Communicating Haskell Processes tutorial

Neil C. C. Brown

February 5, 2010

Introduction

This document is a tutorial for the Communicating Haskell Processes (CHP) library. CHP is based on a model of concurrency known as process-oriented programming, which primarily stems from ideas in the Communicating Sequential Processes calculus of Tony Hoare.

Process-oriented programming is based around having encapsulated processes, with no shared mutable data. Of course, in Haskell, all normal data is immutable. But the processes do not communicate via mutable shared variables (like they do with MVars in Concurrent Haskell and TVars in STM). Instead they communicate using synchronous channels.

If you write to a synchronous channel, you must wait until the reader arrives to take the data before the communication takes place. Thus, you can be sure that the reader has taken the data once the call to write to a channel returns. There are also buffering processes that can be used to get buffering behaviour.

Throughout this guide you will see diagrams of process networks. The labelled boxes are processes and the arrows show the channel connections. On many of the diagrams there are stdin, stdout or stderr channels that are shown as being connected to the outside environment.

Prerequisite Knowledge

This tutorial assumes that you already know Haskell. A basic knowledge of monads is necessary to understand how to use the library. If you know how to use the IO monad, you should get on fine with CHP.

Contents

1 Simple Stream Processing Programs

1.1 Echoing

For the purposes of introducing CHP we will generate some programs that deal with the console (stdin, stdout and stderr). Without further ado, let's look at the simplest program we can imagine, that echoes stdin on stdout:

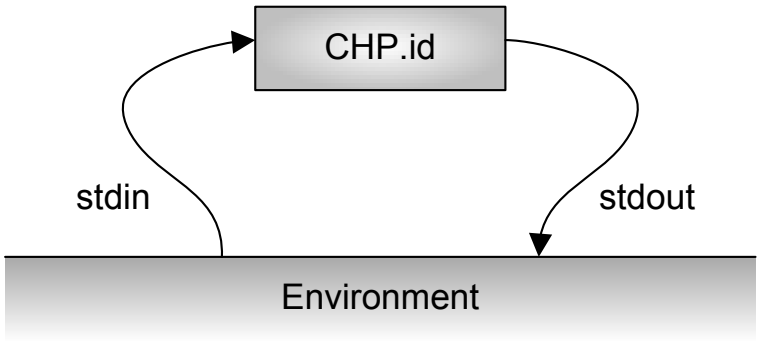
```
import Control.Concurrent.CHP
import qualified Control.Concurrent.CHP.Common as CHP
import Control.Concurrent.CHP.Console

main :: IO ()
main = runCHP_ (consoleProcess echo)

echo :: ConsoleChans -> CHP ()
echo chans = CHP.id (cStdin chans) (cStdout chans)
```

We begin with some module imports. Importing Control.Concurrent.CHP pulls in most of the library, but we also need the console module, and we import some common processes qualified. It may seem odd to import Common qualified as CHP, but writing CHP.id makes more sense than Common.id.

The main function in our program consists of running a CHP program using runCHP_ (type: CSP a ->IO ()). We use the consoleProcess helper function to provide ConsoleChans for us, to our actual echo function. echo uses the common id process to plug together stdin and stdout. The id process forever forwards values from its input to its output.

So this program does echo all characters typed on stdout[1], by simply copying the input to the output.

[1] If you run it, characters may appear twice because your terminal is also echoing the characters, but the program itself is doing what we intended! You may also find that characters don't show up until you press return, due to output buffering – running it through GHCi is probably best for these simple examples.



1.2 Processing

We will now add to our program by doing some processing. Instead of just echoing the character, we will filter out some characters. As an example, we will filter out all lowercase x characters:

```
import Control.Concurrent.CHP
import qualified Control.Concurrent.CHP.Common as CHP
import Control.Concurrent.CHP.Console

main :: IO ()
main = runCHP_ (consoleProcess not_x)

not_x :: ConsoleChans -> CHP ()
not_x chans = CHP.filter (/= 'x') (cStdin chans) (cStdout chans)
```

Again we are using a standard CHP process, this time `filter`, which is the process equivalent of the normal filter function.

