CPSC 502: Interim Report
Implementation Status of CMPL

Student: Jared Pon. Supervisor: Dr. Robin Cockett

December 9, 2021

1 Introduction

Categorical Message Passing Language (abbr. CMPL), originally designed by [5], is a statically typed programming language which features constructs for sequential computation and concurrent computation. For sequential computation, CMPL exhibits a functional language with *function* definitions (which permit arbitrary recursion), and *data/codata* definitions. For concurrent computation, CMPL uses message passing as the concurrency primitive where communication between processes is given by the exchange of messages on *channels*. Here, we outline the structure of CMPL programs, and give some examples to show the expressiveness of CMPL when writing concurrent programs. In particular, we pay attention to examples given in [1] to see how such programs can be written in CMPL.

To facilitate the description of CMPL programs, we incrementally introduce the various constructs of CMPL starting with a short tour of the sequential constructs (which shares many ideas from traditional functional languages), and after we introduce various concurrent structures of CMPL alongside with examples. We note that we make no effort to fully describe the formal foundations of the but rather aim to give some intuition on writing CMPL programs. Formal details for operational semantics and proofs regarding CMPL programs can be found in [6, 2].

There is a compiler for CMPL implemented in Haskell which does execute the examples presented here, and this document can be seen as "tutorial-like" introduction for writing programs in CMPL. We finish this document discussing future directions for the current implementation.

2 Language Constructs

CMPL is a "two-tiered" language with a sequential system augmented upon a concurrent system. We will give a brief tour of the sequential system, and incrementally develop the concurrent system after.

2.1 Sequential Constructs

We discuss the sequential constructs of CMPL. The sequential constructs of CMPL are data type declarations, codata type declarations, and function definitions. We used *italicized text* to describe the metavariables, and `print` text to denote code of CMPL. A detailed description can be found in [5].

Data We write sequential type constructors as D_i, type variables as A_i, value constructors as C_i, sequential type expressions as $T_{i,j,k}$, and state variables as Z_i; where all such identifiers must be uppercase letters. The syntax for *data* is given by

```
data
```
$D_1(A_1, \ldots, A_k)$ `->` Z_1 `=`
$C_{1,1}$ `::` $T_{1,1,1}, \ldots, T_{1,1,n_1}$ `->` Z_1
$\vdots \quad \vdots$
C_{1,m_1} `::` $T_{1,m_1,1}, \ldots, T_{1,m_1,n_{m_1}}$ `->` Z_1
```
and
```
$\vdots \quad \vdots$
```
and
```
$D_\ell(A_1, \ldots, A_k)$ `->` Z_ℓ `=`
$C_{\ell,1}$ `::` $T_{\ell,1,1}, \ldots, T_{\ell,1,n_\ell}$ `->` Z_ℓ
$\vdots \quad \vdots$
C_{ℓ,m_ℓ} `::` $T_{\ell,m_\ell,1}, \ldots, T_{\ell,m_\ell,n_{m_\ell}}$ `->` Z_ℓ

Intuitively, we can understand the syntax of data types to read "the maps from the new type $D_i(A_1, \ldots, A_k)$ to the type Z_i are determined by the maps $C_{i,j}$ $(1 \leq j \leq m_i)$ of type $T_{j,k,1}, \ldots T_{j,k,n_j}$ to Z_i" for $1 \leq i \leq \ell$. Hence, the type of a type constructor $C_{i,j}$ is given by the operation

$$(T_{i,j,1}, \ldots, T_{i,j,n_i} \texttt{ -> } Z_i)[D_1(A_1, \ldots, A_k)/Z_1, \ldots, D_\ell(A_1, \ldots, A_k)/Z_\ell]$$

i.e., we substitute all state variables occurring in type expressions by replacing the ith state variable with the ith type constructor with the type variables. Note that the multiple state variables allow one to encode mutually recursive data types.

Data examples We can express a `Bool` data type as follows

```
data
    Bool() -> Z =
        True :: -> Z
        False :: -> Z
```

where we observe that the `Bool` type offers two choices: `True` or `False`.

We can also express a `List` data type as follows.

```
data
    List(A) -> Z =
        Cons :: A,Z -> Z
        Nil :: -> Z
```

where we observe that `List` is polymorphic with in the type variable `A`, and we use the state variable `Z` in the constructor `Cons` to express the recursive structure of a list. To construct a list from 1 to 3, one would write

```
Cons(1, Cons(2, Cons(3, Nil)))
```

Where we use built-in integer types. Alternatively, using the built in list syntax, one may write either of the following forms

```
[1,2,3]
```

Or

```
1:2:3:[]
```

As an example of a mutually recursive data type, we may express a rose tree as follows.

```
data
    Rose(A) -> R =
        Branch :: F -> R
    and
    Forest(A) -> F =
        FNil :: -> F
        FCons :: A,R -> F
```

where we note the use of the state variables `R` and `F` to encode the mutually recursive behavior of this type.

Codata Similarly to data, we write sequential type constructors as D_i, type variables as A_i, destructors as C_i, sequential type expressions as $T_{i,j,k}$, and state variables as Z_i; where all such identifiers must be uppercase letters. The syntax for *codata* declarations is as follows.

```
codata
    Z_1 -> D_1(A_1, ..., A_k) =
        C_{1,1} :: T_{1,1,1}, ..., T_{1,1,n_1}, Z_1 -> T_{1,1}
            ⋮       ⋮
        C_{1,m_1} :: T_{1,m_1,1}, ..., T_{1,m_1,n_{m_1}}, Z_1 -> T_{1,m_1}
    and
        ⋮   ⋮
    and
    Z_ℓ -> D_ℓ(A_1, ..., A_k) =
        C_{ℓ,1} :: T_{ℓ,1,1}, ..., T_{ℓ,1,n_ℓ}, Z_ℓ -> T_{ℓ,1}
```

$$\begin{array}{l} \vdots \quad\quad \vdots \\ C_{\ell,m_\ell} \texttt{ :: } T_{\ell,m_\ell,1},\ldots,T_{\ell,m_\ell,n_{m_\ell}},Z_\ell \texttt{ -> } T_{\ell,m_\ell} \end{array}$$

Intuitively, we can understand the syntax of codata types to read "the maps from the type Z_i to the new type $D_i(A_1,\ldots,A_k)$ are determined by the maps $C_{i,j}$ $(1 \leq j \leq m_i)$ of type $T_{j,k,1},\ldots T_{j,k,n_j},Z_i$ to $T_{j,k}$" for $1 \leq i \leq \ell$. Hence, the type of a type destruvtor $C_{i,j}$ is given by the operation

$$(T_{i,j,1},\ldots,T_{i,j,n_i},Z_i \texttt{ -> } T_{i,j})[D_1(A_1,\ldots,A_k)/Z_1,\ldots,D_\ell(A_1,\ldots,A_k)/Z_\ell]$$

i.e., we substitute all state variables occurring in type expressions by replacing the ith state variable with the ith type constructor with the type variables. Again, the multiple state variables allow one to encode mutually recursive codata types. Codata allows one to represent potentially infinite structures that are evaluated lazily.

Codata examples We can express a tuple with codata as follows.

```
codata
    Z -> Tuple(A,B) =
        Proj0 :: Z -> A
        Proj1 :: Z -> B
```

We can construct *records* of this codata type using the destructors as follows.

```
(Proj0 := -> 'a', Proj1 := -> 4)
```

Where we use built in character and int types to create a record of type `Tuple(Char, Int)`.

We can also express higher order functions with codata. Consider the following definition.

```
codata
    Z -> Fun(A,B) =
        Fun :: A,Z -> B
```

We can construct a record of this type as follows.

```
(Fun := a -> a + 1)
```

Where this record stores the function which increments the input by 1.

Functions A function in CMPL is given by the following syntax.

$$\begin{array}{l} \texttt{fun } functionName \texttt{ :: } T_1,\ldots,T_k \texttt{ -> } T \texttt{ =} \\ \quad pat_{1,1},\ldots,pat_{1,k_1} \texttt{ -> } body_1 \\ \quad\quad \vdots \quad\quad \vdots \quad\quad \vdots \\ \quad pat_{p,1},\ldots,pat_{p,k_p} \texttt{ -> } body_p \end{array}$$

where we note that we give the function $functionName$ an explicit type signature to indicate that this function is a map from $T_1,\ldots,T_k$ to T. We may also omit the explicit type annotation and write the following.

```
fun functionName =
    pat_{1,1}, ..., pat_{1,k_1} -> body_1
        ⋮     ⋮     ⋮
    pat_{p,1}, ..., pat_{p,k_p} -> body_p
```

Note that *functionName* must start with a lower case letter. Each pat_i is a *pattern* which allows destructuring user defined data types, user defined codata types, or primitive types. In particular, a pattern may be either of the following forms.

- A variable such as a which must be lower case.
- A data constructor such as $C_i(pat_1, \ldots, pat_k)$.
- A record pattern match such as $(C_1 := pat_1, \ldots, C_k := pat_k)$.
- A primitive type pattern much such as `4` or `'a'`.
- A pattern match which uses built in list syntax or tuple syntax such as [] or (a, b).

The $body_i$ of a function may be an expression, written as *expr* or $expr_i$, which is either of the following forms.

- A case expression to destructure a data type of the following form.

  ```
  case expr of
      C_1(pat_{1,1}, ..., pat_{1,n_1}) -> expr_1
          ⋮     ⋮
      C_p(pat_{p,1}, ..., pat_{p,n_p}) -> expr_p
  ```

- A constructor to build a data type of the following form.

  $$C_1(expr_1, \ldots, expr_k)$$

- A destructor applied to a record which destructors a codata type of the following form. (note this is the same syntax for data constructor application)

  $$C_1(expr_1, \ldots, expr_k)$$

- A record to build a codata type of the following form.

  ```
  ( C_1 := pat_{1,1}, ..., pat_{1,n_1} -> expr_1
  , ...
  , C_p := pat_{p,1}, ..., pat_{p,n_p} -> expr_p)
  ```

- An `if` or a `switch` statement for working with built-in bool types of the following form.

