Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

Communicating Haskell Processes: Composable Explicit Concurrency using Monads

Neil C.C. BROWN

Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF, England.

neil@twistedsquare.com

Abstract. Writing concurrent programs in languages that lack explicit support for concurrency can often be awkward and difficult. Haskell's monads provide a way to explicitly specify sequence and effects in a functional language, and monadic combinators allow composition of monadic actions, for example via parallelism and choice – two core aspects of Communicating Sequential Processes (CSP). We show how the use of these combinators, and being able to express processes as first-class types (monadic actions) allow for easy and elegant programming of process-oriented concurrency in a new CSP library for Haskell: Communicating Haskell Processes.

Keywords. Communicating Sequential Processes, CSP, Haskell, monads, explicit concurrency.

Introduction

Communicating Sequential Processes (CSP) is a formal algebra encompassing processes, events and synchronous channel communications [1]. CSP is the basis for the occam-π process-oriented programming language [2], and CSP libraries for a wealth of other languages, including: Java [3,4], C++ [5,6], C [7], Python [8] and the .NET languages [9,10].

This paper introduces Communicating Haskell Processes (CHP), a library for Haskell that is also based on CSP. The primary difference between the CHP library and its predecessors is that Haskell is a functional language rather than imperative. However one of Haskell's distinctive features is monads, which allow control over how operations are sequenced and thus allow for imperative-style programming [11].

Haskell's monads and CSP's algebra share an elegant feature: ease of composition. Monadic combinators have been used in the past to implement sequence (and iteration), exceptions, concurrency and choice – CSP allows composition of processes in similar ways. CHP thus marries CSP's process composition with Haskell's monad composition. This allows a level of composition across function/procedure boundaries that is not present in any other CSP-based language or library.

Using our choice operator `<->` and our parallelism operator `<||>` we can write a process that reads from either of its two input channels and sends the value down both its output channels in parallel:

```
proc (in0, in1) (out0, out1)
  = do x <- readChannel in0 <-> readChannel in1
       writeChannel out0 x <||> writeChannel out1 x
```

We are then able to compose this process again using choice, sequence, parallelism or iteration[1]. Taking p and q to be processes, and using the notation from Hoare's book [1], the compositions are as follows:

```
p <-> q    -- choice, CSP: P|Q
p <||> q   -- parallelism, CSP: P||Q
p >> q     -- sequence, CSP: P ; Q
forever p  -- iteration, CSP: *P
  where
    p = proc (in0, in1) (out0, out1)
    q = proc (in2, in3) (out2, out3)
```

We will explain the necessary background in Haskell required to be able to understand the examples and concepts in the paper (section 1), and then examine each type of combinator in CHP:

- Sequence (section 2),
- Parallelism (section 3),
- Exception (poison) handling (section 5),
- Choice, often referred to as ALTing (section 6), and
- Iteration constructs (section 7).

We go on to examine building simple processes with several different combinators, and demonstrate the closeness of the Haskell code to the CSP algebra as well as the use of choice over outputs (section 8).

We will also examine how Haskell makes it easy to wire up networks of processes and channels (section 4). Brief details on the implementation of CHP using Software Transactional Memory (section 9) are also provided, as is discussion of related work (section 10).

1. Background – Haskell

Haskell is a statically-typed lazily-evaluated functional programming language. This section provides the necessary background information on Haskell to understand this paper. We explain how to read Haskell types and some basic syntax. Further details are supplied where necessary throughout the paper.

1.1. Types

We precede every Haskell function with its type signature. The format for this is:

```
functionName :: typeOfParameter1 -> typeOfParameter2 -> ... -> resultType
```

The statement $x :: t$ should be read as "x has type t". Each parameter is separated by an arrow, so that $a \texttt{->} b$ is "a function that takes a parameter of type a, and returns type b." Any type beginning with an upper-case letter is a specific type, whereas any type beginning a lower-case letter (by convention, single lower-case letters are used) is a parameterised type. This is the type of the *map* function, that applies a transformation to a list:

```
map :: (a -> b) -> [a] -> [b]
```

This takes a function transforming a value of type a into type b, and maps from a list of type a to the corresponding list of type b. The a and b types can be the same (and frequently are).

[1]Note that in CHP, unlike the formal algebra of CSP, there is no difference in type between an event and a process.

1.2. Functions

A Haskell function definition simply consists of the function name, followed by a label for each of its parameters, then an equals sign and the definition of the function. For example, this function adds the squares of two numbers:

```
addSquares :: Int -> Int -> Int
addSquares x y = (x * x) + (y * y)
```

There is also an infix notation for functions. The prefix function call *addSquares* 3 5 may also be written 3 `` `addSquares` `` 5; the infix form is created by using backquotes around the function name.

2. Sequential Composition

A monad type in Haskell defines how to compose operations in sequence. Examples of common monads include state monads, error-handling monads (that can short-circuit computation) and the *IO* monad for dealing with input and output.