1.3 Writing our own process

The previous two examples used library-supplied processes. We will now write our own process that prints out a character's ordinal value when it is input:

```
import Control.Concurrent.CHP
import qualified Control.Concurrent.CHP.Common as CHP
import Control.Concurrent.CHP.Console
import Control.Monad

main :: IO ()
main = runCHP_ (consoleProcess printOrd)

printOrd :: ConsoleChans -> CHP ()
printOrd chans = forever (do
  x <- readChannel (cStdin chans)
  let ordXStr = show (fromEnum x) ++ "\n"
  mapM_ (writeChannel (cStdout chans)) ordXStr
  )
```

This process uses the Haskell function `forever` (from `Control.Monad`) to repeat our monadic `do` block. Each time, it first reads a character from stdin, then converts it to a `String` showing the ordinal value, then writes that out, one character at a time, on stdout.

2 Adding Choice and Parallelism

2.1 Choice

So far, all of our programs could be easily written using normal Haskell functions in the IO monad, and shorter too. Now we will start to introduce some of the more powerful features of CHP, beginning with choice.

We will write a program that waits for the user to press a single character, and then print the ordinal value on the screen as in the last example. But then we will wait for either the user to press another key, or for a second to elapse. If the second elapses without the user pressing a key, we will repeat the last ordinal code. If the user does press a key, the new ordinal code will be printed, and we will wait for either another keypress or another second to expire:

```
import Control.Concurrent.CHP
import qualified Control.Concurrent.CHP.Common as CHP
import Control.Concurrent.CHP.Console
import Control.Monad

main :: IO ()
main = runCHP_ (consoleProcess printOrdWhileWaiting)

printOrdWhileWaiting :: ConsoleChans -> CHP ()
printOrdWhileWaiting chans = readChannel cin >>= inner
  where
    inner :: Char -> CHP ()
    inner x = do
      printString (show (fromEnum x) ++ "\n")
      (waitFor 1000000 >> inner x) <-> (readChannel cin >>= inner)

    cin = cStdin chans
    cout = cStdout chans

    printString = mapM_ (writeChannel cout)
```

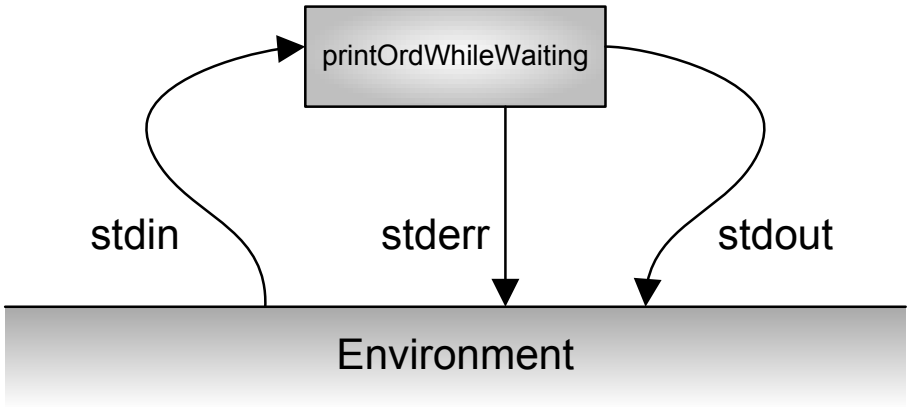
Our outer function reads the first character and feeds this to the inner function. The inner function then prints the ordinal value. The line after that uses the `<->` operator, which represents choice. The operator causes the process to wait for the first of the two events, *choosing* the first that is ready. In this case the two events are waiting for a second (one million microseconds) and reading from a channel, because these are the first actions in either of the monadic blocks being combined.

2.2 Parallel composition

In this example, once a character is pressed, we will print the string on both standard output and standard error. For demonstration purposes, we will send the strings in parallel:

```
import Control.Concurrent.CHP
import qualified Control.Concurrent.CHP.Common as CHP
import Control.Concurrent.CHP.Console
import Control.Monad

main :: IO ()
main = runCHP_ (consoleProcess printOrdWhileWaiting)

printOrdWhileWaiting :: ConsoleChans -> CHP ()
printOrdWhileWaiting chans = readChannel cin >>= inner
  where
    inner :: Char -> CHP ()
    inner x = do
      printString (show (fromEnum x) ++ "\n")
      (waitFor 1000000 >> inner x) <-> (readChannel cin >>= inner)

    cin = cStdin chans
    cout = cStdout chans
    cerr = cStderr chans

    printString s
      = mapM_ (writeChannel cout) s <||> mapM_ (writeChannel cerr) s
```

Here we have used the `<||>` operator to run two monadic actions in parallel. You can also run **do** blocks and functions in parallel.