`if` $expr_0$ `then` $expr_1$ `else` $expr_2$

and

`switch`
$expr_1$ `->` $expr'_1$
$\vdots$
$expr_p$ `->` $expr'_p$

- A function call of the following form.

  $functionName$`(`$expr_1, \ldots, expr_p$`)`

- A let binding of the following form.

  `let`
  $funDefn_1$
  $\vdots$
  $funDefn_n$
  `in` $expr$

  where $funDefn_1, \ldots, funDefn_n$ are arbitrary function definitions.

- A literal such as a string literal, character literal, or int literal which may take the following form. `"some string"`, `'a'`, or `4`.

- A built in binary arithmetic operations such as $expr_1$ `+` $expr_2$, $expr_1$ `-` $expr_2$, $expr_1$ `*` $expr_2$, or $expr_1$ `/` $expr_2$.

- A built in list constructor or tuple constructor which may take the form `1:2:[]` or `(1,2)` as discussed previously.

Function example We can write the list append function as follows.

```
fun myAppend :: [A], [A] -> [A] =
    [],ts -> ts
    s:ss,ts -> s : myAppend(ss,ts)
```

where we note that this uses explicit recursion to append the two given input lists.

Summary We have given a short tour of the sequential constructs of CMPL. Most of the sequential constructs are similar to what one would expect in a functional language, so we hope this gives the reader a feel for the syntax of the sequential part of CMPL.

2.2 Concurrent Constructs

We develop the concurrent parts of CMPL. The concurrent communication of CMPL is centered around two *processes* sending or receiving sequential messages along exactly one *channel* for which these interactions are governed by the channel's type.

Processes We write process names as $processName$ which must start with a lowercase letter, concurrent types as P_i, Q_j, sequential types as S_i, patterns as pat_i, and channels as α_i, β_j. A *process* is given by the following syntax.

$$\begin{aligned}&\texttt{proc}\ procName\ \texttt{::}\ S_1,\ldots,S_\ell\ \texttt{|}\ P_1,\ldots,P_n\ \texttt{=>}\ Q_1,\ldots,Q_m\ \texttt{=}\\ &\quad pat_{1,1},\ldots,pat_{1,\ell}\ \texttt{|}\ \alpha_1,\ldots,\alpha_n\ \texttt{=>}\ \beta_1,\ldots,\beta_m\ \texttt{->}\ procBody_1\\ &\qquad\vdots\qquad\vdots\\ &\quad pat_{p,1},\ldots,pat_{p,\ell}\ \texttt{|}\ \alpha_1,\ldots,\alpha_n\ \texttt{=>}\ \beta_1,\ldots,\beta_m\ \texttt{->}\ procBody_p\end{aligned}$$

or, omitting the type signature, we may write

$$\begin{aligned}&\texttt{proc}\ procName\ \texttt{=}\\ &\quad pat_{1,1},\ldots,pat_{1,\ell}\ \texttt{|}\ \alpha_1,\ldots,\alpha_n\ \texttt{=>}\ \beta_1,\ldots,\beta_m\ \texttt{->}\ procBody_1\\ &\qquad\vdots\qquad\vdots\\ &\quad pat_{p,1},\ldots,pat_{p,\ell}\ \texttt{|}\ \alpha_1,\ldots,\alpha_n\ \texttt{=>}\ \beta_1,\ldots,\beta_m\ \texttt{->}\ procBody_p\end{aligned}$$

where we discuss the forms $procBody_i$ may take later. In this document, we will only be working with processes of the form

$$\begin{aligned}&\texttt{proc}\ procName\ \texttt{::}\ S_1,\ldots,S_\ell\ \texttt{|}\ P_1,\ldots,P_n\ \texttt{=>}\ Q_1,\ldots,Q_m\ \texttt{=}\\ &\quad pat_1,\ldots,pat_\ell\ \texttt{|}\ \alpha_1,\ldots,\alpha_n\ \texttt{=>}\ \beta_1,\ldots,\beta_m\ \texttt{->}\ procBody\end{aligned}$$

since internally all processes are desugared to a process of this form. We call the channels appearing to the left of `=>`, namely $\alpha_1,\ldots,\alpha_n$, *input polarity channels*; and we call the channels appearing to the right of `=>`, namely $\beta_1,\ldots,\beta_m$, *output polarity channels*. We say that α_i and β_j have *opposing polarity* or *opposite polarity* if α_i has input polarity and β_j has output polarity (vice versa). We also say that the channels $\alpha_1,\ldots,\alpha_n$ and $\beta_1,\ldots,\beta_m$ are *in scope* of the *procBody*. Operationally, this means that *procBody* may access channels α_i and β_j. We shall see later that the scope of channels is important for completely describing process interactions along channels. Note that pat_i has type sequential type S_i, input channel α_i has concurrent type P_i, and output channel β_i has concurrent type Q_i.

Process bodies We discuss the forms a process body, *procBody*, may take. A process body is a `do` block of process commands, written as *procCmd*, ending with either: a process command, an `if` statement, or a `case` statement. A process body can also be a single process command as well. Explicitly, a process body has the form:

```
do
```

$procCmd_1$

$\vdots$

$procCmd_n$

Or

`do`

$procCmd$

$\vdots$

`if` $expr$

`then` $procBody_1$

`else` $procBody_2$

Or

`do`

$procCmd$

$\vdots$

`case` $expr$ `of`

$C_1(pat_{1,1}, \ldots, pat_{1,n_1})$ `->` $procBody_1$

$\vdots \quad \vdots$

$C_p(pat_{p,1}, \ldots, pat_{p,n_p})$ `->` $procBody_p$

Or

$procCmd$

In a process command block, each process command is executed in order. If the process command block reaches an `if` or `case`, then the expression is evaluated and the corresponding process command block is selected and execution again proceeds executing each process command in order. Note that in the first form of a process command block, we say that $procCmd_n$ is the *last command* in a process command block, and similarly, for the third form of a process command block, we say that $procCmd$ is the *last command* as well (since it is the only command in the process command block).

In the following paragraphs, we discuss each form of a process command, and the operational meaning of each process command.

Plug process command The plug process command allows one to connect processes together along exactly one channel (of opposing polarity), and hence permit the processes to interact with each other along the given channel. The syntax is given as follows.

`plug`

$procName_1$`(`$arg_{1,1}, \ldots, arg_{1,\ell_1}$ `|` $\alpha_{1,1}, \ldots, \delta, \ldots, \alpha_{1,n_1}$ `=>` $\beta_{1,1}, \ldots, \beta_{1,m_1}$`)`

$procName_2$`(`$arg_{2,1}, \ldots, arg_{2,\ell_2}$ `|` $\alpha_{2,1}, \ldots, \alpha_{2,n_2}$ `=>` $\beta_{2,1}, \ldots, \delta, \ldots, \beta_{2,m_2}$`)`

Where we assume we have defined processes $procName_1$ and $procName_2$. We say that the channels $\alpha_{1,1}, \ldots, \alpha_{1,n_1}, \beta_{1,1}, \ldots, \beta_{1,m_1}, \alpha_{2,1}, \ldots, \alpha_{2,n_2}, \beta_{2,1}, \ldots, \beta_{2,m_2}, \delta$ are *used* in the plug process command. We read this process command as "process $procName_1$ and process $procName_2$ run in parallel and may interact only along channel δ." Note that δ is used as an input polarity channel for process $procName_1$, but δ is used as an output polarity channel for process $procName_2$.

It must be the case that we are calling $procName_1, procName_2$ with sequential arguments and concurrent channels of the expected type, every used channel in the plug process command must be distinct excluding δ, and δ *must be the same type for both* $procName_1, procName_2$. Moreover, the plug command must be the last command in a process command block, all channels in scope must occur as a used channel in the plug process command, and all used channels must be in scope (excluding δ). In general, we will make the implicit assumption that a channel may only be used if it is in scope (excluding the specific case of δ here).

Alternatively, instead of explicitly calling processes $procName_1$ and $procName_2$, one may explicitly write the process commands within the `plug` command alongside the channels which will be put in scope in the corresponding process body as follows.

`plug`

$\alpha_{1,1}, \ldots, \delta, \ldots, \alpha_{1,n_1}$ `=>` $\beta_{1,1}, \ldots, \beta_{1,m_1}$ `->` $procBody_1$

$\alpha_{2,1}, \ldots, \alpha_{2,n_2}$ `=>` $\beta_{2,1}, \ldots, \delta, \ldots, \beta_{2,m_2}$ `->` $procBody_2$

Here, we again say the channels $\alpha_{1,1}, \ldots, \alpha_{1,n_1}, \beta_{1,1}, \ldots, \beta_{1,m_1}, \alpha_{2,1}, \ldots, \alpha_{2,n_2}, \beta_{2,1}, \ldots, \beta_{2,m_2}$, and δ are *used* in the plug process command and are subject to the same restrictions mentioned previously. In $procBody_1$, the channels in scope are set to $\alpha_{1,1}, \ldots, \alpha_{1,n_1}, \beta_{1,1}, \ldots, \beta_{1,m_1}, \delta$, and similarly in $procBody_2$ the channels in scope are set to $\alpha_{2,1}, \ldots, \alpha_{2,n_2}, \beta_{2,1}, \ldots, \beta_{2,m_2}, \delta$ where we recall that δ has opposite polarity in $procBody_1$ and $procBody_2$.

