState` data type, and the behaviour is written in the `tileP` process, which means that new player types (e.g. a smarter AI) cannot be added without changing the data type and tile process, which is an instance of bad software design. (This is a little reminiscent of the expression problem in Haskell [12].)

We wish to decouple the player behaviour from the tile process such that we can add new players, but this is complicated by the need to be able to transfer players between tiles. There are two ways to do this: one way is to setup communication channels between the tile and player, and send the channels around to effect the player's movement (this is known in other systems as *mobile channels*); the other way is to move the player process between tiles by suspending it and sending it (this is known as *mobile processes*). We will illustrate the second way, as it has a neat implementation in Haskell. The library implementation of mobile processes is short enough that we can give it here:

```
data CHPMobileState suspendRet args ret =
  Suspended suspendRet (MobileProcess suspendRet args ret)
  | Finished ret

type MobileProcess suspendRet args ret
  = args -> CHPMobile suspendRet args ret

data CHPMobile suspendRet args ret = CHPMobile
  { runMobile :: CHP (CHPMobileState suspendRet args ret) }

instance Monad (CHPMobile suspendRet args) where
  m >>= f = CHPMobile $ do
    x <- runMobile m
    case x of
      Suspended r g -> return $ Suspended r (\a -> g a >>= f)
      Finished val -> runMobile $ f val
    return = CHPMobile . return . Finished

runMobileProcess :: CHPMobile r args ()
  -> CHP (Maybe (r, MobileProcess r args ()))

runMobileProcess p = do
  x <- runMobile p
  case x of
    Suspended r proc -> return $ Just (r, proc)
    Finished _ -> return Nothing

suspend :: r -> CHPMobile r args args
suspend x = CHPMobile $ return $ Suspended x return

class MonadCHP m where liftCHP :: CHP a -> m a
instance MonadCHP CHP where liftCHP = id
instance MonadCHP (CHPMobile sr a) where
  liftCHP = CHPMobile . fmap Finished
```

The `CHPMobile` type is a monad (effectively a CPS-like monad transformer on top of `CHP`). This supports `CHP` operations through the idiomatic lifting type-class, but also supports the `suspend` primitive. This returns the new arguments which are supplied when resuming the process from its suspension. The `suspend` call freezes the process and immediately returns it to the outer `runMobileProcess` call. Once suspended, the process can be sent over a channel and then resumed on the other side by passing it a new set of arguments. The arguments can be channels if the programmer wishes the mobile process to plug into its environment directly, or it can simply be some item of state – in our case, it is a boolean indicating whether the tile contains any food. The `suspendRet` type is returned when the process suspends to give information to the outer process; in our case that is the new food state, and a direction to head in:

```
type Player = MobileProcess (Bool, Maybe Direction) Bool ()
```

```
humanPlayer :: Chanin (Maybe Direction) -> Player
humanPlayer dirInPut = go (0 :: Int)
  where
    go untilEat food = do
      dir <- liftCHP $ readChannel dirInPut
      suspend (food', dir)
      >>= go untilEat '
    where
      (untilEat', food')
        | food && untilEat == 0 = (3, False)
        | otherwise = (max 0 (untilEat - 1), food)

aiPlayer :: Direction -> Player
aiPlayer d _ = forever $ suspend (False, Just d)
```

Note that the players are written in a simple sequential style, even though during the `suspend` calls they may be transferred between tiles – it makes sense to write the player behaviour as a simple sequence, and this mobile process monad allows us to do exactly that. The `tile` process can now be re-written to support the use of the mobile process players:

```
type TileConn = ChannelPair PlayerMove
newtype PlayerMove = PlayerMove Player
type TileChans = FourWay TileConn TileConn TileConn TileConn

tile , tile' :: Maybe Player -> Shared Chanout TileDraw
  -> TileChans -> FrameBarrier -> CHP ()
tile = autoPoison tile'
tile' initial output neighbours frame
  = goPlayer (Tile False Nothing) (Tile False initial)
  where
    goPlayer prev s = do
      PlayerPhase <- syncBarrier frame
      case s of
        Tile b (Just p) -> do
          Just ((food, mdir), p') <- runMobileProcess (p b)
          case outputPair . flip dirToChan neighbours <$> mdir of
            Nothing -> goUpdate prev (Tile food (Just p'))
            Just out -> (do writeChannel out p'
                          goUpdate prev (Tile food Nothing)
                          ) <|> goUpdate prev (Tile food (Just p'))
        Tile b Nothing ->
          (do p <- alt (map readChannel neighbourInputs)
              goUpdate prev $ Tile b (Just p)
          ) <|> goUpdate prev s
```

The code for `goUpdate` is unchanged so we omit it for brevity. The change here is that now the new state of the tile is calculated by running the mobile process (player) currently held. Thus the player behaviour has been entirely moved from the `tile` process into the new player mobile processes.