2.3 Process pipelines

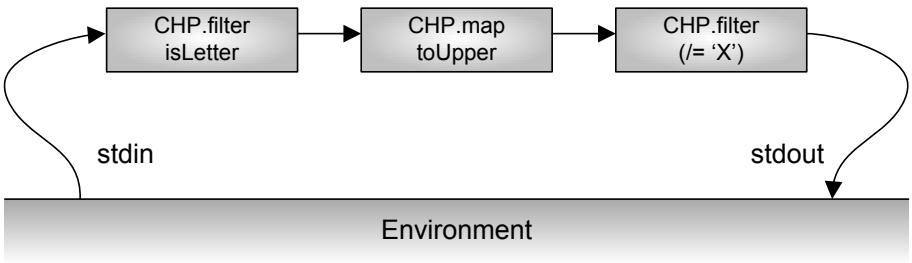
One common pattern in process-oriented programming is to create a pipeline of processes, similar to forming a function composition in Haskell. For this purpose there are helper functions in the library to easily wire up process pipelines with channels, in the Control.Concurrent.CHP.Connect module in the `chp-plus` library.

For the purposes of demonstration, we will remove all the non-alphabetic characters from the input, then make the character upper-case, then drop all X characters from the output:

```
import Control.Concurrent.CHP
import qualified Control.Concurrent.CHP.Common as CHP
import Control.Concurrent.CHP.Console
import Control.Concurrent.CHP.Connect
import Control.Monad
import Data.Char

main :: IO ()
main = runCHP_ (consoleProcess crazyPipeline)

crazyPipeline :: ConsoleChans -> CHP ()
crazyPipeline chans = do
  pipelineConnect [CHP.filter isLetter, CHP.map toUpper, CHP.filter (/= 'X')]
    (cStdin chans) (cStdout chans)
  return ()
```

The pipeline function takes a list of single-input single-output processes, an input channel-end for the start of the pipeline, an output channel-end for the end of the pipeline, and then wires up the processes and runs them all in parallel.

This example could be done with a single process, but often it is easier to wire up a list of already-written processes.



3 Tracing

As in all programming, while writing concurrent programs you may encounter a bug. Concurrency-specific bugs include deadlock (the whole program grinding to a halt because everyone is waiting for someone else) and livelock (programs doing something but not getting anywhere). To try to help in understanding what your program is actually doing, CHP provides a tracing facility. For demonstration, we will use the classic dining philosophers problem. The description of the problem can easily be found on the Internet (including its original description in Hoare's book at `http://www.usingcsp.com/`), and will not be repeated here.

We will use channels carrying the unit type to represent the forking picking-up and putting-down events. This is preferred to barriers since we do not need to deal with all the enrolling. For the purposes of demonstration (shorter code and more likely deadlock) we will only use two philosophers.

```
import Control.Concurrent.CHP
import Control.Monad
import Control.Monad.Trans
import System.Random

spaghettiFork :: Chanin () -> Chanin () -> CHP ()
spaghettiFork claimA claimB = forever
  ( (readChannel claimA >> readChannel claimA)
    <-> (readChannel claimB >> readChannel claimB) )

philosopher :: Chanout () -> Chanout () -> CHP ()
philosopher left right = forever (do
  randomDelay -- Thinking
  writeChannel left () >> writeChannel right ()
  randomDelay -- Eating
  writeChannel left () <||> writeChannel right ()
  )
  where
    randomDelay = liftIO (getStdRandom (randomR (0, 50000))) >>= waitFor

college :: CHP ()
college = do
  phil0Left <- newChannel
  phil0Right <- newChannel
  phil1Left <- newChannel
  phil1Right <- newChannel
  runParallel_
    [ philosopher (writer phil0Left) (writer phil0Right)
    , philosopher (writer phil1Left) (writer phil1Right)
    , spaghettiFork (reader phil0Left) (reader phil1Right)
    , spaghettiFork (reader phil1Left) (reader phil0Right) ]

main :: IO ()
main = runCHP_ college
```

Usually we would pick up the forks in parallel, but while that design can still deadlock, it run for a long time before deadlocking! To advance our lesson, we pick up the forks in sequence and we have a fairly small time for thinking and eating (tens of milliseconds). On my GHC setup, the program deadlocks within a minute, with the message "thread blocked indefinitely".