On its own, the `plug` provides a method for two processes to communicate along a channel, but does not describe how these processes may actually communicate with each other along the shared channel. We develop how these processes may communicate with each other in the following paragraphs.

Generalized plug syntax We briefly discuss the generalized syntax for `plug`. Readers may skip this paragraph as it is syntactic sugar upon the previous plug forms mentioned. In [2], it is observed that the plug command (formally, a cut) is associative i.e., to lighten notation, write the plug command of processes p_1, p_2 along channel δ as $p_1 ;_\delta p_2$, then it is asserted that $p_1 ;_\delta (p_2 ;_{\delta'} p_3)$ can be identified with $(p_1 ;_\delta p_2) ;_{\delta'} p_3$. Intuitively, this amounts to asserting that if we run a process p_1 in parallel with a process which runs p_2 and p_3 in parallel, this is the same as running a process which runs a process p_1 and p_2 in parallel with process p_3 in parallel. So, in understanding this equivalence, this amounts to admitting a generalized plug form which may be written as follows.

`plug`

$procName_1$ (`(`$arg_{1,1}, \ldots, arg_{1,\ell_1}$ `|` $\alpha_{1,1}, \ldots, \delta_{12}, \ldots, \alpha_{1,n_1}$ `=>` $\beta_{1,1}, \ldots, \beta_{1,m_1}$ `)`
$procName_2$ (`(`$arg_{2,1}, \ldots, arg_{2,\ell_2}$ `|` $\alpha_{2,1}, \ldots, \alpha_{2,n_2}$ `=>` $\beta_{2,1}, \ldots, \delta_{12}, \ldots, \beta_{2,m_2}$ `)`
$\vdots$
$procName_n$ (`(`$arg_{n,1}, \ldots, arg_{n,\ell_n}$ `|` $\alpha_{n,1}, \ldots, \alpha_{n,n_2}$ `=>` $\beta_{n,1}, \ldots, \delta_{(n-1)n}, \ldots, \beta_{n,m_n}$ `)`

Or alternatively

`plug`
$arg_{1,1}, \ldots, arg_{1,\ell_1}$ `|` $\alpha_{1,1}, \ldots, \delta_{12}, \ldots, \alpha_{1,n_1}$ `=>` $\beta_{1,1}, \ldots, \beta_{1,m_1}$ `->` $procBody_1$
$arg_{2,1}, \ldots, arg_{2,\ell_2}$ `|` $\alpha_{2,1}, \ldots, \alpha_{2,n_2}$ `=>` $\beta_{2,1}, \ldots, \delta_{12}, \ldots, \beta_{2,m_2}$ `->` $procBody_2$
$\vdots$
$arg_{n,1}, \ldots, arg_{n,\ell_n}$ `|` $\alpha_{n,1}, \ldots, \alpha_{n,n_2}$ `=>` $\beta_{n,1}, \ldots, \delta_{(n-1)n}, \ldots, \beta_{n,m_n}$ `->` $procBody_n$

Where we note that arbitrary permutations of $procName_i$ (or $procBody_i$ with its corresponding used channels) are permitted. In other words, this runs all processes $procName_1, \ldots, procName_n$ in parallel which may pairwise communicate along channels $\delta_{(i-1)i}$ for $1 \leq i \leq n$. This admits similar rules for how channels are used as the previous plug forms, but with the following additional restrictions.

- Each $\delta_{(k-1)k}$ must be used exactly twice in in two distinct $procName_i, procName_j$.
- In the two uses of each $\delta_{(k-1)k}$, each use must have opposite polarity.
- Considering an undirected graph of nodes $procName_i$ with edges between $procName_i$ and $procName_j$ if $\delta_{(k-1)k}$ is used by $procName_i$ and $procName_j$, the graph must be completely connected and there must be no cycles.

These rules are put in place for technical reasons of the cut rule to help ensure progress of the system (deadlock/livelock freedom) – see [6] for details.

Close and halt process commands The `close` α command closes the channel α i.e., this expresses that the process wishes to terminate the interaction along channel α. The `halt` α command closes the channel α but also expresses that this process is finished all interactions. We say that the channel α is *used* by the process command `close` and `halt` respectively (recall we assume that all used channels must satisfy the restriction that they are in scope). Note that `halt` α must be the last command in a process command block and α must also be the only channel in scope – this restriction forces all interactions to be completed before the process command block may terminate. Moreover, note that `close` α removes α from being in scope as well. Both `close` α and `halt` α give the channel α the concurrent type `TopBot` which is a built in concurrent type expressing the end of an interaction.

As a simple first process one may write the following process. Note that `--` is a line comment.

```
proc noInteraction :: | TopBot => TopBot =  -- (1)
    | chA => chB -> do                      -- (2)
        close chA                           -- (3)
        halt chB                            -- (4)
```

We describe this process in more detail.

- `(1)` gives the type signature of the process `noInteraction` which has no sequential arguments, an input polarity channel of type `TopBot` and an output polarity channel of type `TopBot`.
- `(2)` puts channels `chA` and `chB` in scope where we note that both channels have concurrent type `TopBot` as given at `(1)`. Since both `chA` and `chB` are of type `TopBot`, recall this indicates that `chA` and `chB` both want to stop all interactions.
- `(3)` closes `chA` and hence removes `chA` from scope. Hence, this means that `chA` is no longer in scope at `(4)`.
- `(4)` halts `chB` which halts `chB` and moreover must be the last command in this process command block (terminating the process command block) with no other channels in scope (which ensures all channels finish their interactions).

This concludes the discussion of the `close` and `halt` process commands.

Get and put process commands The `get` *pat* `on` α command forces the current process to wait for a value to be received, and when the value is received, it binds the value to *pat* and proceeds down the process command block. Note that the `get` process command cannot be the last command in a process command block. We say that the channel α is *used* in the `get` process command, and hence must be in scope.

We discuss the typing of the channel α with the `get` *pat* `on` α process command. First, write the sequential type of *pat* as S. Now, if α has output polarity, then the process command `get` *pat* `on` α forces the type of the channel α to be `Get(S|P)` which reads as "the output polarity channel α will wait for a sequential value of type S and proceed behaving as concurrent type P." Note that "`|`" is a separator for the sequential type S and the concurrent type P. This syntax "`|`" is frequently used for this purpose in CMPL. But dually, if α has input polarity, the process command `get` *pat* `on` α forces the type of the channel α to be `Put(S|P)` which reads as "the input polarity channel α will wait for a sequential value of type S and proceed behaving as concurrent type P."

The `put` *expr* `on` α command forces the current process to evaluate *expr*, then send *expr* along the channel α. Similarly to the `get` process command, we say that the channel α is *used* by the `put` process command and hence must be in scope. Note that the `put` process command cannot be the last command in a process command block. Letting S denote the sequential type of *expr*, if α

has output polarity, then the process command `put` *expr* `on` α forces the type of the channel α to be `Put(S|P)` which reads as "the output polarity channel α will send a sequential value of type S and proceed behaving as concurrent type P." Dually, if α has input polarity, then the process command `put` *expr* `on` α forces the type of the channel α to be `Get(S|P)` which reads as "the input polarity channel α will send a sequential value of type S and proceed behaving as concurrent type P."

While it may appear strange that the type of the channel depends both on the process command used and its polarity, this is necessary to ensure reciprocating interactions along a plugged channel. Spelling this out more explicitly, if α is an input polarity channel, then `get` *pat* `on` α forces the channel α to have type `Put(A|P)`; but if β is an output polarity channel, then `put` *pat* `on` β forces the channel β to have type `Put(A|P)` i.e., we see α and β are of opposite polarity, but are of the same type and can hence be plugged together by the `plug` command for which these channels have reciprocating actions of getting and putting messages.

With the `get` and `put` constructs, we can write process which simply forwards a sequential message to a different channel.

```
proc forwardMessage ::
    | Get(A|TopBot) => Get(A|TopBot) = -- (1)
        | chA => chB -> do
            get val on chB              -- (2)
            put val on chA              -- (3)
            close chB                   -- (4)
            halt chA                    -- (5)
```

We explain each line.

- `(1)` specifies the type of the process `forwardMessage`. It tells us that we have an *input polarity channel* of type `Get(A|TopBot)` and hence is understood as "putting a sequential value of type `A`, and proceeding as `TopBot` to terminate all interactions along this channel," and moreover, it tells us that we have an *output polarity channel* of type `Get(A|TopBot)` which is understood as "getting a sequential value of type `A`, and proceeding as `TopBot` to terminate all interactions along this channel."
- `(2)` and `(3)` are the process commands which do the getting then the putting of `val` to forward the message from `chA` to `chB`.
- `(4)` and `(5)` close the channels `chA` and `chB`.

Indeed, we could have switched `(3)` and `(4)` which would instead close `chB` before forwarding `val` to the channel `chA`.

Now, we attempt to model a less trivial scenario with these process commands. We want to model a classic scenario of a client who wishes to buy dog treats on a remote server. This example, inspired by [1], will be a guiding example for which we develop into a more sophisticated server as we progress introducing more built in CMPL constructs.

The scenario from the client's perspective is described as: a client attempts to buy a dog treat by sending the server a product name and a credit card number, after which the client receives back a receipt. From the client's perspective, assuming this interaction occurs on an output polarity channel, the type which models this interaction is

```
Put([Char]|Put(Int|Get([Char]|TopBot)))
```

Where we assume that the product name is represented by a list of characters `[Char]`, the credit card number is represented by an int `int`, and the receipt is also represented with a list of characters `[Char]`. Hence, the process, `client`, which implements this kind of interaction can be given as follows.

```
proc client ::
  [Char],Int | => Put([Char]|Put(Int|Get([Char]|TopBot))) = -- (1)
    product, card | => chA -> do
        put product on chA                  -- (2)
        put card on chA                     -- (3)
        get receipt on chA                  -- (4)
        halt chA                            -- (5)
```

We explain how this works.