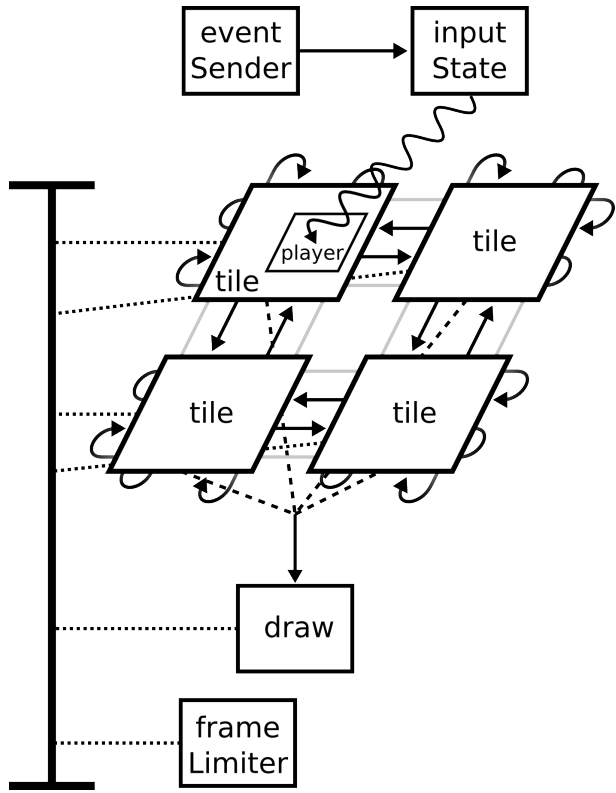


Figure 1. The process network for our game example, shown with a 2×2 tile grid for simplicity. The boxes are processes, the arrows are communication channels, and the I-beam on the left is a barrier (enrollment is indicated by the dotted line attaching processes to the barrier).

10. Process Composition and Wiring

The connectivity diagram of our process network is shown in figure 1; the ability to create these diagrams is a useful aspect of CHP's concurrency model. We now need to implement this in our code, wiring together all the processes. The "long-hand" way of doing this is to create all the channels and the barrier from the diagram, name them individually, and then perform all appropriate enrolling on the barriers and pass all the names to the correct processes.

This long-hand approach is error-prone. While type-checking will prevent, say, mixing up an update channel with an inter-tile channel (or mixing up a reading and writing end of a channel), all the inter-tile channels have the same type so it would still be possible to mis-wire the process grid. Other concurrent languages tend to use arrays for wiring this kind of design, but it is still easy to make an error in the indexing, especially with the channels which must wrap from one side of the grid (and thus array) to another.

Fortunately, CHP has a solution to this problem in the form of composition combinators [2]. It provides a composition monad: `Composed a`. Rather than being a sequence of imperative actions, in this monad each item is a "wiring instruction" that composes more processes, and eventually the list of fully-wired processes are returned to be run in parallel by the run function. For example, there is the `enrollAllIR` function which enrolls each of the given list of processes on the given barrier:

```
enrollAllIR :: Unenrolled PhasedBarrier phase
            -> [PhasedBarrier phase -> a] -> Composed [a]
```

Assume that we have a list of processes which all need to be enrolled on two different barriers:

```
processes :: [PhasedBarrier a -> PhasedBarrier b -> CHP ()]
```

We can enroll them as follows:

```
wired :: Composed [CHP ()]
wired = do
  a <- newPhasedBarrier startPhaseA
  b <- newPhasedBarrier startPhaseB
  enrollAllIR a processes >>= enrollAllIR b
```

We can then run this using one of the outer-level run functions:

```
run :: Composed [CHP a] -> CHP [a]
run_ :: Composed [CHP a] -> CHP ()
```

Another function provided by the library which is very useful for our example is:

```
wrappedGridFourR ::
  (Connectable below above, Connectable right left) =>
  [[FourWay above below left right -> a]] -> Composed [[a]]
```

The `Connectable` type-class declares that the two types can be connected together. We leave aside further details, but there is an instance for `ChannelPair`:

```
instance Connectable (ChannelPair a) (ChannelPair a)
```

This `wrappedGridFourR` function composes together a rectangle of processes by joining them in a wrapped grid, just as we want. The existence of this function makes our own code less error-prone and much terser for not having to implement this ourselves. It may seem fortuitous that this exact function is provided, but wiring together a grid like this is a very common need in simulations, which meant it was worth providing as a library function (which has been in the library for some time now).