This error message is not very detailed when you are trying to track down a bug in your program. To help get more information, you can add one more import declaration at the top of your program:

```
import Control.Concurrent.CHP.Traces
```

And then change the definition of your main function:

```
main = runCHP_CSPTraceAndPrint college
```

Now when you re-run your program, instead of the deadlock message, you will get a trace printed that looks like this:

```
< c0, c1, c0, c1, c2, c3, c2, c3, c0, ... , c3, c2, c3, c0, c2 >
```

That's not too informative, even on a system as small as this. The problem is that CHP doesn't know how to label your channels, so it assigns them arbitrary names. The list is actually a list of channel communications, but we need to make it more informative. For this, we must label our channels, using newChannel' instead of newChannel. You can use the former function whether tracing is being used or not, so it's fine to just use it throughout all your programs, regardless of whether you are planning to use tracing or not.

Now that we have turned tracing on and added channel labels, our code looks like this:

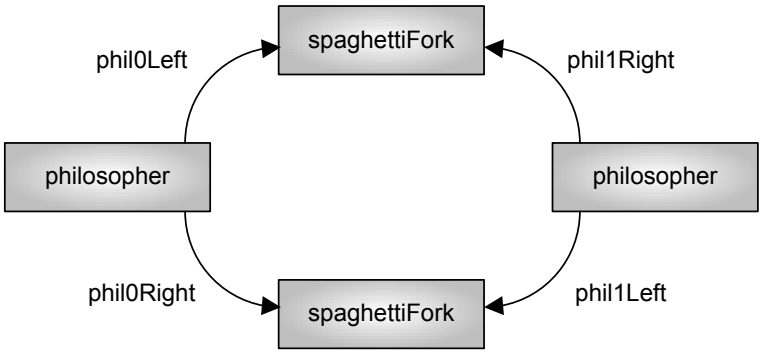
```
import Control.Concurrent.CHP
import Control.Concurrent.CHP.Traces
import Control.Monad
import Control.Monad.Trans
import System.Random

spaghettiFork :: Chanin () -> Chanin () -> CHP ()
spaghettiFork claimA claimB = forever
  ( (readChannel claimA >> readChannel claimA)
    <-> (readChannel claimB >> readChannel claimB) )

philosopher :: Chanout () -> Chanout () -> CHP ()
philosopher left right = forever (do
  randomDelay -- Thinking
  writeChannel left () >> writeChannel right ()
  randomDelay -- Eating
  writeChannel left () <||> writeChannel right ()
  )
  where
    randomDelay = liftIO (getStdRandom (randomR (0, 50000))) >>= waitFor
```

```
college :: CHP ()
college = do
  phil0Left <- newChannel' $ chanLabel "phil0Left"
  phil0Right <- newChannel' $ chanLabel "phil0Right"
  phil1Left <- newChannel' $ chanLabel "phil1Left"
  phil1Right <- newChannel' $ chanLabel "phil1Right"
  runParallel_
    [ philosopher (writer phil0Left) (writer phil0Right)
    , philosopher (writer phil1Left) (writer phil1Right)
    , spaghettiFork (reader phil0Left) (reader phil1Right)
    , spaghettiFork (reader phil1Left) (reader phil0Right) ]

main :: IO ()
main = runCHP_CSPTraceAndPrint college
```

Now we can run the program again and examine the trace. The end of the trace is the behaviour leading up to the deadlock, so that's the most interesting bit. Here is the end of the trace from one of my runs:

```
..., phil1Left, phil1Right, phil0Left, phil0Right, phil0Left,
phil0Right, phil1Left, phil0Left >
```

At the end, we can see that the last two actions were the philosophers sending a message to their left forks. Unfortunately, we can't tell if those are pick-up or put-down messages due to the design of our system, but it least gives us a clue to go and start tracking down the deadlock. In real situations, you may find traces helpful or you may not, but at least the facility is there in the library to be able to turn them on and see what is happening.



4 Termination

So far all our programs have run endlessly (with the exception of deadlock). In reality, you usually want your programs to finish, and in a nicer way than deadlock. Unfortunately this adds a bit of extra effort on the programmer's behalf, but it is mainly verbosity rather than complexity.

CHP programs are usually terminated using poison. The concept of poison is to set channels into a poisoned state, and any further attempt to use them causes a poison exception to be thrown. Thus, when a process wants to shut down the whole system, it poisons all its channels. The other processes get a poison exception when they try to access this channel, and in response they should poison all their channels. Thus, in a fully connected network, the poison should spread, and eventually all processes will have terminated (and all channels should be poisoned). Processes may also perform further actions when they catch a poison exception (such as closing files, closing GUI windows, etc).

Poison is different to an asynchronous exception because poison exceptions are only thrown either by the user (you) explicitly throwing them, or when you use a channel or barrier. So, for example, poison exceptions cannot occur during lifted IO actions. This should make poison easier to reason about.