- `(1)` gives the type of the process `client`. Indeed, we observe that `client` takes two sequential types for the product name (as the type `[Char]`) and the credit card number (as the type `Int`), and offers an output polarity channel of type `Put([Char]|Put(Int|Get([Char]|TopBot)))` which says "put a list of characters on this channel, then proceed to put an int on this channel, then proceed to wait and get for a list of characters on this channel, then terminate all communications with this channel. "
- `(2)`, `(3)`, `(3)`, `(4)`, `(5)` are straightforward unwrapping of the type of the channel `chA`.

The server implementing the corresponding interaction (along an input polarity channel) could be implemented as follows.

```
proc server ::
    [Char],Int | Put([Char]|Put(Int|Get([Char]|TopBot))) => =
        | chA  => -> do
            get product on chA
            get card on chA
            put genReceipt(product, card) on chA
            halt chA
```

Where we assume we have some sequential function `genReceipt` of type `[Char],Int -> [Char]`. Note the apparent "duality" between the client and the server's process commands: the client puts (sends) a product but the server gets (receives) a product, the client puts (sends) a credit card number but the server gets (receives) a

credit card number, and the client gets (receives) a receipt but the server puts (sends) a receipt.

A practical implementation would probably want to execute some sort of side-effect to actually *give* the receipt to the client (instead of just holding it as a local variable), or the server would actually like to *modify* their stock during the interaction. The handling of side effects will be briefly discussed later when we discuss *service channels*, but for now we may observe the expressivity of the type system to represent the specification.

Now, recalling the `plug` command, we can see that we can make a process named `runner` which plugs the client and server together. Note that we may actually plug the process `client` and `server` together along channel `ch` since `ch` is used as an input polarity channel in `client`, is used as an output polarity channel in `server`, and `ch` has the same type in both `client` and `server`. Hence, we have

```
proc runner :: | => =
    | => -> plug
        client( | => ch)
        server( | ch => )
```

We will later develop this example to permit window shopping of a client browsing dog treats and their prices. Currently, we cannot express such scenario since the type forces the client to send the product, then their credit card number, then they get a receipt without really knowing anything about the product they were buying. We will fix these issues later by using *protocols* and *coprotocols*.

Protocols and coprotocols Protocols and coprotocols are concurrent types which allow one to encode choice and arbitrary recursion for an interaction along a channel. In other words, a protocol/coprotocol provides a concurrent type for a channel which determines the sequencing of messages which are allowed to be sent or received on the channel. We write S_i for sequential types, $P_i, Q_{j,k}$ for concurrent types, Z_i for state variables, and C_i for protocol/coprotocol handles (abbr. handles). The syntax for protocols is given as follows.

```
protocol
    D_1(S_1, ..., S_{k_s} | P_1, ..., P_{k_p}) => Z_1 =
        C_{1,1} :: Q_{1,1} => Z_1
            ⋮       ⋮
        C_{1,m_1} :: Q_{1,m_1} => Z_1
    and
        ⋮       ⋮
    and
    D_ℓ(S_1, ..., S_{k_s} | P_1, ..., P_{k_p}) => Z_ℓ =
        C_{ℓ,1} :: Q_{ℓ,1} => Z_ℓ
            ⋮       ⋮
```

C_{ℓ,m_ℓ} `::` Q_{ℓ,m_ℓ} `=>` Z_ℓ

The syntax for coprotocols is as follows.

```
coprotocol
```
Z_1 `=>` $D_1(S_1,\ldots,S_{k_s}|P_1,\ldots,P_{k_p})$ `=`
C_1 `::` Z_1 `=>` $Q_{1,1}$
$\vdots$ $\vdots$
C_{m_1} `::` Z_1 `=>` Q_{1,m_1}
`and`
$\vdots$ $\vdots$
`and`
Z_ℓ `=>` $D_\ell(S_1,\ldots,S_{k_s}|P_1,\ldots,P_{k_p})$ `=`
C_1 `::` Z_ℓ `=>` $Q_{\ell,1}$
$\vdots$ $\vdots$
C_{m_ℓ} `::` Z_ℓ `=>` Q_{ℓ,m_ℓ}

It will be clear in a moment how these data types manipulate the channel type when we give the `hput` and `hcase` constructs and how the type effects the corresponding operation of the process.

hcase and hput process commands We keep the protocol and coprotocol definitions' notation from the previous paragraphs. Intuitively, we can think the `hput` process command constructs a protocol/coprotocol handle on a channel; and correspondingly, the `hcase` command allows one to choose how to respond to each possible protocol/coprotocol handle. This situation is the concurrent analogue to data constructors and the `case` expression. Note that the difference between a protocol and a coprotocol is significantly less dramatic than it is in the concurrent world. We describe how this mechanism works in more detail.

If we are given a protocol handle C_j `::` $Q_{i,j}$ `=>` Z_i, and an *output polarity channel* α, then the process command `hput` C_j `on` α sends the process handle C_j along the channel α. Note that an `hput` process command cannot be the last command in a process command block. We say that the channel α is *used* by the process command `hput` and hence α must be in scope. Moreover, `hput` C_j `on` α forces α to be of type $Z_i[D_i(S_1,\ldots,S_{k_s}|P_1,\ldots,P_{k_p})/Z_i]$, but the channel α then proceeds as type $Q_{i,j}[D_1(S_1,\ldots,S_{k_s}|P_1,\ldots,P_{k_p})/Z_1,\ldots,D_\ell(S_1,\ldots,S_{k_s})/Z_\ell]$. Dually, if we are given an *input polarity channel* β, the process command of `hput` is given as follows.

`hcase` β `of`
$C_{i,1}$ `->` $procBody_1$
$\vdots$
C_{i,m_i} `->` $procBody_{m_i}$

Where we understand the operational semantics to make this process to wait for an appropriate protocol handle $C_{i,j}$ to be received, then the process will

proceed as $procBody_j$. Note that `hcase` must be the last command in a process command block, and we say that the channel β is *used* by the process command `hcase` and hence must be in scope. Moreover, this forces β to have type $Z_i[D_i(S_1, \ldots, S_{k_s}|P_1, \ldots, P_{k_p})/Z_i]$, and β for every j, β in $procBody_j$ has type $Q_{i,j}[D_1(S_1, \ldots, S_{k_s}|P_1, \ldots, P_{k_p})/Z_1, \ldots, D_\ell(S_1, \ldots, S_{k_s})/Z_\ell]$.

The case when we are given a coprotocol handle C_j `::` $Q_{i,j}$ `=>` Z_i is essentially identical to the case of a protocol handle except for swapping whether we have an input polarity channel or an output polarity channel. Spelling this out more explicitly, if we are given a coprotocol handle C_j `::` $Q_{i,j}$ `=>` Z_i, and an *input polarity channel* α, then the process command `hput` C_j `on` α sends the process handle C_j along α. Also, if we are given an *output polarity channel* β, the process command

`hcase` β `of`
$C_{i,1}$ `->` $procBody_1$
$\vdots$
C_{i,m_i} `->` $procBody_{m_i}$

makes the process to wait for an appropriate protocol handle $C_{i,j}$ to be received, in which case, the process will proceed as $procBody_j$. The typing of the channels α, β is identical to the case of protocols as well.

Going back to our client/server example, recall we would like to model a dog treat window shopper who would like to either: get a quote for the price of a dog treat, or simply purchase a product as before. We can consider the following protocol.

```
protocol
    Transaction( | ) => Z =
        Buy :: Put([Char]|Put(Int|Get([Char]|TopBot))) => Z
        Quote :: Put([Char]|Get(Int|TopBot)) => Z
```

Where we note that the protocol handle `Buy` matches the type exactly as before when the client purchased a dog treat, whereas the protocol handle `Quote` with the type `Put([Char]|Get(Int|TopBot))` models sending a product then receiving the price of the product and then ending the interaction. Then, a client who wishes to get a quote can be written as follows.

```
proc clientQuote :: [Char] | => Transaction =
    product | => chA -> do
        hput Quote on chA
        put product on chA
        get price on chA
        halt chA
```

Or alternatively, a client who wishes to actually purchase a dog treat could be written as

```
proc clientPurchase :: [Char],Int | => Transaction =
    product,card | => chA -> do
```

```
        hput Buy on chA
        put product on chA
        put card on chA
        get receipt on chA
        halt chA
```

It is important to note that in both instances, the type of `chA` starts as the type `Transaction` given by its protocol definition, but the choice of the constructor `Quote` or `Buy` completely determines `chA`'s proceeding behaviour.

The corresponding server such clients can be written as follows.

```
proc server :: | Transaction => =
    |  chA => -> do hcase chA of
        Buy -> do
            get product on chA
            get card on chA
            put genReceipt(product, card) on chA
            halt chA
        Quote -> do
            get product on chA
            put productPricing(product) on chA
            halt chA
```

Where we again assume we have some sequential function `genReceipt` of type `[Char],Int -> [Char]`, and a function `productPricing` of type `[Char] -> Int`. Note how the server will choose how to interact with the client based on the protocol handle given, so this server provides reciprocating actions to both the `clientQuote` process and the `clientPurchase` process.

Our next command will allow us to improve upon this example by allowing repeated interaction. In particular, a client may want to get a quote on a few dog treats, then (and only then) proceed to actually buy a dog treat.