We will not fully explain the underlying mechanics of monads here, but thankfully Haskell provides a **do** notation that should render our programs readable to those not familiar with Haskell and monads. Consecutive lines in a **do** block are sequential monadic operations. A **do** block follows standard Haskell indentation rules, lasting until the indentation decreases (similar to occam-π's indentation rules), or the end of the expression (e.g. a closing parenthesis that began before the **do** block). The output values of monad operations can be labelled using the **<-** notation. For example, this program reads in a character and writes it out again twice (then finishes):

```
main :: IO ()
main = do c <- getChar
          putChar c
          putChar c
```

The type of *getChar* is *IO Char*. This indicates that the function is a monadic action in the *IO* monad that returns a value of type *Char*. The type of the *main* function is *IO* (); the Haskell unit-type ("()", which can be read as an empty tuple) is used to indicate that there is no useful return value, and is very common. Note that there is no difference in types between a **do** block and a single monadic action; the former simply composes several of the latter in sequence, and either or neither may give back a useful value.

Recursion is possible in **do** blocks. We present here the standard *forever* function that repeats a monadic action[2]:

```
forever :: Monad m => m a -> m ()
forever action = do action
                    forever action
```

The type signature of *forever* states that for any monad *m*, *forever* takes a monadic action that returns some value and gives back a monadic action that returns no value. In this instance, the implementation of the function is easier to follow than its type.

CHP defines the *CHP* monad, in which all of its operations take place. Two elementary monadic operations are *readChannel*, which reads from a given channel, and *writeChannel* which writes a given value (second parameter) to a given channel (first parameter). Thus, we can

[2]As we will see later on, errors such as poison can break out of a *forever* block.

write the classic *id* process that continually reads from one channel and writes to another, omitting support for poison (see section 5):

```
idNoPoison :: Chanin a -> Chanout a -> CHP ()
idNoPoison input output
  = forever (do x <- readChannel input
                writeChannel output x
            )
```

CHP uses the idea of channel-ends, as most CSP frameworks do. Both channels carry type *a*, which may be any type – but the type carried by the input channel must match the type carried by the output channel; otherwise the compiler will give a type-checking error. Because **in** is a reserved word in Haskell, we use *input* as a variable name instead.

The process could also be written recursively:

```
idNoPoison input output
  = do x <- readChannel input
       writeChannel output x
       idNoPoison input output
```

Where possible, we prefer the *forever* idiom, both to shorten definitions and also because it makes clear that no state is carried between iterations of the process. Iteration constructs are examined in more detail in section 7.

The channels in CHP are synchronous. This means that when a process attempts to write to a channel, it must wait until the reader arrives and takes the data before the write operation will complete. Synchronous channels are used in all CSP-based frameworks, and this is one difference between CHP and, for example, Erlang.

3. Parallel Composition

Processes can be composed in parallel using the runParallel function:

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

Its type can be read as follows: *runParallel* takes a list of processes that return values of type *a*, and composes them into a single process that returns a list of values of type *a*. It gets these values by running the processes in parallel and waiting for them all to complete.

We also provide an operator, `<||>`, such that *p* `<||>` *q* is semantically identical to *runParallel* [*p*, *q*]. The types are slightly different however: the operator returns a pair of values (which can have different types), whereas the function returns a list of identically-typed values. A sum type could be used if heterogenous return types are required.

In contrast to other CSP frameworks, parallelism here supports returning the result values of the sub-processes. This was primarily out of necessity; if we only had the *runParallel_* function that does not return the output of the sub-processes[3]:

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

Then there would be no easy way to return any values from the parallel processes. Assignment to variables cannot be used because there is no assignment in functional languages, and values could not be communicated back to the parent process because it would be waiting for the sub-processes to finish (and hence deadlock would ensue).

[3]The underscore-suffix on a monadic function is a Haskell convention indicating that the output is discarded.

3.1. Forking with Monads

In occam-π, it is possible to use a `FORKING` block to dynamically start new processes. At the end of the `FORKING` block, the completion of all the processes is waited for. This idea was carried across to C++CSP2, using the scope of objects to enforce a similar rule – although with an added danger because of the ordering of object destruction [5].

We can again implement this concept in CHP using monads. There is no danger of object destruction, as CHP channels are garbage-collected only when they are no longer in use. We declare a forking monad[4] that gives us the following functions:

```
forking :: ForkingCHP a -> CHP a
fork :: CHP () -> ForkingCHP ()
```

The *forking* function takes a monadic *ForkingCHP* block and runs it, waiting for all the processes at the end before returning the output. The *fork* function forks off the given *CHP* process from inside the *ForkingCHP* block. Unlike our normal parallelism operators described previously, there is no way for a forked process to directly return a value. Forked processes that need to pass back a value to the parent process may do so using a channel communication.

4. Channel Wiring

In occam-π, **PROC**edures are not first-class types. A block of monadic code is a first-class type in Haskell, and can be passed around, as we have already seen with our combinators. We can also pass around functions that yield a monadic item: in CHP terms, this is a process that still needs parameters.