We will now return to our earlier examples that read in characters and printed them. We will continue to read in characters and print them until a lower-case q is entered:

```
import Control.Concurrent.CHP
import qualified Control.Concurrent.CHP.Common as CHP
import Control.Concurrent.CHP.Console

main :: IO ()
main = runCHP_ (consoleProcess echoUntilQ)

echoUntilQ :: ConsoleChans -> CHP ()
echoUntilQ chans = idUntilQ (cStdin chans) (cStdout chans)

idUntilQ :: Chanin Char -> Chanout Char -> CHP ()
idUntilQ cin cout = (readChannel cin >>= inner)
                      `onPoisonTrap` end
  where
    inner :: Char -> CHP ()
    inner 'q' = end
    inner c = writeChannel cout c >> (readChannel cin >>= inner)

    end = poison cin >> poison cout
```

Our end action is to poison all our channels. We do this if we encounter a lower-case q, or in response to catching poison ourselves. It is possible that the console channels can be poisoned. For example, if there is an error on stdin (such as a broken pipe), the channel will become poisoned.

This poisoning works when composed with other processes. For example, if we now add a process that turns the characters into lower-case before our process, the poison still shuts that process down too:

```
import Control.Concurrent.CHP
import qualified Control.Concurrent.CHP.Common as CHP
import Control.Concurrent.CHP.Console
import Control.Concurrent.CHP.Connect
import Data.Char

main :: IO ()
main = runCHP_ (consoleProcess echoUntilQ)

echoUntilQ :: ConsoleChans -> CHP ()
echoUntilQ chans = do
  pipelineConnect [CHP.map toLower, idUntilQ] (cStdin chans) (cStdout chans)
  return ()

idUntilQ :: Chanin Char -> Chanout Char -> CHP ()
idUntilQ cin cout = (readChannel cin >>= inner)
                      `onPoisonTrap` end
  where
    inner :: Char -> CHP ()
    inner 'q' = end
    inner c = writeChannel cout c >> (readChannel cin >>= inner)

    end = poison cin >> poison cout
```

You can see that we did not need to alter idUntilQ at all to still terminate the process network. We just added another process and we retained the same termination behaviour.

4.1 Poison Propagation

4.1.1 Poison and Parallel Composition

When an exception is thrown during sequential code, the remainder of the code is aborted and execution jumps to the handler. With parallel code, the issue is not as clear. The semantics we use are that poison does not immediately abort any parallel-sibling processes, but that at the end of a parallel composition of processes, if any of the sub-processes died of untrapped poison, the parent rethrows the poison exception. So for example, with the following process snippet:

```
do x <- readChannel in_
   writeChannel out0 x <||> writeChannel out1 x
```

If attempting to write to channel out1 throws a poison exception, the write to channel out0 still goes ahead. Only when the write to channel out0 completes (or itself throws poison) does the parallel composition finish. At this point, because poison was thrown in one of the sub-processes, the parent (in this case, the **do** block) throws a poison exception.

There are two possible handlers for poison: onPoisonTrap and onPoisonRethrow. The first handler does not rethrow the poison (unless the handler does), whereas the latter handler always rethrows the poison. The rethrowing version is almost always more useful, but the trapping version can be used if you do not want the aforementioned behaviour:

```
do x <- readChannel in_
   (writeChannel out0 x `onPoisonTrap` poison out0)
     <||> (writeChannel out1 x `onPoisonTrap` poison out1)
```

This way, any poison thrown in the sub-processes will not cause the parent to also throw poison.

You may notice this funny behaviour of poisoning channels that you know must be poisoned already. This will not cause any problems, and in many cases (if not this simple case) trying to poison only the channels you need to is generally more effort than just poisoning all the channels and not worrying about which one threw the poison.

4.1.2 Poison and Choice

The semantics of choosing between channel reads/writes and barrier synchronisation is as follows with regard to poison. A poisoned channel or barrier is considered ready for reads, writes and synchronisations. If it is then chosen, a poison exception will be thrown as normal. For example, with this piece of code:

```
readChannel in0 <-> readChannel in1
```

If channel in1 is poisoned and in0 is ready to be read from, either a poison exception will be thrown, or in0 will be read from. If in0 was not ready, then the behaviour of this code would be to throw a poison exception.