Process call command Given a process named *processName*, we may call this process with the process command

processName`(`$expr_1, \ldots, expr_\ell$ `|` $\alpha_1, \ldots, \alpha_n$ `=>` $\beta_1, \ldots, \beta_m$`)`

where $expr_1, \ldots, expr_\ell$ are expressions, $\alpha_1, \ldots, \alpha_n$ are input polarity channels, and $\beta_1, \ldots, \beta_m$ are output polarity channels. Note that each argument (expressions and all channels) must be an instance of the type *processName*, and a process call must be the last process command in a process command block. We say the channels $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m$ are *used* by the process call command and hence must all be in scope. Moreover, all channels currently in scope must be used in the process call command. Intuitively, process call commands give us a means for recursion for processes.

With a process call command, we can extend our client/server example. In particular, we extend it by allowing the server to handle clients who repeatedly continue buy or quote different products, or close the connection when they wish.

We start off showing how we may make a client who will get a quote then buy a dog treat (provided it is within budget) then leave. We first must modify the `Transaction` protocol so the type permits us to recurse as follows.

```
protocol
    Transaction( | ) => Z =
        Buy :: Put([Char]|Put(Int|Get([Char]|Z))) => Z
        Quote :: Put([Char]|Get(Int|Z)) => Z
        Leave :: TopBot => Z
```

Where we note that we change the type the handles `Buy` and `Quote` to use the state variable `Z`, and we introduce the handle `Leave` which forces the channel to be closed. Then, we may write the client process `clientQBL` who will quote a product and then purchase it (provided it is less than 20 dollars) as follows.

```
proc clientQBL :: [Char], Int | => Transaction =
    product, card | => chA -> do
        hput Quote on chA                -- (1)
        put product on chA               -- (2)
        get price on chA                 -- (3)
        if price < 20                    -- (4)
            then do
                hput Buy on chA          -- (5)
                put product on chA       -- (6)
                put card on chA          -- (7)
                get recipt on chA        -- (8)
                hput Leave on chA        -- (9)
                halt chA                 -- (10)
            else do
                hput Leave on chA        -- (11)
                halt chA                 -- (12)
```

We explain in more detail what this example is doing.

- `(1)` – `(3)` is getting the quote of the desired product.
- `(4)` is using the built in comparison operator `<` on `Int`s to check if the price is less than 20 dollars.
- `(5)` – `(10)` is executed when `(4)` succeeds i.e., the dog treat is a reasonable price. Recalling that we changed the handle type of `Quote`, this justifies line `(5)`, and so we may proceed through lines `(5)` – `(8)` to purchase the product, and finish with lines `(9)` – `(10)` which will close the interaction.
- `(11)` – `(12)` is executed when `(4)` fails i.e., the dog treat is deemed far too expensive. Again, recalling that we changed the handle type of `Quote`, this justifies `(11)`, and we proceed to `(12)` which closes the interaction.

Now, we give the corresponding definition of the server.

```
proc server :: | Transaction => =
    | chA  => -> hcase chA of
        Buy -> do                                          -- (1)
            get product on chA                             -- (2)
            get card on chA                                -- (3)
            put genReceipt(product, card) on chA           -- (4)
            server( | chA => )                             -- (5)
        Quote -> do                                        -- (6)
            get product on chA                             -- (7)
            put productPricing(product) on chA             -- (8)
            server( | chA => )                             -- (9)
        Leave -> do                                        -- (10)
            halt chA                                       -- (11)
```

Where we again assume we have some sequential function `genReceipt` of type `[Char],Int -> [Char]`, and a function `productPricing` of type `[Char] -> Int`. We describe in more detail what this process does.

- `(1)` – `(4)` and `(6)` – `(8)` are the same from the previous `server`.
- `(5)` and `(9)` is the recursive call on the `server` process which allows the server to respond to another `Buy`, `Quote`, or `Leave` handle.
- `(10)` – `(11)` close the channel in the case when the `Leave` handle is received i.e., the corresponding client is done with the interaction.

Note that this `server` process may handle arbitrary client interactions not limited to just `clientQBL` e.g. this process may respond to clients who quote, then quote, then leave; or buy, then buy, then leave; etc. Hence, we have shown how clients with extended interactions may interact with a server.

We are missing one crucial aspect in this example: we cannot model having multiple clients connected to a server. In the following paragraphs, we will introduce more constructs to permit such interactions.

Split and fork The `split` command allows a channel to be discarded and replaced by two new distinct channels. Precisely, if α is an *input polarity channel*, then the process command `split` α `into` α_1, α_2 removes the channel α from being in scope, and puts two new *input polarity channels* α_1, α_2 in scope for further interaction. We say that α is *used* by the process command split and hence must be in scope. If α_1 is of type P and α_2 is of type Q, this forces the type of *input polarity channel* α to be P `(*)` Q (pronounced "P tensor Q"). Note that the `split` process command must not be the last process command in a process command block.

Similarly, if α is an *output polarity channel*, then the process command `split` α `into` α_1, α_2 removes the channel α from being in scope, and puts two new *output polarity channels* α_1, α_2 in scope for further interaction. If α_1 is of type P and α_2 is of type Q, this forces the type of the *output polarity channel* α to be P `(+)` Q (pronounced "P par Q").

Again, note the importance that polarity plays in the typing of the channels – the input polarity channel is given the type P `(*)` Q whereas the output polarity channel is given the type P `(+)` Q, but the operational semantics are the same.

The reciprocating process command to `split` is the `fork` command. Given an *input polarity channel* α, the `fork` is written as follows.

`fork` α `as`
    α_1 `->` $procBody_1$
    α_2 `->` $procBody_2$

We say that the channel α is *used* by the fork process command and hence must be in scope. Operationally, the fork command removes the channel α from being in scope; and puts an *input polarity channel* α_1 in scope exclusively for $procBody_1$ and similarly puts an *input polarity channel* α_2 in scope exclusively for $procBody_2$ where $procBody_1$ and $procBody_2$ are executed in parallel. Moreover, $procBody_1$ and $procBody_2$ must use all channels in scope and may not share any channels i.e., the set of channels used in $procBody_1$ and $procBody_2$ must be disjoint. If the type of the input polarity channel α_1 is P and the type of the input polarity channel α_2 is Q, then since α is an *input polarity channel* the type of α is then P `(+)` Q. Note that the `fork` process command must be the last process command in a process command block.

For the case when α is an *output polarity channel*, this is identical to the case when α is an input polarity channel but we replace all occurrences of "input polarity channel" with "output polarity channel" and the resulting type of α is P `(*)` Q instead. Explicitly, for an *output polarity channel* α, the process command

`fork` α `as`
    α_1 `->` $procBody_1$
    α_2 `->` $procBody_2$

discards the channel α; and creates a new *output polarity channel* α_1 available only in $procBody_1$ and similarly creates a new *output polarity channel* α_2 available only in $procBody_2$. Moreover, $procBody_1$ and $procBody_2$ may not share any channels i.e., the set of channels used in $procBody_1$ and $procBody_2$ must be disjoint. If the type of the output polarity channel α_1 is P and the type of the output polarity channel α_2 is Q, then since α is an *output polarity channel* the type of α is then P `(*)` Q.

With these commands in mind, we extend our example to permit multiple clients communicating on duplicated servers following the example given by [1]. We first show how this can be done with exactly two clients, then we show how this may be generalized to an arbitrary number of clients. We suppose we have two clients `clientQBL` as implemented previously with the corresponding `Transaction` protocol. Then, we want each process `clientQBL` to communicate with an instance of `server` as implemented previously. So, the only issue is how we would `plug` these processes together i.e., how can we arrange a process `runner` which plugs these multiple processes together. A naive thought would

be to just `plug` the processes together, but recalling that `plug` may only plug processes together with exactly one channel in common, this poses an issue for our case of wanting to plug multiple clients to duplicated servers. Hence, this motivates the need for plugging a channel together which uses the `split` and the `fork` process command as follows.

```
proc runner :: | => =
    | => -> plug                                          -- (1)
        => ch -> fork ch as                               -- (2)
            ch1 -> clientQBL("Bacon",0 | => ch1)          -- (3)
            ch2 -> clientQBL("Bone",1 | => ch2)           -- (4)
        ch => -> do                                       -- (5)
            split ch as ch1,ch2                           -- (6)
            plug                                          -- (7)
                ch1 => z -> do                            -- (8)
                    close z                               -- (9)
                    server( | ch1 => )                    -- (10)
                ch2, z => -> do                           -- (11)
                    close z                               -- (12)
                    server( | ch2 => )                    -- (13)
```

We explain in more detail what this process is doing and in particular point out where the reciprocating process commands are happening.

- `(1)`, `(2)`, and `(5)` execute the `plug` command to connect the two processes along the channel `ch`. Note we use the alternative syntax for the `plug` command where we may write the process body inline instead of explicitly calling a process. The channel `ch` has the following type.

  ```
  Transaction (*) Transaction
  ```

  Where we note that the type `Transaction` appears twice to emulate the two clients.

- `(2)` executes the `fork` command where we recall this removes `ch` from scope, and executes `(3)` and `(4)` in parallel. Note how the reciprocating `split` command occurs in line `(6)`.

- `(3)` executes the client who wishes to buy the `"Bacon"` dog treat product with credit card number `0`. Note that the reciprocating server call is at `(10)`. Similarly, we can see this effect with the client call at `(4)` with the reciprocating server at `(13)`.

- The `plug` at `(7)` opens a new channel `z` of type `TopBot` which immediately closes itself at `(9)` and `(12)`. It is necessary to have this since a `plug` command must connect two processes along *exactly* one channel[1]. Then, recalling that `plug` executes `(8)` – `(10)` and `(11)` – `(13)` in parallel, this

[1] In future revisions of the language, this restriction may be changed to *at most one* channel to make such operations less awkward

gives us two servers at `(10)` and `(13)` which independently run in parallel communicating with their corresponding clients which was the desired behaviour.