We can take advantage of this to provide functions for standard wiring idioms. An obvious example is wiring a list of processes into a pipeline:

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

This function takes a list of processes that require a reading- and writing-end of a channel carrying type *a*. The *pipeline* function also takes the channel ends to be used at the very beginning and end of the pipeline, and returns the parallel composition of the processes in the pipeline.

The *pipeline* function can be defined in several ways. Here we use an elegant recursive definition of a helper function *wirePipeline* that wires up all the processes and returns them in a list:

```
pipeline procs input output
  = do wiredProcs <- wirePipeline procs input output
       runParallel wiredProcs

wirePipeline :: [Chanin a -> Chanout a -> CHP b]
    -> Chanin a -> Chanout a -> CHP [CHP b]
wirePipeline [p] input output = return [p input output]
wirePipeline (p : ps) input output
  = do c <- newChannel
       rest <- wirePipeline ps (reader c) out
       return ((p input (writer c)) : rest)
```

The first line of *wirePipeline* is a base case, and matches a single-process list. The remaining lines are the recursive step, with a pattern-match to decompose the process list into its

[4]Technically, this is a monad transformer that composes the *CHP* monad with a *ForkingT* monad transformer.

head *p* (a single process) and the remainder of the list *ps* (a list of processes). The : constructor is used again in the last line to join an item onto the head of the list *rest*.

Here is an example of using the function:

```
fifoBuffer :: Int -> Chanin a -> Chanout a -> CHP ()
fifoBuffer n input output
  = do pipeline (replicate n idProcess) input output
       return ()
```

The *replicate* function takes a replication count and a single item, and returns a list containing the item repeated that many times.

We can also easily define a function for wiring up a cycle of processes, by making the two ends of the pipeline use the same channel:

```
cycle :: [Chanin a -> Chanout a -> CHP b] -> CHP [b]
cycle procs = do c <- newChannel
                 wiredProcs <- wirePipeline (reader chan) (writer chan) procs
                 runParallel wiredProcs
```

It would not be difficult to make general functions for wiring up other common idioms.

4.1. Channel Type Inference

For channels there is a bijective mapping between the two channel-end types and the channel implementation. A *Chanin* and *Shared Chanout* are associated with an any-to-one channel. A *Chanin* and *Chanout* are associated with a one-to-one channel.

This means that if the Haskell type-checker (which uses type inference) knows either the two channel-end types or the channel type, it can infer the other. Typically this is used to allocate a channel using the *newChannel* function, and have the type-checker figure out what type the channel needs to be, based on what processes the ends are passed to. Programmers who prefer to be explicit can still use individual *oneToOneChannel* functions.

The difference in types between the various channel-ends prevents channel-ends being used incorrectly (for example, using a shared channel-end without claiming it), so there is no possibility for error, and it also makes the code simpler. No other CSP framework or language has this capability, because of the lack of such type inference.

Channels do not need to be explicitly destroyed in CHP – instead, they will be garbage-collected when no longer in use (using standard Haskell mechanisms). This removes any worry about correctly nesting the scope of channels.

5. Poison and Exception Handling

Poison is a technique for safely shutting down a process network, without inviting deadlock or forcefully aborting processes [12,13,14]. A channel can either be in a normal operating state, or it can be poisoned. Any attempt to read or write on a poisoned channel will result in a poison exception being thrown.

Poison propagates throughout a network as follows. When a process catches a poison exception, it poisons all its channel-ends. Thus its neighbours (according to channel connections in a process graph) will also get poison thrown, and they will do the same, until all channels in a process network have been poisoned. Once processes have poisoned their channels, they shut down, and thus the process network terminates. This mechanism has previously been incorporated into C++CSP, JCSP and others.

Our discussion here is centred around poison, but the ideas should generalise to any notion of exceptions in process-oriented programs. Haskell supports exceptions in three ways:

in pure code, in the *IO* monad, and in special error monads. The latter approach is the neatest solution. Thus we allow poison exceptions to occur in our *CHP* monad.

The *onPoisonTrap* function may be used (typically infix) to trap and handle poison. For example, here is the identity process with poison handling:

```
idProcess :: Chanin a -> Chanout a -> CHP ()
idProcess input output
  = (forever (do x <- readChannel input
                 writeChannel output x
             )
    ) `onPoisonTrap` (do poison input
                         poison output)
```

It is important that the *forever* combinator is used inside the body of the poison, e.g. (*forever* ...) `*onPoisonTrap*`(...). If the process was composed as *forever* ((...) `*onPoisonTrap*` (...)) then it would form an infinite loop, forever catching the poison, handling it, and looping again.

We also add another poison handler (*onPoisonRethrow*) that *always* rethrows after the handler has finished. This handler can be used either inside or outside of the *forever* combinator, without encountering the infinite loop problem. The use of this function is further examined in section 8.

5.1. Parallelism and Poison

One problem with poison has been deciding on the semantics of poison and parallel composition. In short, when p and q are composed in parallel and p exits due to poison, what should happen to q, and what should happen to their parent process?