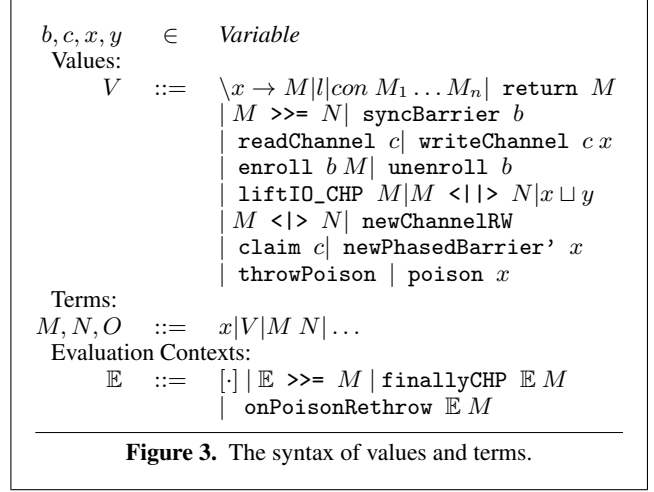
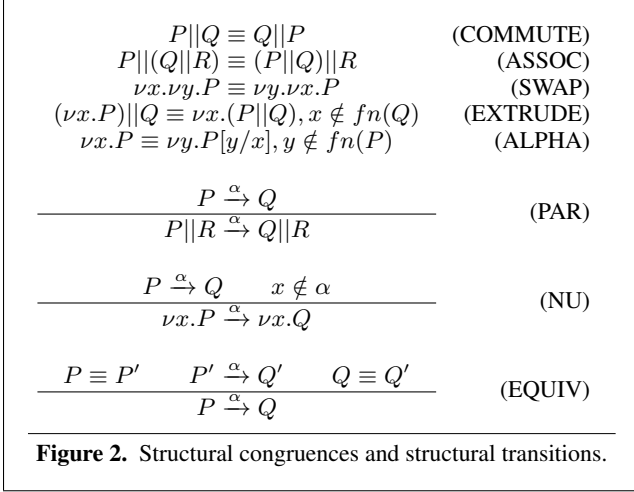
With all these wiring combinators in hand, we can now wire together our process network:

```
game :: CHP ()
game = run_ $ do
  playerKeys <- newChannelRW
  (recvUpdate, sendUpdate) <- newChannelRW
  let thePlayer = player (fst playerKeys)
      player x y
          | x == 0 && y == 0 = Just thePlayer
          | x == 0 = Just $ aiPlayer DirRight
          | otherwise = Nothing
      tiles <- concat <$> wrappedGridFourR
          [[ tile (player x y)
              (mapSharedChanout ((,) (x,y)) sendUpdate)
              | x <- [0..(size-1)] | y <- [0..(size-1)]]]
      barOpts = defaultBarOpts { barPriority = -1 }
      bar <- newPhasedBarrier' UpdatePhase barOpts
      procs <- enrollAllIR bar
      (draw recvUpdate : frameLimiter 20 PlayerPhase : tiles)
      return $ ((eventSender . defaultBindings)
                |<=> inputState $ snd playerKeys) : procs
```

The final combinator we are using does not use the `Composed` monad, but it is another combinator for automatically wiring processes together:

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

This joins together a producer-like process on the left-hand side, with a processor on the right-hand side, into a new producer.



11. Operational Semantics

CHP is heavily influenced by – and named for – the Communicating Sequential Processes (CSP) calculus [6, 10] which can provide a formal description of process behaviour. We forgo CSP and choose to give an operational semantics in the style of Peyton Jones [8] instead, as this style is more common in the Haskell literature.