Having investigated the two client case, we can note that this essentially generalizes to a "list-like" protocol to handle an arbitrary number of clients. We give the definition of our "list-like" protocol which permits an arbitrary number of clients as follows.

```
protocol
    Clients( | T) => S =
        Cons :: T (*) S => S
        Nil :: TopBot => S
```

Where we note that we generalize the `Transaction` protocol type to any concurrent type `T`. Then, the corresponding `duplicateServer` process would be defined as follows.

```
proc duplicatedServer :: |  Clients( | Transaction) => =
    | clients =>  -> hcase clients of
        Cons -> do
            split clients into ch1,clients
            plug
                ch1 => z -> do
                    close z
                    server( | ch1 => )
                clients,z => -> do
                    close z
                    duplicatedServer( | clients => )
        Nil -> halt clients
```

Where we note that main difference from the two client case and this case is the recursive call of `duplicatedServer( | clients => )`. A corresponding process `someClients` to model multiple clients interacting with the server in distinct ways can then be developed as follows.

```
proc someClients :: |  Clients( | Transaction) => =
    | clients =>  -> do
        hput Cons on clients
        fork clients as
            -- Client who wants a quote for chicken
            ch1 -> clientQuote( "Chicken" | ch1 => )
            clients -> do
                hput Cons on clients
                fork clients as
                    -- Client who wants to purchase bacon
                    ch2 -> clientPurchase( "Bacon", 0 | ch2 => )
                    clients -> do
                        hput Cons on clients
```

```
fork clients as
    -- Client who wants to purchase beef
    ch2 -> clientPurchase( "Beef", 1 | ch2 => )
    clients -> do
        hput Nil on clients
        halt clients
```

Where this example models having 3 different clients trying to: get a quote for `"Chicken"`, purchase `"Bacon"`, and purchase `"Beef"`.

In summary, we have shown how the `fork` and `split` process command can be used to allow a `plug` command to essentially communicate along two distinct channels. Moreover, we have demonstrated how when these process commands are used in combination with a protocol, we may have a potentially unbounded number of channels interacting.

Id and Neg We introduce the process command for identification and negation of two channels. Intuitively, these process commands gives us a way to manipulate channels between processes.

Given an input polarity channel α and an output polarity channel β, we may write the *identification process command* as α `|=|` β. We understand this command to identify the channels α and β as the same channels i.e., all messages on α are now messages on β as well and vice versa. This must be the last command in a process command block (with no other channels in scope), and forces the channels α and β to have the same concurrent type. Moreover, we say that the channels α, β are used in the identification process command and hence must be in scope. Similarly, we may also write β `|=|` α which has identical meaning.

It is also of use to identify channels of the *same polarity*. Hence, if α and β are of the same polarity and β has concurrent type `T`, we may write the *negation process command* as α `|=| neg` β which must be the last command in a process command block (with no other channels in scope), and forces the channel α to have concurrent type `Neg(T)` where `Neg(_)` is a built in type. Also, we say that α, β are used by the negation process command and hence must be in scope. This has identical meaning to the *identification process command* where we identify α and β as the same channels.

We give the *memory cell* example which offers a way to provide mutual exclusion of some shared state between two processes. We first give the definition of the `memCell` process (the process which holds the shared state) and its `MemCell` protocol to define how other processes may interact with the shared state. Consider the following definitions.

```
protocol MemCell (A | ) => S =
    MemPut :: Put(A|S) => S      -- (1)
    MemGet :: Get(A|S) => S      -- (2)
    MemCls :: TopBot => S        -- (3)

proc memCell :: A | MemCell(A | ) => =
```

```
val | memch => -> hcase memch of
    MemPut -> do                        -- (4)
        get nval on memch               -- (5)
        memCell(nval | memch => )       -- (6)
    MemGet -> do                        -- (7)
        put val on memch                -- (8)
        memCell(val | memch => )        -- (9)
    MemCls -> halt memch                -- (10)
```

We understand the `memCell` process to hold some sequential state of type `A` in its argument `val`, and access to this state is provided by the channel `memch` for which the interactions of `memch` is guided by its type `MemCell(A | )`. The concurrent type `MemCell(A | )` provides three possible interactions given by the process handles defined above, and each of these interactions are implemented in the process `memCell`. We describe how this mechanism works as follows.

- `(1)` is the process handle which allows one to put a new value of type `A` in the memory cell overriding the original value in the memory cell. This is idea is implemented from `(4)` to `(6)` where we see that the `memCell` process will receive a new value `nval` on the channel `memch` and recurse with the new value at `(6)` to hold `nval` as the new state value.
- `(2)` is the process handle which allows one to get the current value of type `A` in the memory cell. This is idea is implemented from `(7)` to `(9)` where we see that the `memCell` process will put its current value `val` on the channel `memch` and recurse with the current value `val` at `(6)` to leave the current value unchanged.
- `(3)` is the process handle which allows one to close the memory cell which indicates we are no longer interested in storing values in this memory cell. This idea is implemented at `(10)` where we see that we close the channel `memch`.

One example of an interaction with the memory cell is as follows.

```
proc runner :: | => =
    | => -> plug
        memch => ->
            memCell( 0 | memch => )     -- (1)
        => memch -> do
            hput MemGet on memch        -- (2)
            get val on memch            -- (3)
            hput MemPut on memch        -- (4)
            put (1 + val) on memch      -- (5)
            hput MemCls on memch        -- (6)
            halt memch                  -- (7)
```

We describe this interaction.

- `(1)` starts the `memCell` with `0` as the initial state.

- `(2)` and `(3)` show how a process may get the current value from the memory cell.
- `(4)` and `(5)` show how a process may overwrite the current value of the memory cell. In particular, we see that we put the value `1 + val` in the memory cell.
- `(6)` and `(7)` ends the interaction with the memory cell.

So, we have established how one may store some form of state with the process `memCell`. All that remains is to show is how this shared state stored in the process `memCell` can be passed between two processes. Recalling that processes manipulate the shared state of the process `memCell` by interacting along the channel provided by `memCell`, the action of passing shared state between two processes is hence given by manipulating the channel provided by `memCell` to be passed between the two processes. To implement this idea, we need to make use of the identification process command and the negation process command such that the channel which provides access to the memory cell is passed between the two processes.

We first show how we can pass `memCell` from a process `p1` to another process `p2`. Then, we will show how this can be developed so we can pass `memCell` back from `p2` to `p1` arbitrarily. Consider the following definitions which passes `memCell` from process `p1` to `p2`.

```
proc p1 :: |  MemCell(Int|) => MemCell(Int|) =
    | passer => mem -> do
        hput MemGet on mem       -- (1)
        get inp on mem           -- (2)
        passer |=| mem           -- (3)

proc p2 :: |  => MemCell(Int|) =
    | => passer -> do
        hput MemGet on passer    -- (4)
        get inp on passer        -- (5)
        hput MemPut on passer    -- (6)
        halt passer              -- (7)

proc runner :: | => =
    | => -> plug
        memCell(0 | memch => )       -- (8)
        => memch -> plug             -- (9)
            p1(| passer => memch)    -- (10)
            p2(| => passer)          -- (11)
```

We describe the interaction in more detail as follows.

- From `(8)` to `(11)` the initial interaction begins. `(8)` starts the memory cell with an initial value `0` and provides access to the memory cell along

channel `memch`. From `(9)` to `(11)`, we run `p1` and `p2` in parallel, for which `p1` starts with access to the memory cell with channel `memch`, and later identifies `memch` with `passer` to move the memory cell so `p2` may access it at `(11)`.

- From `(1)` to `(2)` we see that `p1` may interact with the memory cell. At `(3)`, we identify the memory cell's channel `mem` with the channel `passer` which is plugged to `p2`. This channel identification is the manipulation of channels which allows `p1` to now have access to the memory cell.
- From `(4)` to `(7)`, we see that `p2` now has access to the memory cell and may interact with it.

This illustrates how one can pass the memory cell from `p1` to `p2`. We now improve upon this example to show how we can modify `p1` and `p2` such that the memory cell can be passed from `p1` to `p2`, then from `p2` back to `p1`. Consider the following definitions.

```
proc p1 :: | MemCell(Int|) (+) Neg(MemCell(Int|)) => MemCell(Int|) =
    | passer => mem -> do
        hput MemGet on mem                          -- (1)
        get inp on mem                              -- (2)
        fork passer as                              -- (3)
            nmem -> mem |=| nmem                    -- (4)
            negmem -> plug                          -- (5)
                => z -> do                          -- (6)
                    hput MemCls on z                -- (7)
                    halt z                          -- (8)
                z,negmem => negmem |=| neg z        -- (9)

proc p2 :: |  => MemCell(Int|) (+) Neg(MemCell(Int|)) =
    | => passer -> do
        split passer into nmem,negmem               -- (10)
        hput MemGet on nmem                         -- (11)
        get inp on nmem                             -- (12)
        negmem |=| neg nmem                         -- (13)
```

We highlight the key differences from the previous example.

- Similarly, to the previous example, `(1)` and `(2)` show that `p1` may initially interact with the memory cell.
- At `(3)` in `p1`, instead of identifying the channel `mem` with the channel `passer` to pass over the memory cell to `p2`, we instead fork `passer` to pass over the memory cell at `(4)`. Whereas from `(5)` to `(9)` we execute the machinery to pass the memory cell back from `p2`.
- At `(10)` in `p2`, the reciprocating split call is executed, for which we get two channels: `nmem` and `negmem` which are the channels to communicate

with the memory cell and pass the memory cell back to `p1` respectively. Indeed, from `(11)` to `(12)`, we see a usual interaction with the memory cell, but at `(13)`, we see the memory cell is then passed back to `p1` via the channel `negmem`. Note how the negation process command is necessary here to identify two channels of the same polarity.