Forcibly killing off q is an ugly solution that goes against the main principle of poison (allowing for controlled termination). Doing nothing at all is an odd solution, because the parent will not know whether its sub-processes terminated successfully or died because of poison. Consider the following process:

```
delta2 :: Chanin a -> Chanout a -> Chanout a -> CHP ()
delta2 input output0 output1
  = forever (do x <- readChannel input
                writeChannel output0 x <|> writeChannel output1 x)
    `onPoisonRethrow` (do poison input
                          poison output0
                          poison output1)
```

If the parent is never notified about its subprocesses dying of poison, the *delta2* process would continue running if one, or even *both*, of its output channels was poisoned, because the poison exception would be masked by the parallel composition.

The semantics we have chosen are straight-forward. The parent process spawns off all the sub-processes, and waits for them *all* to complete, either normally (no poison) or abnormally (with poison). Once they have all completed, if *any* of the sub-processes exited in a state of exception (with poison), the *runParallel* function (or similar) throws a poison exception in the parent process.

This solution corresponds to the ideas in Hilderink's CSP exception operator [13] and Hoare's concurrent flowcharts [15]. It maintains the associativity of PAR; the following two lines are semantically equivalent:

```
runParallel [runParallel [p, q], r]
runParallel [p, runParallel [q, r]]
```

The other preserved useful property is that running one process in parallel is the same as running the process directly: *runParallel* [*p*] is semantically identical to *p*. Commutativity of PAR is also maintained. It should be noted that the types differ slightly between all the aforementioned examples, but our concern here is only with semantics.

6. Composition using Choice – Alts and Implicit Guards

In occam-π, it is possible to choose between several events using the *ALT* construct, or *PRI ALT* which gives its guards descending priority. Each option has a guard, followed by a body:

```
PRI ALT
  c ? x -- input guard
    d ! x -- body (output)
  SKIP -- guard
    d ! 0 -- body (output)
```

SKIP is a guard that is always ready. Thus the above code checks the channel `c` to see if input is waiting. If some input is waiting it is read and sent out on channel `d`, otherwise the **SKIP** guard is chosen and the value `0` is sent instead.

Frameworks such as JCSP and C++CSP2 have translated the ALT into a construct that takes an array of guards and returns an integer denoting which guard is ready. The program then follows this up by acting on the guard. For example, the occam-π code above would be written as follows in C++CSP2:

```
Alternative alt (c.guard()) (new SkipGuard);
switch (alt.priSelect()) {
   case 0: {
      c >> x;
      d << x;
   } break;
   case 1: {
      d << 0;
   } break;
}
```

Note how the input must be performed separately from the guard. This was a design decision (taken from JCSP) to easily allow either normal or extended input on the channel after it has been found to be ready by the `Alternative` construct.

6.1. Implicit Guards

In CHP, we are able to integrate the guard and its body. We can write, similar to CSP:

```
alt [ do x <- readChannel c
        writeChannel c x
    , do skip
        writeChannel d x ]
```

We say that choice is implicitly available here, because the first action in each body supports choice – such actions are *skip*, a channel read or write (normal or extended), a wait action, a barrier synchronisation or another *alt* (which allows alts to be nested). This is achieved by constructing a special monad that allows us to keep track of the first action (and a hidden associated guard) in any given monadic action. It is possible to supply only one action, such as *skip*, to the alt without a **do** block if no body is required.

In addition to the *alt* and *priAlt* functions, we supply corresponding operators: **<->** for choice without priority, and **</>** for choice with left-bias. That is, the expression p **<->**q is identical to *alt* [p, q] and the expression p **</>**q is identical to *priAlt* [p, q]. The operators are associative. The functions also have the property that *alt* [p], *priAlt* [p] and p are all identical, provided that p supports choice (otherwise a run-time error will result). The duality between choice (a sum of processes) and parallelism (a product of processes) is clearer in CHP than it is in occam-π.

An eternal *fairAlt* that cycles priority between the guards is also easy to construct – we choose to represent it here with recursion:

```
fairAlt :: [CHP a] -> CHP ()
fairAlt (g:gs) = do priAlt (g:gs)
                    fairAlt (gs ++ [g])
```

6.2. Composition of ALTs

ALTs in occam-π are composable to a certain degree. Directly nested `ALT`s are possible:

```
ALT
  ALT
    c ? x
      d ! x
    e ? x
      d ! x
  tim ? AFTER t
    d ! 0
```

The above code chooses between inputs on `c` and `e`, and waiting for a timeout (for the time `t` to occur). The body of each guard is an output on channel `d`.