We begin with standard common definitions and structural congruences in figure 2, and the definition of terms and evaluation contexts in figure 3. A transition in our semantics is represented as $\xrightarrow{\alpha}$, where α is a set of events that occurred. We focus here on the concurrent semantics of CHP, which sit alongside the existing semantics for IO. One feature that unavoidably interacts with CHP is exceptions. We note that exceptions (i.e. those that can be caught in the IO monad) arise from four different sources:

1. Exceptions in pure code (such as failed pattern matches) we consider to be errors, and they will be propagated to the outermost runCHP call.
2. Similarly, any deadlock exceptions (which should not occur when poison is used correctly) are considered to be errors.
3. Exceptions arising in lifted IO code (which must be wrapped in liftIO.CHP) that are not caught in the lifted code are also considered errors.
4. Asynchronous exceptions are at odds with the CHP programming model, and while they are propagated by the semantics, they are not encouraged in CHP programs.

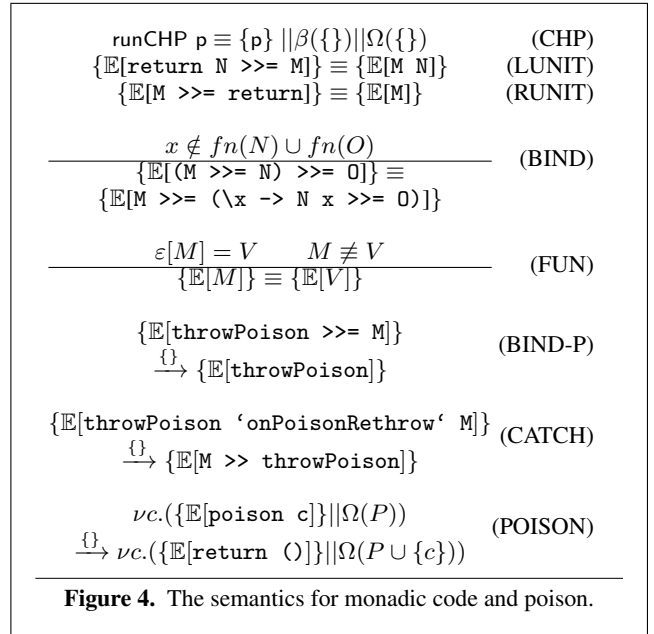
11.1 Monad and Poison

We give the semantics for the monad transitions in figure 4. Many are standard, and the poison transitions are mostly straightforward. The semantics introduce the set P , tagged using $\Omega(P)$ which is the set of names that are poisoned. This is effectively global, as the only instance of set is introduced by the top-level runCHP call (along with a β set which is needed for the barrier semantics).

11.2 Channels and Barriers

The formulaic semantics for operating on channels and barriers when they are poisoned are given in figure 5. Simply, using them when poisoned leads to a poison exception being thrown.

The semantic rule for creating channels is given in figure 6. We use the subscripts R and W to indicate the channel ends (since they have a different type in CHP), but semantically c_R and c_W are simply aliases for c . Communicating on a non-poisoned channel is



straightforward: a writer and reader are required, and the value is transferred between the two processes.

Barriers require keeping track of the current enrollment count of the barrier, and the current (and future) phase. To do this we introduce an associative map, B , tagged as $\beta(B)$ which maps barrier identifiers to a pair with the enrollment count and infinite phase list (the head of which is the current phase) – we use the notation $B \triangleleft B'$ to mean a union of associative maps with a bias to the right (i.e. the mappings from the right override those from the left). The barrier creation rule (given in figure 6) adds the new barrier to the map with an initial enrollment count of 0. Similar to channels, we use the U subscript for unenrolled barrier ends and E for enrolled, but b_U and b_E are just aliases for b .

Synchronising on a barrier requires as many processes as the enrollment count, and they are all returned the current phase (which is removed from the phase list, thus advancing the phase). Enrolling on a barrier adds to the enrollment count; we use the `unenroll` function (which is a semantic construct that is never visible to