4.1.3 Poison and Unsafe Usage

One thing that you must not do is to poison channels that are in use concurrently. So for example, do **NOT** do something like this to kill off your sibling process:

```
do x <- readChannel in_
   (writeChannel out0 x `onPoisonRethrow` (poison out0 >> poison out1))
     <||> (writeChannel out1 x `onPoisonRethrow` (poison out0 >> poison out1))
   -- Unsafe code!
```

This may be tempting, and it has the bonus benefit of killing your sibling early. But the algorithms underlying channels are built on the assumption that channels only have one writer and one reader, so the above code could produce some strange behaviour.

5 Extended Transactions

5.1 Extended Input

Sometimes a single synchronisation for a channel communication is insufficient. You might have a server listening for requests to alter a GUI. Another process, P, might want to send it an instruction to close the window. Once P has sent the message, it knows that the server has receieved it, but then (due to scheduling) it doesn't know how long it will be until the server processes this message, during which time the GUI may generate further events.

One method to be sure would be to get the server to send a message back to P. But the server may not have a channel to P, and the whole thing is a bit awkward. In this case, an extended read is an easier method.

In an extended read, the recipient takes the data, but is able to perform an *extended action* before freeing the writer; in our example, handling the request and closing the GUI window.

```
do handled <- extReadChannel c $ \req ->
     case req of
       CloseRequest ->
         do -- Handle close request
            return Nothing
       _ -> return Just req
```

This code will make the writer wait while it examines the request. If the request is a close request, it is handled and the extended block returns Nothing. If is any other request, Just the request is returned. This way, close requests are handled in an extended read, but all other requests can be handled later on. Of course, if you wanted, you could just handle all requests in an extended read.

5.2 Extended Output

Extended output is similar to extended input. Once the writer and reader have agreed to communicate, the writer may engage in an extended action, and it is the result of this action that is sent down the channel. Thus, if both the writer and reader perform an extended action, you know that the writer's (which gives the value to send) must happen before the reader's (which uses the value received).

Extended outputs are generally used less than extended inputs, because it is more often the case that you can just perform the extended action before engaging in the communication.

6 Barriers and Buffers

6.1 Barriers

The channels in CHP are synchronous, meaning that both processes must be present for the communication to take place. The processes synchronise and send data. With the exception of broadcast channels (introduced later), channel communications always involve exactly two processes.

Barriers are purely for synchronisation, and do not exchange any data. At any time, a barrier has a certain number of processes (zero or more) enrolled on it. When a process synchronises on a barrier, it must wait until all the other enrolled processes also synchronise.

6.2 Buffers

In some cases you do not want synchronous channels. To get asynchronous channels (where the writer can write to the channel without having to wait for the reader), you can use buffers from the Control.Concurrent.CHP.Buffers module. There are four different types of buffer:

1. Limited capacity plain-FIFO buffers. These are the most common buffers used. You provide a size for the buffer. Items are sent in order, and when the buffer is full, it no longer accepts any input.

2. Infinite capacity plain-FIFO buffers. These are as above, but without a limit. Thus, they always accept more input. Be aware that if a writer continues to write to one of these buffers faster than the reader can take it, the buffer will continue to fill up. This is effectively a memory leak. Use with caution; a thousand-sized FIFO buffer is safer than an infinite buffer, and will probably achieve what you want.

3. Overflowing FIFO buffer. This buffer acts like a limited-capacity plain-FIFO buffer, except that when it is full, it will continue to accept input and discard it. This is how buffers for user input (key presses and mouse clicks) usually work.

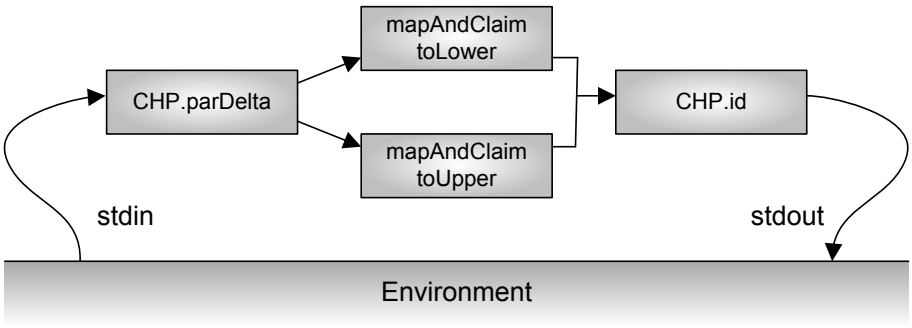
4. Overwriting FIFO buffer. This buffer also acts like a limited-capacity plain-FIFO buffer, except that when it is full, it adds the new item to the buffer and discards the previous oldest item. This is often used for inputs from sensors and similar, where newer values are more useful than older values and the writer updates faster than the reader can read.