However, you cannot pull out guards into a separate procedure:

```
PROC alt.over.all (CHAN INT c?, CHAN INT e?, CHAN INT d!)
  ALT
    c ? x
      d ! x
    e ? x
      d ! x
:
ALT
  alt.over.all (c, e, d)
  tim ? AFTER t
    d ! 0
```

This was a design decision, taken in classical occam, to treat the guards differently. In our Haskell implementation, we only require the first action of any given monadic action to support choice. Since an *alt* supports choice, we can nest them – regardless of function boundaries. Therefore this *is* valid in CHP:

```
altOverAll :: Chanin Int -> Chanin Int -> Chanout Int -> CHP ()
altOverAll c e d = alt [ do x <- readChannel c
                           writeChannel d x
                       , do x <- readChannel e
                           writeChannel d x ]

alt [ altOverAll c e d
    , do waitUntil t
         writeChannel d 0 ]
```

This new composability overcomes one of the shortcomings that Reppy pointed out in the ALT construct when he developed Concurrent ML [16]. He noted that function composition was incompatible with choice. With implicit guards in CHP, this is not the case. This idea would also be possible to build into occam-π or Rain [17], where the presence of choice could be checked at compile-time. The compiler could eliminate the run-time errors that can occur in CHP if you try to choose between something that does not support choice (for example, poisoning a channel).

A further example of using choice can be seen in section 8.2.

7. Iteration

Most processes have repeating behaviour. It is very common to see `WHILE some.condition` or even `WHILE TRUE` at the beginning of occam-π processes. The latter can be expressed with the Haskell combinator *forever*, and can be broken out of using poison or other monadic exception mechanisms.

The *forever* combinator repeatedly runs the same block of code. It does not support easily stopping on a certain condition, or retaining any idea of state between subsequent runs of the same block. For many small processes, such as the identity process, this is acceptable. To demonstrate two different ways state can be implemented in the presence of iteration, we will use the example of a *runningTotal* process that continually reads in numbers and outputs the current total after each one.

The first obvious mechanism is to use recursion. We define the *runningTotal* process as simply setting off another inner process[5]:

```
runningTotal :: Chanin Int -> Chanout Int -> CHP ()
runningTotal input output
  = runningTotal' 0 `onPoisonRethrow` (do poison input
                                          poison output)
  where
    runningTotal' :: Int -> CHP ()
    runningTotal' prevTotal = do x <- readChannel input
                                 let newTotal = prevTotal + x
                                 writeChannel output newTotal
                                 runningTotal' newTotal
```

We take advantage of the scoping of Haskell's **where** clause; *input* and *output* are in scope for *runningTotal'*.

This recursion can get messy if many variables need to be passed to the recursive call. Haskell's state monad-transformer provides another alternative. The state monad-transformer provides *get* and *put* monadic functions for dealing with the state, and a whole block can be evaluated with a given state[6]:

```
runningTotal input output
  = runWithState 0 runningTotal' `onPoisonRethrow` (do poison input
                                                       poison output)
  where
    runningTotal' = forever (do x <- readChannel input
                                prevTotal <- get
                                let newTotal = prevTotal + x
                                put newTotal
                                writeChannel output newTotal
                            )
```

[5]We use the suffix ' here: a valid character in Haskell identifiers often used for this purpose.

[6]Technically, our *runWithState* function here is defined as *flip evalStateT*.

With the state monad, the reads and writes to and from the state can be placed more appropriately throughout the code block, rather than having to name all the variables at the start of the function, and pass them all again at the end of the function.

With the recursive method it is possible to control the looping by providing a base case, whereas the state monad has no support for this. However, it is possible to support some more easily controlled looping in Haskell, using yet another monad.

Inspired by Ian East's revival of the DO-WHILE-DO loop (transmuted into his Honeysuckle programming language as repeat-while [18]), CHP offers a loop-while construct using another monad-transformer.

The *loop* function takes a block and executes it. Inside this block may be one or several (or none, to loop forever) *while* statements. As an example of its use, here is a modified identity process that stops (between the input and output) when a certain target value is seen:

```
idUntil :: a -> Chanin a -> Chanout a -> CHP ()
idUntil target input output
  = loop (do x <- readChannel input
             while (x /= target)
             writeChannel output x)
```

This particular process would not be as elegantly expressed using recursion or the state monad. It is possible to combine this looping monad with the state monad.

8. Further Composition

We have now presented five types of composition: sequence, parallelism, choice, exception (poison) handling and iteration (cyclic sequence). All of these compositions can cross function boundaries in Haskell. We first show some general examples of all these types of composition, and also give an example of practical uses while implementing buffers.

8.1. General Composition

In this section we show how to compose several very simple processes. Each process is given both in Haskell code and using CSP notation (with parameters omitted). We borrow Hilderink's exception-handling operator [13]: $P \overrightarrow{\triangle} Q$ behaves as P, but if P throws a poison exception it behaves instead like Q. We also invent a process, $\Omega(..)$ that poisons all channels passed to it, and THROW that throws a poison exception.