- Then, going back to `(5)` to `(9)` from `p1`, we see that `negmem` and `z` are identified (and hence `z` provides access to the memory cell), which justifies `(7)` to `(8)` using the memory cell again along channel `z`.

So, we have shown how one may pass a memory cell between processes twice. Now, we would like to show how this generalizes to passing the memory cell between two processes an unlimited number of times (the case for a finite number of times is similar). This is essentially a straightforward generalization of the previous example where we use reciprocating fork and split commands to aid in threading the channel (by using the negation and identification commands) between the two processes. Consider the following definitions.

```
protocol
    Passer( | M) => S =
        Passer :: M (+) (Neg(M) (*) S) => S

proc p1 :: |  Passer(| MemCell(Int|)) => MemCell(Int|) =
    | passer => mem -> hcase passer of
        Passer -> do
            hput MemGet on mem
            get inp on mem
            fork passer as
                nmem -> nmem |=| mem
                negmemnpasser -> do
                    split negmemnpasser into negmem, npasser
                    plug
                        p1( | npasser => z)
                        z, negmem => -> negmem |=| neg z

proc p2 :: |  => Passer( | MemCell([Char]|)) =
    | => passer -> do
        hput Passer on passer
        split passer into mem, negmemnpasser
        hput MemGet on mem
        get inp on mem
        fork negmemnpasser as
            negmem -> negmem |=| neg mem
            npasser -> p2( | => npasser)
```

This infinitely gets the state from the memory cell starting in `p1`, then moving to `p2`, then moving ack to `p1`, and so on forever.

Races We discuss the `race` process command. This allows one to introduce controlled non-determinism into CMPL programs. If $\alpha_1, \ldots, \alpha_n$ are of *input polarity* with each α_i of type `Put(`A_i`|`S_i`)` (i.e., α_i is waiting for a sequential value of type A_i and proceeds as S_i), and $\beta_{n+1}, \ldots, \beta_m$ are of *output polarity* with each β of type `Get(`B_i`|`T_i`)` (i.e., β_i is waiting for a sequential value of type B_i and proceeds as T_i), then `race` command may be written as follows.

`race`
α_1 `->` $procBody_1$
$\vdots$
α_n `->` $procBody_n$
β_{n+1} `->` $procBody_{n+1}$
$\vdots$
β_m `->` $procBody_m$

Where we note that each α_i `->` $procBody_i$ and β_i `->` $procBody_i$ may be arbitrarily permuted. We say that $\alpha_1, \ldots, \alpha_n, \beta_{n+1}, \ldots, \beta_m$ are used in the `race` process command and hence must be in scope. Note that the `race` process command must be the last process command in a process command block. Operationally, since each channel α_i and β_j is waiting for an input, the first α_k or β_k (for some k) which receives an input makes the process proceed as $procBody_k$. In other words, this races all channels waiting for input, and proceeds as the process body for the channel which first receives the input.

A first example of using the race command is to execute a parallel or which waits for a `Bool` from two channels, and outputs the result of logical or upon those bools, but may output before it has received both bools if it knows the result of the other bool will not effect the final result of the logical or operation. Consider the following definition which receives two bools on channels `l` and `r`, and outputs the result of logical or on channel `output`.

```
proc pOr :: | Put(Bool|TopBot), Put(Bool|TopBot) => Put(Bool|TopBot) =
    | l,r => output -> race
        l -> helper( | l,r => output )  -- (1)
        r -> helper( | r,l => output )  -- (2)

proc helper :: | Put(Bool|TopBot),Put(Bool|TopBot) => Put(Bool|TopBot) =
    | winner, loser => output -> do
        get val on winner           -- (3)
        if val                      -- (4)
            then do
                put val on output   -- (5)
                get _ on loser      -- (6)
                close winner        -- (7)
                close loser         -- (8)
                halt output         -- (9)
            else do
```

```
    get nval on loser    -- (10)
    put nval on output   -- (11)
    close winner         -- (12)
    close loser          -- (13)
    halt output          -- (14)
```

We make some remarks on what this process is doing.

- At `(1)` and `(2)`, we execute the race and call `helper` with the winner of the race as the first argument of the input channels, and the loser as the second argument of the input channels. Of course, we pass the output channel as the only output channel to `helper` as well.
- At `(3)`, we get the `val` on the channel `winner` which has won the race. Indeed, this should essentially immediately return `val` since `winner` has won the race.
- At `(4)`, we test if `val` is true or false. If `val` is true, then we proceed to `(5)` and immediately output `val` on the channel `output`. Then, from `(6)` to `(9)`, we clean up the value remaining on `loser` and close all channels. But, if the test at `(4)` fails, then we proceed to `(10)` which gets the `loser`'s value and outputs that value to `output`. After, the program closes all the remaining channels.

This shows one program one may use a `race` for non-determinism.

Service channels Thus far, we have shown various constructs in CMPL, but one crucial part is missing – how does an CMPL program interact with the outside world? Currently, this is done through *service channels* which are channels in the main process of a special built-in protocol or coprotocol type that executes side effects. There are currently three service channels defined as follows.

```
-- This opens a terminal for which one may
-- output strings and get strings on.
protocol StringTerminal => S =
    StringTerminalGet :: Get([Char]|S) => S
    StringTerminalPut :: Put([Char]|S) => S
    StringTerminalClose :: TopBot => S

-- This allows console input and output of strings, and the
-- opening of new string terminals.
coprotocol S => Console =
    ConsolePut :: S => Get([Char]|S)
    ConsoleGet :: S => Put([Char]|S)
    ConsoleClose :: S => TopBot

    ConsoleStringTerminal :: S => S (*) Neg(StringTerminal)
```

```
-- This is a timer which waits a given number of microseconds
coprotocol S => Timer =
    -- Timer in microseconds
    Timer :: S => Get(Int|S (*) Put(()|TopBot))
    TimerClose :: S => TopBot
```

To use a service channel, one must `hput` the corresponding protocol (or coprotocol) handle on the service channel. Note that one may not `hcase` a service channel. Moreover, one may only put a service channel in scope the main process i.e., one may only put a service channel in scope in the distinguished process named `run`.

For example, to print `Hello world!`, we would use the built-in `Console` service protocol as follows.

```
coprotocol S => Console =
    ConsolePut :: S => Get([Char]|S)
    ConsoleGet :: S => Put([Char]|S)
    ConsoleClose :: S => TopBot

    ConsoleStringTerminal :: S => S (*) Neg(StringTerminal)

proc run :: | Console => =                  -- (1)
    | console => -> do                      -- (2)
        hput ConsolePut on console          -- (3)
        put "Hello world!" on console       -- (4)
        hput ConsoleGet on console          -- (5)
        get val on console                  -- (6)
        hput ConsoleClose on console        -- (7)
        halt console                        -- (8)
```

The interpretation of this program is as follows.

- `(1)` is the definition of the `run` process which is the distinguished function for which a CMPL program begins execution.
- `(2)` puts the input channel `console` in scope. Since `console` has type `Console` (as declared at `(1)`), we get that `console` is a special built-in service channel which may execute side effects.
- `(2)` and `(3)` put the handle `ConsolePut` on the channel `console`, then puts the string `"Hello world!"` on `console`. The intuition being captured here is that `console` is a channel which is connected to the real world, for which we send a message to the real world to declare that we want to put a message on the console (with `ConsolePut`) and the message we want to put on the console is `"Hello world!"`.
- `(5)` and `(6)` put the handle `ConsoleGet` on the channel `console`, then the user input string is bound to the variable `val`. Again, the intuition

being captured here is that `console` is a channel which is connected to the real world, for which we send a message to the real world to declare that we want to get a message on the console (with `ConsoleGet`) and this process will wait until we the real world sends the message of the string we are waiting to get back.

- `(7)` and `(8)` put the handle `ConsoleClose` on the channel `console`, and closes the channel `console`.

This shows how one can do side effects relating to the console in CMPL.

We will now discuss the example of using the `Timeout` coprotocol to wait for user input and tell the user if they are too slow are gave their input in time. Crucially, we use both the `Timeout` coprotocol and the `Console` coprotocol to execute the necessary side effects. Consider the following definitions.

```
coprotocol S => Console =
    ConsolePut :: S => Get([Char]|S)
    ConsoleGet :: S => Put([Char]|S)
    ConsoleClose :: S => TopBot

    ConsoleStringTerminal :: S => S (*) Neg(StringTerminal)

coprotocol S => Timer =
    Timer :: S => Get(Int|S (*) Put(()|TopBot))
    TimerClose :: S => TopBot

defn
  proc run :: | Console,Timer => =
      | console,timer => -> do              -- (1)
          hput ConsoleGet on console        -- (2)
          hput Timer on timer               -- (3)
          put 100000 on timer               -- (4)
          split timer into ntimer,timedval  -- (5)
          hput TimerClose on ntimer         -- (6)
          close ntimer                      -- (7)
          race                              -- (8)
              console ->
                  helper("On time!"| console,timer => )
              timedval ->
                  helper("Timed out!"| console,timer => )
where
  proc helper :: [Char] | Console,Timer => =
      message | console,timer => -> do
          get input on console              -- (9)
          hput ConsolePut on console        -- (10)
          put message on console            -- (11)
          hput ConsoleClose on console      -- (12)
```

```
close console                    -- (13)
get () on timedval               -- (14)
halt timedval                    -- (15)
```

We describe how this interaction works.