Buffers in CHP are processes. To use a buffer between two processes (P and Q), you must connect P to the buffer with one channel, and then the buffer to Q with a second channel.

7 Sharing and Broadcasting

7.1 Shared Channels

A shared channel is one with either the writing and/or reading ends shared between several processes. To use this channel, the shared end(s) must be claimed, and can then be used like normal channels for the duration of the claim. You can think of shared channel-ends as having a mutex attached to them, which must be claimed.



For example, here is a process network that reads in a character, sends it to two processes that then both attempt to print upper and lower-case versions of it on a shared channel connected to stdout:

```
import Control.Concurrent.CHP
import qualified Control.Concurrent.CHP.Common as CHP
import Control.Concurrent.CHP.Console
import Control.Monad
import Data.Char

main :: IO ()
main = runCHP_ (consoleProcess dualEcho)

dualEcho :: ConsoleChans -> CHP ()
dualEcho chans = do (c, d) <- newChannels
                    shared <- anyToOneChannel
                    runParallel_
                      [ CHP.parDelta (cStdin chans) [writer c, writer d]
                      , CHP.id (reader shared) (cStdout chans)
                      , mapAndClaim toLower (reader c) (writer shared)
                      , mapAndClaim toUpper (reader d) (writer shared) ]

mapAndClaim :: (a -> b) -> Chanin a -> Shared Chanout b -> CHP ()
mapAndClaim f in_ sout = forever (do
  x <- readChannel in_
  claim sout ( flip writeChannel (f x)) )
```

Whether the lower-case or upper-case character is written first will be arbitrary. But the program is safe, because we are using shared channels. Using unshared channels here might result in undefined behaviour – if you want multiple concurrent processes to be able to write to or read from the same channel, you must use these explicitly shared channels.

7.2 Broadcast Channels

As discussed in the previous section, shared channels still involve one writer and one reader in a communication. Broadcast channels are part of a common pattern where there is one writer and many readers; the writer broadcasts the same value to all the readers in one communication.

If the number of readers is fixed, consider using the delta process from the Control.Concurrent.CHP.Common module instead, as it will be easier. Broadcast channels are mainly useful if you want a dynamic number of readers, or if you want to use extended writes/reads.

Broadcast channels are a mixture of channels and barriers. Readers must enroll on the channel, and can then engage in the communications. A communication only completes when the writer and all readers engage in the communication.

8 Laws of CHP

There are some useful properties in CHP that are worth stating here. The left-hand column of the tables below gives an algebraic-style equality, and the right-hand column states it in words. To represent sequence in CHP I've used the `>>` operator, but the same laws apply if you are using `>>=`. `p`, `q` and `r` are all processes. Unless otherwise stated, the rules apply for any `p`, `q` and `r` (that are at least valid monadic actions – bottom breaks most of these laws). Finally, note that these rules apply only to the concurrent *semantics*; the *types* of the left- and right-hand sides of the equalities may not match exactly.

8.1 Rules for Sequence and Parallel

`(p >> q) >> r == p >> (q >> r)`	Sequence is associative
`p >> skip == p`	SKIP is a right-unit of sequence
`stop >> p == stop`	Nothing follows STOP
`(p <||> q) <||> r == p <||> (q <||> r)`	Parallel is associative
`p <||> q == q <||> p`	Parallel is commutative
`p <||> skip == p`	SKIP is a unit (left- and right-, given the above law) of parallel
`(p <||> stop) >> q == p >> stop`	A parallel with a STOP in it will never complete
`runParallel [p] == p`	Parallel on a singleton doesn't matter

Note that SKIP is *not* a left-unit of sequence, because it will act differently when placed in a choice. The code `skip >> p` is always ready, and will execute `p` if chosen. However, `p` is only ready if its first action is ready.

8.2 Rules for Disjunctive Choice

All of the rules in this section only apply if all of p, q and r that are used in the law support choice.