Generally, the smallest composite process in process-oriented programming is the identity process – but this already contains two compositions (sequence and iteration), and three in frameworks with poison such as C++CSP2. We start here with a *forward* process that is one iteration of the identity process:

```
-- CSP: forward = input?x --> output!x --> SKIP
forward :: Chanin a -> Chanout a -> CHP ()
forward input output = do x <- readChannel input
                          writeChannel output x
```

This can then be composed into several other processes:

```
-- CSP: forwardForever = *forward
forwardForever :: Chanin a -> Chanout a -> CHP ()
forwardForever input output = forever (forward input output)
```

$-- \text{ CSP: forwardSealed} = \text{ forward} \overrightarrow{\triangle} (\Omega(input, output))$

```
forwardSealed :: Chanin a -> Chanout a -> CHP ()
forwardSealed input output
  = (forward input output)
    `onPoisonTrap` (do poison input
                       poison output)

-- CSP: forwardRethrow = forward ⟶△ (Ω(input, output) ; THROW)
forwardRethrow :: Chanin a -> Chanout a -> CHP ()
forwardRethrow input output
  = (forward input output)
    `onPoisonRethrow` (do poison input
                          poison output)
```

We include both of the latter two processes so that we can demonstrate their relative composability below. Consider these further-composed processes:

```
-- CSP: id1 = forwardForever ⟶△ (Ω(input, output))
id1 :: Chanin a -> Chanout a -> CHP ()
id1 input output
  = (forwardForever input output)
    `onPoisonTrap` (do poison input
                       poison output)

-- CSP: id2 = forwardForever ⟶△ (Ω(input, output) ; THROW)
id2 :: Chanin a -> Chanout a -> CHP ()
id2 input output
  = (forwardForever input output)
    `onPoisonRethrow` (do poison input
                          poison output)

-- CSP: id3 = *forwardSealed
id3 :: Chanin a -> Chanout a -> CHP ()
id3 input output = forever (forwardSealed input output)

-- CSP: id4 = *forwardRethrow
id4 :: Chanin a -> Chanout a -> CHP ()
id4 input output = forever (forwardWithRethrow input output)

-- CSP: id5 = (*forwardRethrow) ⟶△ SKIP
id5 :: Chanin a -> Chanout a -> CHP ()
id5 input output
  = (forever (forwardWithRethrow input output))
    `onPoisonTrap` skip
```

Intuitively, *id2* is semantically identical to *id4*, and *id1* is semantically identical to *id5*; proving this is left as an exercise for the reader. We prefer *id4* and *id5*, which locate the poison-handling as close as possible in the composition to the channel-events. Processes *id1* and *id5* are *not* identical to *id3*, as the latter will never terminate, even if its channels are poisoned.

We can see that, pragmatically, the *forwardWithRethrow* function was much more composable than the *forwardSealed* function. The implication in turn is that *id2* and *id4* will prove more composable than their "sealed" counterparts, *id1* and *id5* – and we believe that in practice, processes involving poison should always rethrow in order to make them more composable.

Our example shows that simple CHP programs can be reasoned about. The documentation supplied with CHP contains many useful laws to support such reasoning, for example:

```
runParallel [p] == p
throwPoison >> p == throwPoison
(p >> throwPoison) <|||> q == (p <|||> q) >> throwPoison
```

The >> operator represents sequence. These laws are similar to the laws of occam presented by Roscoe and Hoare [19].

There is a close correspondence between our extended CSP and the CHP code, especially in the presence of composition. Even if the user of the library knows nothing about CSP, the CHP compositions have inherited the beauty, and some of the reasoning power, of the original CSP calculus.

8.2. Output Guards and Buffers

It is sometimes desirable to introduce buffering between two processes, rather than having direct synchronous communication. A pipeline of N identity processes forms a limited capacity First-In First-Out (FIFO) buffer of size N. Sometimes, more complex buffering is required, such as overwriting buffers. An overwriting buffer also provides a limited capacity FIFO buffer, but when the buffer is full, it continues to accept new data and overwrites the oldest value in the buffer.

It is not possible in occam-π to define an overwriting buffer process with a single input and single output channel. Consider the case of a size-one overwriting buffer process. The process begins by reading in an item of data. If it subsequently writes out the data, it is committed to the write because all writes must be committed to in occam-π . Another piece of data arriving cannot affect this write, and thus the value cannot be overwritten. If the process does not send out the data, it is breaking the semantics of the buffer that the data should be available to be read.

Many frameworks, such as JCSP and C++CSP2 solve this problem by supplied buffered *channels* that encapsulate this behaviour. This complicates the API for channels. In CHP we allow choice over outputs (which no previous framework has done) and we can use this to construct overwriting buffer processes.

Our CHP overwriting buffer process does not have to commit to the output. It therefore chooses between reading a new data item in and writing out an item from the buffer. This allows us to express the correct behaviour:

```
overwritingBuffer :: Int -> Chanin a -> Chanout a -> CHP ()
overwritingBuffer n input output
  = ( overwritingBuffer' [] ) `onPoisonRethrow` (do poison input
                                                    poison output)
  where
    overwritingBuffer' :: [a] -> CHP ()
    overwritingBuffer' s | null s          = takeIn
                         | n == length s   = takeInReplace <-> sendOut
                         | otherwise       = takeIn <-> sendOut
      where
        takeIn        = do x <- readChannel input
                           over (s ++ [x])
        takeInReplace = do x <- readChannel input
                           over (tail s ++ [x])
        sendOut       = do writeChannel output (head s)
                           over (tail s)
```

We define our buffer as simply setting off an inner process with an empty list[7]. The inner process takes a list of items in the buffer. It then has three guards (indicated by the "|"

[7]In our real buffers we use a data structure with $O(1)$ append, but we use lists here for simplicity.

symbol). The first guard that evaluates to *True* is chosen. These are checked in sequential order based on the function arguments, and should not be confused with guards used for alting.