- `(1)` puts channels `console` and `timer` in scope. From their corresponding types, we know that the channel `console` is used for side effects relating to the console, and similarly, `timer` is used for side effects relating to sleeping a thread. We shall soon see how sleeping a thread can give us the ability to time out other threads in CMPL.
- `(2)` indicates that we want the channel `console` to wait for an input string from the console.
- `(3)` indicates that we want to use the `Timer` handle. The alternative is to use the `TimerClose` handle which ends all interactions which use the timer, but that is not what we want here.
- `(4)` indicates that we want to the timer to sleep for 100000 microseconds.
- `(5)` splits the timer into two new channels `ntimer` and `timedval`. `ntimer` is a new timer that one may use again, and `timedval` will receive a sequential value of type `()` (the empty tuple type) after `100000` microseconds.
- `(6)` and `(7)` close the channel `ntimer` since we only need one timer for this example.
- `(8)` races the channel `console` (which gets the user input) and the channel `timedval` (which gets the unit type after 100000 microseconds). Then, in either case, the process `helper` is called with differing strings indicating whether the user timed out or not. Indeed, if `console` wins the race, then this means that the user gave input in under 100000 microseconds. Whereas if `timedval` wins the race, this means that the user took at least 100000 microseconds to give user input.
- In the call to the process `helper`, from `(9)` to `(15)` we output whether the user timed out with their console input, and unwrap the types of the channels accordingly with the corresponding process commands.

Hence, this shows how one may use a timer and the built in CMPL process commands to implement a time out.

Summary We have discussed the concurrent constructs of CMPL and how one may develop some concurrent programs in CMPL.

2.3 General Constructs

We discuss some general structure about CMPL programs now. CMPL programs are read sequential from the top to the bottom, and constructs are put in scope top to bottom as well. Clearly, this makes mutually recursive definitions impossible to define, hence CMPL supports the introduction of mutually recursive definitions with the `defn` keyword. Moreover, `defn`s may have local definitions specified by the `where` keyword. Explicitly, these constructs are of the form

`defn`
$defn_1$
$\vdots$
$defn_m$
`where`
$whereDefn_1$
$\vdots$
$whereDefn_m$

Where the `where` binding and its definitions are optional, and $defn_1, \ldots, defn_n$ and $whereDefn_1, \ldots, whereDefn_m$ are arbitrary CMPL structures.

3 Future work

We discuss future work for improving the current implementation of CMPL.

User defined services We may recall that service channels of CMPL allow CMPL programs to execute side effects. In particular, there were three service channels which exhaustively described all side effects a CMPL program may make. Clearly, these three services do not totally encompass all side effects one may wish to write for their program – indeed, one may want to write their own service channels i.e., one would like to write a *user-defined service channel*.

The current implementation implements the three built in service channels by essentially doing a foreign function call in the host language Haskell to a C function call. Ideally, we would like to expose such an ability to CMPL as well. A method to achieve this would be to have a foreign function interface (abbr. FFI) in CMPL to allow CMPL programs to call functions from other languages such as C.

While this approach of developing an FFI is not new (see [9]), there are some design space issues that need to be filled in CMPL. In particular, there needs to be thought of the syntax for a foreign function call, and how one may wish to encode such foreign function calls as a protocol.

Type classes A classic problem in a Hindley/Milner type inference system is the overloading of functions based on their type. Precisely, ad-hoc polymor-

phism occurs when a function is defined over several types, acting in a different way for each type. We want to offer such methods of ad-hoc polymorphism in CMPL as well. Some classic examples of ad-hoc polymorphism include overloading of arithmetic operators which work on `Int`s and `Float`s, or an overloaded equality test `==` which works for numerous types. Type classes are one method to achieve this.

Type classes have been well studied in the sequential programming world in languages such as Haskell see [11, 8]. We would like develop a similar system in the context of CMPL.

Recalling that CMPL is a two-tiered programming language with a sequential aspect and a concurrent aspect. It should be relatively straightforward to refurbish the type class ideas in Haskell to the sequential part of CMPL, but the concurrent side of CMPL poses some design space difficulties of reinterpreting type classes a processes. In particular, there are some questions such as – how useful is it even have a type class of processes? And moreover, what would the syntax of this be for CMPL? All these ideas need a bit more careful thought before being implemented in CMPL.

Syntax changes Here, we discuss some issues in the syntax of CMPL. We list these issues in bullet point form.

- CMPL does not currently permit user defined operators, and hence this could be added to CMPL.
- CMPL does not permit explicit type signatures on expressions or channels. Indeed, sometimes programmers would like to be able to do this.
- CMPL does not permit binding of local variables in a process command block[2], so we would like to add some sort of "let" binding in a do block similar to Haskell.
- The `switch` statement will be changed to `when`. This reads a little bit nicer than switch.

Relaxing the plug command Currently, the plug command must connect processes along exactly one channel. But, there are scenarios for which one wants to just run two processes in parallel without plugging them together on a channel. Indeed, this can be done by simply plugging along a channel and immediately closing the channel in both processes, but this is a bit of a nuance and relaxing the plug rule to allow parallel composition of processes without plugging along a channel would achieve this as well without changing the formal semantics. Although, this makes the word "plug" a bit of a poor choice for this command, and hence may be renamed to "join" instead.

[2]Technically one can do this with the case construct, but it is a bit ugly

Generalizing the race command The intuition of the `race` command is to allow racing of channels along channels which are listening for an input. This idea was captured by forcing the type of the raced channels to be either a `Get(A|S)` or `Put(A|S)` (depending on polarity of the channel) to ensure the channels are all listening for an input. But there are other commands which for a process to wait for an input, namely processes may wait for a handle from an `hput`, or even wait for the new channels from a `split` command. Hence, we would like to generalize the `race` command to permit racing on all sorts of channels which are "listening" for an input. Indeed, this complicates type inference, and the mechanism for type classes may be one way to implement this feature.

Abstract Machine The current implementation of CMPL uses an abstract machine written with Concurrent Haskell [10] to execute CMPL programs. While Haskell is a beautiful language which makes implementation rather straightforward, all Haskell programs suffer from one fundamental problem – the global stop the world garbage collector (see [7]). In the context of CMPL, Haskell's default garbage collection shortcomings are not a sensible default for CMPL since in CMPL processes' memory is isolated from each other (except when messages are sent and received), so we should be able to isolate garbage collection to each process locally; thus avoiding Haskell's stop the entire world garbage collection. There is work on developing hybrid heaps of reference counted messages and local process heaps (see [3]) which can built upon for CMPL's abstract machine.

Unfortunately, undertaking a task like this would require much more low level control of how the programs are compiled which would essentially require a redesign of the abstract machine. There is work in [4] to demonstrate how one can compile high level languages directly to assembly as well – although this reference is concerned with the compilation of a lazy functional language which differs from CMPL (a by value language). But, should this route be taken, this would also give the opportunity to improve the performance of the current abstract machine which has high constant factors[3].

Module system Lastly, there is no module system in CMPL i.e., all CMPL programs must be written in one large file. Clearly, most users of a programming language would like to separate their code into different files, and hence this is a feature that would be beneficial to have in CMPL. This needs significantly more thought before proceeding.

References

[1] Luis Caires, Frank Pfenning, and Bernardo Toninho. "Linear logic propositions as session types". In: *Mathematical Structures in Computer Science* 26.3 (2016), pp. 367–423. DOI: 10.1017/S0960129514000218.

[3] A formal benchmark has not been done yet, but it feels noticeably slower than Haskell.

[2] J.R.B Cockett and Craig Pastro. “The logic of message-passing”. eng. In: *Science of Computer Programming* 74.8 (2009), pp. 498–533. ISSN: 0167-6423.

[3] Erik Johansson, Konstantinos Sagonas, and Jesper Wilhelmsson. “Heap Architectures for Concurrent Languages Using Message Passing”. In: *SIGPLAN Not.* 38.2 supplement (June 2002), pp. 88–99. ISSN: 0362-1340. DOI: 10.1145/773039.512440. URL: https://doi.org/10.1145/773039.512440.

[4] Peyton Jones, Simon L, and Simon Peyton Jones. “Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine”. In: *Journal of Functional Programming* 2 (July 1992), pp. 127–202. URL: https://www.microsoft.com/en-us/research/publication/implementing-lazy-functional-languages-on-stock-hardware-the-spineless-tagless-g-machine/.

[5] Prashant Kumar. *Implementation of Message Passing Language*. 2018. DOI: 10.11575/PRISM/10182. URL: https://prism.ucalgary.ca/handle/1880/106402.

[6] Reginald Lybbert. “Progress for the Message Passing Logic”. Undergraduate thesis. 2019.

[7] Simon Marlow, Tim Harris, and Simon Peyton Jones. “Parallel Generational-Copying Garbage Collection with a Block-Structured Heap”. In: *ISMM '08: Proceedings of the 7th International Symposium on Memory Management*. ACM, June 2008. URL: https://www.microsoft.com/en-us/research/publication/parallel-generational-copying-garbage-collection-with-a-block-structured-heap/.

[8] John Peterson and Mark Jones. “Implementing Type Classes”. In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. PLDI '93. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1993, pp. 227–236. ISBN: 0897915984. DOI: 10.1145/155090.155112. URL: https://doi.org/10.1145/155090.155112.

[9] Simon Peyton Jones. “Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell”. In: *Engineering Theories of Software Construction*. IOS Press, Jan. 2001, pp. 47–96. ISBN: ISBN 1 58603 1724. URL: https://www.microsoft.com/en-us/research/publication/tackling-awkward-squad-monadic-inputoutput-concurrency-exceptions-foreign-language-calls-haskell/.

[10] Simon Peyton Jones, Andy Gordon, and Sigbjorn Finne. “Concurrent Haskell”. In: *POPL '96 Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, Jan. 1996, pp. 295–308. URL: https://www.microsoft.com/en-us/research/publication/concurrent-haskell/.

[11] P Wadler and S Blott. “How to make ad-hoc polymorphism less ad hoc”. eng. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. ACM, 1989, pp. 60–76. ISBN: 0897912942.