`(p <-> q) <-> r == p <-> (q <-> r)`	Unprioritised choice is associative
`p <-> q == q <-> p`	Unprioritised choice is commutative
`p <-> stop == p`	STOP is a unit (left- and right-, given the above law) of unprioritised choice
`(p </> q) </> r == p </> (q </> r)`	Prioritised choice is associative
`p </> stop == p`	STOP is a right unit of prioritised choice
`stop </> p == p`	STOP is a left unit of prioritised choice
`skip </> p == skip`	SKIP is an always-ready left-zero of prioritised choice
`(skip >> q) </> p == q`	SKIP is always ready, and does nothing
`alt [p] == p`	Choice on a singleton doesn't matter
`priAlt [p] == p`	Choice on a singleton doesn't matter
`alt [p >> r, q >> r] == alt [p, q] >> r`	Sequence is right-distributive over choice

8.3 Rules for Conjunctive Choice

All of the rules in this section only apply if all of p, q and r that are used in the law support choice (or, where they are used as parameters to every, the rules that apply to parameters to that function).

`(p <&> q) <&> r == p <&> (q <&> r)`	Every is associative
`p <&> q == q <&> p`	Every is commutative
`stop <&> p == stop`	STOP is a (left- and right-, given the above law) zero of every
`skip <&> p == p`	SKIP is a (left- and right-, given the above law) unit of every
`every [p >> q, r >> s] == every [p, q] >> runParallel [r, s]`	
`every [skip >> p, skip >> q] == runParallel [p, q]`	Every is equivalent to parallel if all branches are immediately ready
`every [p] == p`	Every on a singleton doesn't matter (as long as p meets the conditions for every)

8.4 Rules for Poison

`p `onPoisonRethrow` (return ()) == p`	An empty rethrowing handler is irrelevant
`p `onPoisonRethrow` throwPoison ==p`	A rethrowing handler that only rethrows is irrelevant
`throwPoison `onPoisonRethrow` q == q >> throwPoison`	Throwing and rethrowing
`throwPoison `onPoisonTrap` p == p`	Semantics of poison trapping
`p `onPoisonTrap` throwPoison == p`	Semantics of poison trapping and rethrowing
`return x `onPoisonTrap` p == return x`	Return is a left-unit of poison handling
`return x `onPoisonRethrow` p == return x`	Return is a left-unit of poison handling
`throwPoison <||> p == p >> throwPoison`	Poison in a PAR is rethrown afterwards
`(p >> throwPoison) <||> q == (p <||> q) >> throwPoison`	Poison in a PAR is rethrown afterwards
`throwPoison >> p == throwPoison`	Poison breaks out of a sequence
`p `onPoisonTrap` (q >> throwPoison) == p `onPoisonRethrow` q`	Trapping poison and always rethrowing is the same as the rethrowing handler

9 Frequently Asked Questions

9.1 How can I use CHP to make my programs run faster by forcing computation to take place in a particular process?

Increased performance is perhaps the most common motivation for people to get into concurrency. While CHP does allow you to easily run IO actions concurrently, trying to get computation to run concurrently in Haskell is difficult due to its lazy evaluation. The same problem applies to all of Haskell – trying to force evaluation is surprisingly difficult. Any advice you find elsewhere on forcing evaluation applies in CHP, and this is used for the writeChannelStrict function.

9.2 Why is my application behaving strangely?

There are several misuses of the library that will not cause a run-time error but will instead cause odd behaviour:

- Using a non-shared channel end (Chanin and Chanout) in multiple processes concurrently. If you want to do something like this, you must use shared channel ends.
- Using a barrier (or clock) when you are not enrolled on it. You should only use a barrier end or clock when you are inside an enroll block but not inside a resign call.
- Enrolling on a barrier end or clock and then passing it to multiple concurrent processes to use. You must enroll once per process on each barrier/clock.

9.3 Why isn't my network shutting down when I poison it?

If you poison all your channels, there may be several reasons why this doesn't actually shut down all your processes:

1. Your program is trapping the poison somewhere and not propagating it, thus it isn't travelling up the process hierarchy.
2. Your process network is not fully connected via channels and/or barriers, so poison in one sub-network cannot spread to the other sub-network.
3. A process may be performing a long computation. Poison will only be seen when a process accesses a channel/barrier, so if the process is busy computing a value, it will not notice the poison. To solve this you will either just have to wait for the computation to complete or consider using asynchronous exceptions.

4. A process may be waiting for an external event, such as reading from a socket or file. The process will not see poison until the read completes. In this case you will almost certainly need to use asynchronous exceptions. Look in the code for the Control.Concurrent.CHP.Console module for an example.

It may seem like poison is not all that useful, given the last two points. Our experience is that poison is generally easier to work with than asynchronous exceptions, but you can always use a combination if that makes your job easier.

9.4 Why does my program crash, saying "Thread terminated with: registerDelay: requires -threaded"

In order to use timeouts as part of a choice, you must compile your program with the `-threaded` option.