The first guard checks if the buffer is currently empty. If so, the only action should be to take in new data. If the buffer is full (the second guard), the process chooses between taking in new data (and overwriting the oldest existing value) or sending out an item of data. If the buffer is neither empty nor full (the last guard), the process chooses between taking in new data (and adding it to the buffer) and sending out an item of data.

The process behaviours are at the end of the code above. The *head* function picks the first item from a list, and the *tail* function is all of the list *except* for the first item. They use recursion to provide iteration.

Buffers can be written in CHP quite easily because choice is available on channel writes as well as reads; in other frameworks, choice was only available on channel reads because there was not a fast and safe implementation until recently (see the next section).

9. Implementation

CHP's channels and barriers are built on top of Haskell's Software Transactional Memory (STM) library [20]. Channels offer choice at both ends, using Welch's idea for symmetric channels [3]: both ends synchronise on an alting barrier (a multiway synchronisation with choice) then proceed with the communication. The alting barriers use an STM implementation based on the "oracle" mechanism [21,22].

The only way to start an explicit new concurrent process in Haskell is with the *forkIO* function. It takes a process and starts running it. It is a "fork-and-forget" function, providing no means to wait for the completion of the forked process. Thus we implement our parallel operators (that do wait for the completion of processes and return their outputs) by simply having the processes write their result to a shared channel when they complete. The parallel operator thus forks off N processes, then reads from the channel N times, sorts the results and returns them.

10. Related Work

The idea of combining functional programming and process-oriented concurrency is now quite old; Erlang is an obvious successful example [23]. Erlang is based on the actor model [24], and as such has asynchronous communication and untyped addressed mailboxes instead of CSP's synchronous communication and anonymous typed channels. Erlang-style communication has also already been implemented in Haskell [25]. Erlang has explicit sequencing in the language and strict evaluation, in contrast to Haskell's monads and lazy evaluation.

Combining Haskell with some ideas from occam has been done before in Haskell# ("Haskell-hash") [26]. Haskell# began with explicit concurrency and has developed into a separation of computation (expressed in Haskell) and topology/communication (expressed in a separate Haskell Configuration Language), which contrasts to CHP's standing as a Haskell library.

The CHP library is built on top of Software Transactional Memory (STM), a Haskell solution to the problem of explicitly concurrent programming [20]. Most previous concurrent Haskell frameworks were built on top of an older system of shared mutable variables [27]. One of the main advantages of STM over the shared mutable variables is that STM naturally supports choice, which is key to CSP programs, and allows for better composability.

STM allows multiple transactions in the STM monad to be composed via sequence or choice into a single transaction. Parallelism is handled externally to the STM mechanism. STM also allows full choice between sequentially composed transactions; both transactions

in sequence must succeed for it to be chosen. This is possible because STM only commits transactions when they are successful. It does not make it possible to rollback a change which can be viewed by another process.

Concurrent ML is another obvious predecessor to CHP [16], and in turn was an influence on STM. It had the notion of an event, and choice between events. Events could be derived from channels by supplying a destination/source for a read/write. Events could be composed via choice, and independently executed later on. This corresponds in CHP to forming a monadic action using choice, and separately executing the action.

Concurrent ML did not feature the idea of poison or anything similar. Poison could probably be built on top of Concurrent ML's primitives, but the idea of poison requires careful thought about the semantics of composing it via parallelism and iteration. ML permits side effects and uses eager evaluation; Haskell's purity and lazy evaluation may offer opportunity for safer programming and different programming methods.

STM and Concurrent ML share some of the elegance in composition of CHP, but we find that CSP is a more comprehensible programming model for explicit concurrency, and in addition provides a formal basis for reasoning about programs.

11. Conclusions

We have presented a library that makes the CSP-based process-oriented programming model available in Haskell, and have shown how natural, powerful and elegant its composition of elements is. Code in CHP can have a strong correspondence to the original CSP algebra, with identical semantics.

We believe that CHP is just as powerful as explicitly concurrent languages such as occam-π for writing concurrent programs, and that parallelism, choice and composing process networks with formulaic wiring are all easy in CHP, with the added feature of support for poison. The occam-π implementation retains a memory and speed advantage over Haskell, the former being able to allocate as little as 32 bytes per process and at least an order of magnitude faster than the latter. CHP contains many of occam-π's other features [2] such as barriers, explicitly-claimed shared channels, extended input, as well as extended output that is not currently present in occam-π.

The examples throughout this paper have also demonstrated how recursion can be used with the CSP model to create small elegant understandable programs. Recursion has not often been used in CSP implementations in the past – previous versions of occam did not support recursion, and other frameworks shy away from it, as non-optimised recursion can lead to needing a large amount of stack memory. In most frameworks, each process requires a separate stack, so efforts are made to minimise the use of the stack.