$$\begin{array}{l}
\nu c.(\{\mathbb{E}[\text{readChannel } c]\}|\Omega(P)) \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\text{throwPoison}]\}|\Omega(P)) \quad , c \in P \quad (\text{READ-P}) \\
\nu c.(\{\mathbb{E}[\text{writeChannel } c]\}|\Omega(P)) \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\text{throwPoison}]\}|\Omega(P)) \quad , c \in P \quad (\text{WRITE-P}) \\
\nu b.(\{\mathbb{E}[\text{syncBarrier } b]\}|\Omega(P)) \xrightarrow{\{\}} \nu b.(\{\mathbb{E}[\text{throwPoison}]\}|\Omega(P)) \quad , b \in P \quad (\text{SYNC-P}) \\
\nu x.(\{\mathbb{E}[\text{enroll } x \ M]\}|\Omega(P)) \xrightarrow{\{\}} \nu x.(\{\mathbb{E}[\text{throwPoison}]\}|\Omega(P)) \quad , x \in P \quad (\text{ENROLL-P}) \\
\nu x.(\{\mathbb{E}[\text{unenroll } x]\}|\Omega(P)) \xrightarrow{\{\}} \nu x.(\{\mathbb{E}[\text{throwPoison}]\}|\Omega(P)) \quad , x \in P \quad (\text{UNENROLL-P})
\end{array}$$

Figure 5. The semantics for using channels and barriers in the presence of poison.

$$\begin{array}{l}
\{\mathbb{E}[\text{newChannelRW}]\} \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\text{return } (c_R, c_W)]\}, c \notin \text{fn}(\mathbb{E})) \quad (\text{NEW-CHANNEL}) \\
\nu c.(\{\mathbb{E}_1[\text{readChannel } c_R]\}_t || \{\mathbb{E}_2[\text{writeChannel } c_W \ x]\}_u || \Omega(P)) \xrightarrow{\{c\}} \nu c.(\{\mathbb{E}_1[\text{return } x]\}_t || \{\mathbb{E}_2[\text{return } ()]\}_u || \Omega(P)), c \notin P \quad (\text{BASIC-COMM-P}) \\
\frac{b \notin (\text{keys}(B) \cup \text{fn}(\mathbb{E}))}{\{\mathbb{E}[\text{newPhasedBarrier } h]\}|\beta(B) \xrightarrow{\{\}} \nu b.(\{\mathbb{E}[\text{return } b_U]\}|\beta(B \triangleleft \{b \mapsto (0, h)\}))} \quad (\text{NEW-BP}) \\
\frac{(b \mapsto (k, h)) \in B \quad b \notin P}{\nu b.(\{\mathbb{E}[\text{enroll } b_U \ M]\}|\beta(B)||\Omega(P)) \xrightarrow{\{\}} \nu b.(\{\mathbb{E}[M \ b_E \ \text{‘finallyCHP’ } \text{unenroll } b_E]\}|\beta(B \triangleleft \{b \mapsto (k+1, h)\})||\Omega(P))} \quad (\text{ENROLL-BP-P}) \\
\frac{(b \mapsto (k, h)) \in B \quad b \notin P}{\nu b.(\{\mathbb{E}[\text{unenroll } b_E]\}|\beta(B)||\Omega(P)) \xrightarrow{\{\}} \{\mathbb{E}[\text{return } ()]\}|\beta(B \triangleleft \{b \mapsto (k-1, h)\}))} \quad (\text{UNENROLL-BP-P}) \\
\frac{(b \mapsto (k, o : h : h')) \in B \quad b \notin P}{\nu b.(\{\{\mathbb{E}_i[\text{syncBarrier } b_E]\}_{t_i} | i \in \{1..k\}\}|\beta(B)||\Omega(P)) \xrightarrow{\{b\}} \nu b.(\{\{\mathbb{E}_i[\text{return } h]\}_{t_i} | i \in \{1..k\}\}|\beta(B \triangleleft \{b \mapsto (k, h : h')\})||\Omega(P))} \quad (\text{SYNC-BP-P})
\end{array}$$

Figure 6. The semantics for creating channels and barriers, communication on channels and enrolling/synchronising on barriers.

the library user) for unenrolling, which removes one from the synchronisation count.

11.3 Parallel Composition

We write the semantics for parallel composition in terms of a binary parallel operator:

$(\langle | \rangle) :: \text{CHP } a \rightarrow \text{CHP } b \rightarrow \text{CHP } (a, b)$

The list version, `parallel`, can be thought of as being defined:

```
parallel = foldr (\p ps -> uncurry (:) <$> (p <| |> ps))
           (return [])
```

The semantics for parallel composition are given in figure 7. The intuition is that parallel composition waits for its sub-processes to finish, and then IO exceptions trump poison which trumps normal returns. The notable aspect is the rule for asynchronous exceptions. We define that asynchronous exceptions received by the parent process during a parallel composition are propagated to the child processes. This is necessary for a lot of common sense rules about CHP, including the basic rule: `parallel [p] = p`.