We hope that this library will prove interesting and useful to a variety of people, including: CSP theoreticians looking for a suitable development platform, process-oriented programmers who wish to use Haskell, and Haskell programmers who want a simple but powerful explicit concurrency framework.

11.1. Future Work

There is currently work at Kent ongoing to write a new occam-π compiler, Tock, in Haskell. One possibility for future work would be to combine Tock with the CHP library in order to produce an occam-π interpreter written in Haskell.

The performance of CHP could also be investigated and benchmarked. Haskell's lazy evaluation means that more thought is required to achieve speed-up through concurrency, so CHP needs to be exercised on parallel processing tasks for this to be tested.

11.2. Practical Details

Like many Haskell programs, the CHP library uses features that are not part of the latest Haskell standard (Haskell 98), but are likely to be part of the next Haskell standard (currently entitled Haskell-Prime). It runs under the latest branch (6.8) of the most popular Haskell compiler, GHC. The library has now been released, and more details on obtaining it can be found at: `http://www.cs.kent.ac.uk/projects/ofa/chp/`.

Acknowledgements

Many thanks are due to Adam Sampson: for his comments on this paper, for his suggestion to try implementing the library on top of STM, and for his suggestion that alts themselves could provide implicit guards, thus allowing alts to be nested as they can be in occam. Thanks are also due to the anonymous reviewers for their helpful comments, and to Claus Reinke for his suggestions regarding a few aspects of the library.

References

[1] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
[2] Peter H. Welch and Fred R. M. Barnes. Communicating mobile processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, 2005.
[3] Peter Welch, Neil Brown, Bernhard Sputh, Kevin Chalmers, and James Moores. Integrating and Extending JCSP. In Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch, editors, *Communicating Process Architectures 2007*, pages 349–370, 2007.
[4] Jan F. Broenink, Andrè W. P. Bakkers, and Gerald H. Hilderink. Communicating Threads for Java. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 243–262, 1999.
[5] Neil C. C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In *Communicating Process Architectures 2007*, pages 183–205, 2007.
[6] B. Orlic. and J.F. Broenink. Redesign of the C++ Communicating Threads Library for Embedded Control Systems. In *5th Progress Symposium on Embedded Systems*, pages 141–156, 2004.
[7] James Moores. CCSP – A Portable CSP-Based Run-Time System Supporting C and occam. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 147–169, 1999.
[8] John M. Bjørndalen, Brian Vinter, and Otto Anshus. PyCSP – Communicating Sequential Processes for Python. In *Communicating Process Architectures 2007*, pages 229–248, 2007.
[9] Kevin Chalmers and Sarah Clayton. CSP for .NET Based on JCSP. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 59–76, 2006.
[10] Alex Lehmberg and Martin N. Olsen. An Introduction to CSP.NET. In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 13–30, 2006.
[11] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 71–84, New York, NY, USA, 1993. ACM.
[12] Neil C. C. Brown and Peter H. Welch. An Introduction to the Kent C++CSP Library. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 139–156, 2003.
[13] Gerald H. Hilderink. *Managing Complexity of Control Software through Concurrency*. PhD thesis, Laboratory of Control Engineering, University of Twente, 2005.
[14] Peter H. Welch. Graceful termination – graceful resetting. In Andrè W. P. Bakkers, editor, *OUG-10: Applying Transputer Based Parallel Machines*, pages 310–317, 1989.
[15] C.A.R. Hoare. Fine-grain concurrency. In *Communicating Process Architectures 2007*, pages 1–19, 2007.
[16] John H. Reppy. First-class synchronous operations. In *TPPP '94: Proceedings of the International Workshop on Theory and Practice of Parallel Programming*, pages 235–252. Springer-Verlag, 1995.
[17] Neil C. C. Brown. Rain: A New Concurrent Process-Oriented Programming Language. In *Communicating Process Architectures 2006*, pages 237–251, September 2006.
[18] Ian R. East. The Honeysuckle programming language: an overview. *IEE Proc.-Softw.*, 150(2):95–107, April 2003.

[19] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theor. Comput. Sci.*, 60(2):177–229, 1988.

[20] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05*, pages 48–60. ACM, 2005.

[21] P.H. Welch. A Fast Resolution of Choice between Multiway Synchronisations (Invited Talk). In Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors, *Communicating Process Architectures 2006*, pages 389–390, 2006.

[22] P.H. Welch, F.R.M. Barnes, and F.A.C. Polack. Communicating complex systems. In Michael G Hinchey, editor, *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS-2006)*, pages 107–117, Stanford, California, August 2006. IEEE. ISBN: 0-7695-2530-X.

[23] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1993.

[24] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, 1973.

[25] F. Huch. Erlang-style distributed Haskell. In *11th International Workshop on Implementation of Functional Languages*, September 1999.

[26] Francisco Heron de Carvalho Junior and Rafael Dueire Lins. Haskell#: Parallel programming made simple and efficient. *Journal of Universal Computer Science*, 9(8):776–794, August 2003.

[27] Simon L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Symposium on Principles of Programming Languages*, pages 295–308. ACM Press, 1996.