11.4 Choice

The semantics for choice are given in figure 8. The semantics for choice are complicated due to the ability to mix always-ready guards, timeout guards and guards based on events (i.e. communication and synchronisation). A full treatment is given elsewhere [3],

but the intuition is that always-ready guards on the left are chosen over anything on the right, but always-ready guards on the right are only chosen if the left-hand item cannot be chosen. We must also include a few extra rules for return acting as a guard by itself (it is an always-ready guard).

12. Implementation

CHP’s parallel composition is built on top of the `forkIO` primitive, while its channels and barriers are built using Software Transactional Memory (STM). The algorithms behind supporting choice are quite complex, mainly due to the presence of a feature known as conjunction, and a detailed explanation is provided elsewhere [3]. Supporting choice without conjunction is relatively straightforward².

To give an idea of the implementation, we provide the definition of several core primitives of CHP, leaving aside the issue of choice behind the function:

```
select :: [(Guard, IO a)] -> IO a
```

The `Guard` type represents an item in a choice:

```
data Guard = Ready | Timeout Int | Event Event
```

²See the Haskell “sync” library for a reference implementation of choice: <http://hackage.haskell.org/package/sync/>

$$\begin{array}{l}
\{\mathbb{E}[M \langle | \rangle N]\}_t \xrightarrow{\Omega} \nu u. \nu v. (\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\mathbb{E}[M]\}_u \parallel \{\mathbb{E}[N]\}_v), u \notin fn(\mathbb{E}) \wedge v \notin fn(\mathbb{E}) \wedge u \neq v \quad (\text{S-PAR}) \\
\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\text{return } x\}_u \parallel \{\text{return } y\}_v \xrightarrow{\Omega} \{\mathbb{E}[\text{return } (x, y)]\}_t \quad (\text{E-PAR-R}) \\
\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\text{throw } e\}_u \parallel \{\text{return } y\}_v \xrightarrow{\Omega} \{\mathbb{E}[\text{throw } e]\}_t \quad (\text{E-PAR-TR}) \\
\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\text{throw } e\}_u \parallel \{\text{throw } f\}_v \xrightarrow{\Omega} \{\mathbb{E}[\text{throw } e]\}_t \quad (\text{E-PAR-T}) \\
\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\text{throwPoison}\}_u \parallel \{\text{return } y\}_v \xrightarrow{\Omega} \{\mathbb{E}[\text{throwPoison}]\}_t \quad (\text{E-PAR-PR}) \\
\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\text{throwPoison}\}_u \parallel \{\text{throwPoison}\}_v \xrightarrow{\Omega} \{\mathbb{E}[\text{throwPoison}]\}_t \quad (\text{E-PAR-P}) \\
\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\text{throw } e\}_u \parallel \{\text{throwPoison}\}_v \xrightarrow{\Omega} \{\mathbb{E}[\text{throw } e]\}_t \quad (\text{E-PAR-TP}) \\
\{\mathbb{E}[u \sqcup v]\}_t \equiv \{\mathbb{E}[v \sqcup u]\}_t \quad (\text{E-PAR-COMM}) \\
\{\mathbb{E}[u \sqcup v]\}_t \parallel [t \not\downarrow e] \xrightarrow{\Omega} \{\mathbb{E}[u \sqcup v]\}_t \parallel [u \not\downarrow e] \parallel [v \not\downarrow e] \quad (\text{PAR-ASYNC})
\end{array}$$

Figure 7. Semantics for parallel composition. The \sqcup notation is invented purely for describing the semantics, and is not visible to the programmer. Note that, unlike the Haskell function `forkIO`, the thread identifiers are not made visible to the programmer.

$$\begin{array}{l}
\{\mathbb{E}[\text{return } x \langle | \rangle N]\} \xrightarrow{\Omega} \{\mathbb{E}[\text{return } x]\} \quad (\text{CHLRET}) \\
\frac{\{M\} \not\rightarrow \{M'\} \quad M \not\equiv (A \langle | \rangle B)}{\{\mathbb{E}[M \langle | \rangle \text{return } x]\} \xrightarrow{\Omega} \{\mathbb{E}[\text{return } x]\}} \quad (\text{CHRRET}) \\
(M \langle | \rangle N) \langle | \rangle O \equiv M \langle | \rangle (N \langle | \rangle O) \quad (\text{CHASSC}) \\
\frac{\{M\} \xrightarrow{\alpha} \{M'\} \quad M \not\equiv (A \langle | \rangle B)}{\{\mathbb{E}[M \langle | \rangle N]\} \xrightarrow{\alpha} \{\mathbb{E}[M']\}} \quad (\text{CHLEFT}) \\
\frac{((\{M\} \not\rightarrow \{M'\} \wedge \{N\} \xrightarrow{\alpha} \{N'\}) \vee (\{M\} \xrightarrow{\beta} \{M'\} \wedge \{N\} \xrightarrow{\alpha} \{N'\} \wedge \alpha \neq \beta)) \quad M \not\equiv (A \langle | \rangle B)}{\{\mathbb{E}[M \langle | \rangle N]\} \xrightarrow{\alpha} \{\mathbb{E}[M']\}} \quad (\text{CHRIGHT})
\end{array}$$

Figure 8. The semantics for the choice and conjunction operators. The condition $M \not\equiv (A \langle | \rangle B)$ on many of the rules forces the choice to be rearranged into a right-associative form (via CHASSC) before proceeding. The condition $\{M\} \not\rightarrow \{M'\}$ is only satisfied when M cannot make any transition, while $\{M\} \xrightarrow{\beta} \{M'\}$ is satisfied when M can make a transition with a non-empty set of events or M cannot make any transition.

12.1 Monad

The simplest way to define the monad is with a GADT as follows³:

```

data CHP a where
  IO :: IO a -> CHP a
  Return :: a -> CHP a
  Bind :: CHP a -> (a -> CHP b) -> CHP b
  Choice :: [(Guard, CHP a)] -> CHP a
  Rethrow :: CHP a -> CHP b -> CHP a
  ThrowPoison :: CHP a
  Finally :: CHP a -> CHP b -> CHP a

```

instance Monad CHP where

```

return = Return
(>>=) = Bind

```

³ It is possible to reduce the number of constructors; for example `Return x` is equivalent to `IO (return x)`, but for clarity we are verbose.

```

liftIO_CHP = IO
throwPoison = ThrowPoison
onPoisonRethrow = Rethrow
finallyCHP = Finally

```

This is then processed by a “driver” function:

```

data WithPoison a = Poison | NoPoison a

```

```

goCHP :: CHP a -> IO (WithPoison a)
goCHP (Return x) = return (NoPoison x)
goCHP (Bind m k) = goCHP m >>= \v -> case v of
  NoPoison x -> goCHP (k x)
  Poison -> return Poison
goCHP (IO m) = NoPoison <$> m
goCHP (Choice gas) = select [(g, goCHP a) | (g, a) <- gas]
goCHP (Rethrow m h) = goCHP m >>= \v -> case v of
  NoPoison x -> return (NoPoison x)
  Poison -> goCHP (h >> throwPoison)
goCHP ThrowPoison = return Poison
goCHP (Finally m h) = ((Right <$> goCHP m)
  'catch' (\(e :: SomeException) -> return (Left e)))
  >>= \v -> case v of
    Left e -> goCHP h >> throw e
    Right Poison -> goCHP (h >> throwPoison)
    Right (NoPoison x) -> return x

```

12.2 Choice

Choice is carried out using the aforementioned `select` function in `goCHP`; the choice operators simply operate on the GADT:

instance Alternative CHP where

```

empty = Choice []
⟨ | ⟩ a b = Choice (asGuards a ++ asGuards b)

```

```

asGuards :: CHP a -> [(Guard, CHP a)]
asGuards (Choice gs) = gs
asGuards (IO e) = [(SkipGuard, IO e)]
asGuards (Return x) = [(SkipGuard, Return x)]
asGuards (Bind m k) = [(g, Bind a k) | (g, a) <- asGuards m]
asGuards (Rethrow m h)
  = [(g, Rethrow a h) | (g, a) <- asGuards m]
asGuards ThrowPoison = [(SkipGuard, ThrowPoison)]
asGuards (Finally m h)
  = [(g, Finally a h) | (g, a) <- asGuards m]

```

13. Related Work

There are several Haskell concurrency frameworks that provide primitives that can be used for message-passing, such as MVars [9] (one-place buffered communication channels which do not support choice) or TChans [5] (asynchronous channels) or various CML-inspired libraries [4, 11] (which have a slightly different API to CHP, as explained in section 6). These libraries are alternatives to the core of CHP, but the value of CHP lies in its richer features such as mechanisms to easily connect together processes. Many of these mechanisms could be trivially altered to build on top of a different core message-passing Haskell library.

CHP's lineage lies outside functional programming. The occam programming language originally embodied the Communicating Sequential Processes calculus, and begat many libraries in other languages: JCSP (for Java), C++CSP and so on. These libraries generally suffered from embedding a different programming model into their existing syntax – CHP's use of Haskell resulted in a neater API than using a traditional imperative language. Haskell also avoids issues with shared imperative state and issues such as destruction in a concurrent setting.

Erlang is another functional language that supports message-passing. Its use of asynchronous mailbox-based messaging makes it surprisingly far from CHP: choice is not applicable because communications are asynchronous, barriers would require a new API, process composition combinators do not make sense because there are no channels, and so on.

14. Discussion and Conclusions

CHP is a Haskell library that embodies a particular concurrent programming model. It has been suggested in the past that it is in fact an embedded domain-specific language (EDSL) rather than a library. In Haskell the line between the two is often blurred, but one strong counter-argument is simply that CHP is not domain-specific! It can be used for programming GUIs, for programming simulations, web servers and more. Its strength is in writing interactive programs or systems of communicating agents, but it can equally be used for calculating prime numbers or sorting.

Several of CHP's core capabilities (basic parallel composition and channel communications) are already present in many Haskell libraries, such as the CML-based libraries [4, 11]. These capabilities can be considered the basic building blocks of message-passing concurrency – and CHP builds on them to former a much richer framework. CHP adds phased barriers, poison for concurrent termination, mobile processes, and adds combinators for easy process composition. Beyond that – but not explored in this paper – it also has the ability to record traces of concurrent programs, support for conjunction of events and additional support for testing and generating formal specifications [3].

It is these further capabilities of CHP – beyond the basic building blocks – that are only possible because of using Haskell. The poison mechanism can be largely emulated using the exceptions mechanism in other languages, but the automatic poison, mobile processes, process combinators and several further capabilities are only possible because of the support for monads, type-classes and higher-order processes that Haskell provides. CHP shows that a declarative functional lazy language is a good language in which to support imperative concurrent message-passing.

One issue with CHP that makes it less accessible is the use of its own monad. The typical way to program a CHP system is to use the CHP monad, lifting IO actions where necessary. If a large program has already been written purely using IO, it is awkward to change it to use the CHP library. The reasons that CHP has its own monad are to support choice between the left-most action, to support poison and to support tracing. These are some of the distinctive features of

CHP; if a programmer does not want to make use of these, they may be better suited to choose a smaller message-passing library. We note that many of the ideas shown here, of composing processes, phased barriers or mobile processes could be implemented in a message-passing library on top of the IO monad instead.

References

- [1] N. C. C. Brown. Communicating Haskell Processes: Composable explicit concurrency using monads. In *Communicating Process Architectures 2008*, pages 67–83, 2008.
- [2] N. C. C. Brown. Combinators for message-passing in Haskell. In R. Rocha and J. Launchbury, editors, *Practical Aspects of Declarative Languages*, volume 6359 of *Lecture Notes in Computer Science*, pages 19–33. Springer-Verlag Berlin / Heidelberg, 2011.
- [3] N. C. C. Brown. Communicating Haskell Processes, 2011. URL <http://www.twistedsquare.com/thesis.pdf>.
- [4] A. Chaudhuri. A concurrent ML library in concurrent Haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 269–280. ACM, 2009.
- [5] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05*, pages 48–60. ACM, 2005.
- [6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. URL <http://www.usinccsp.com/>.
- [7] S. Marlow, S. Peyton Jones, A. Moran, and J. Reppy. Asynchronous exceptions in Haskell. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 274–285, New York, NY, USA, 2001. ACM.
- [8] S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pages 47–96. IOS Press, 2001. Revised 2002–2010.
- [9] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Symposium on Principles of Programming Languages*, pages 295–308. ACM Press, 1996.
- [10] A. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997. URL <http://www.comlab.ox.ac.uk/people/bill.roscoe/publications/68b.pdf>.
- [11] G. Russell. Events in Haskell, and how to implement them. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 157–168. ACM, 2001.
- [12] W. Swierstra. Data types à la carte. *J. Funct. Program.*, 18:423–436, July 2008. ISSN 0956-7968.
- [13] P. H. Welch. Graceful termination – graceful resetting. In A. W. P. Bakkers, editor, *OUG-10: Applying Transputer Based Parallel Machines*, pages 310–317, 1989.