

Higher-Order Concurrent Processes for Implementing Recursion Patterns

October/Novemberfest 2026

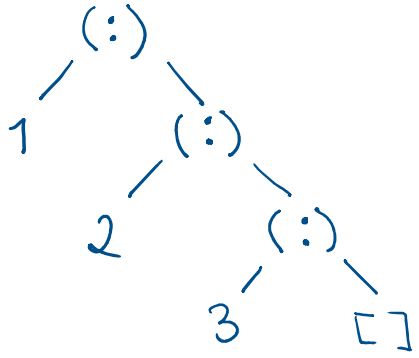
Alexanna (Xanna) Little Berg
Supervised by Professor Robin Cockett
University of Calgary

Outline

- Sequential functional programming background in Haskell
  - Inductive data types, e.g. natural numbers, lists, trees
  - Higher-order functions for patterns of recursion: map, fold
  - Connection to initial functor algebras
  - Dual notions of coinductive codata types, unfold, final coalgebras
- Concurrent analogues in CaMPL
  - Using the functor algebra perspective to define protocols and coprotocols
  - Higher-order processes for patterns of recursion: drive
  - Application to constructing arbitrarily large process networks

Inductive data types

- Recursively defined data structures.
- Constructors for base case(s) and recursive case(s).
- Examples: Natural numbers, Lists, Trees.
- List constructors are “nil” which is the empty list `[ ]::  ->[X]`, and “cons” which adds an element to a list `(:)::X->[X]->[X]`.
- We can express the list `[1,2,3]` using these as `(1:(2:(3:[ ])))`.



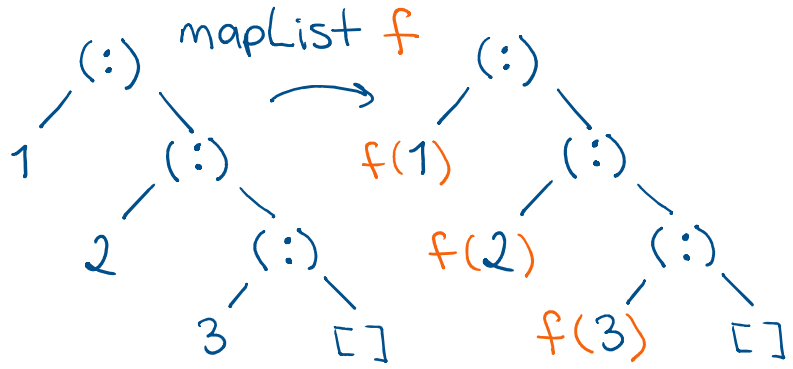
Higher-order functions

- Take other functions as input and/or return functions as output.
- Allow you to **abstract a pattern of recursion** and apply it to different computations by inputting different function(s).
- For example, if we have a list of integers, we might want to apply a function f to each integer.
- Regardless of the function being applied, we need to recursively visit each element, apply the function, and add it back to the list.
- A higher-order function can take an arbitrary function f as input and follow this pattern of recursion.

Map

- Follows the pattern of recursion described on the previous slide.
- For an inductive data type, apply a function to each value without changing the structure of the data type.
- For example, `mapList::(X -> Y) -> [X] -> [Y]`

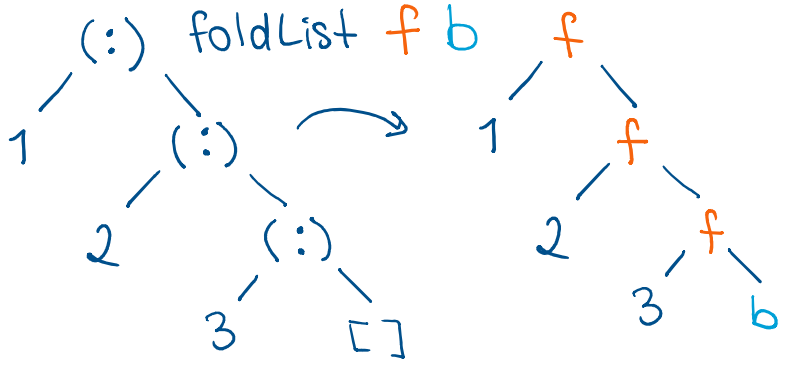
```
mapList f [ ] = [ ]
mapList f (x:xs) = (f x):(mapList f xs)
```



Fold

- For an inductive data type, replace constructors with given functions to accumulate data into a single return value.
- For example, `foldList::(X -> Y -> Y) -> Y -> [X] -> Y`

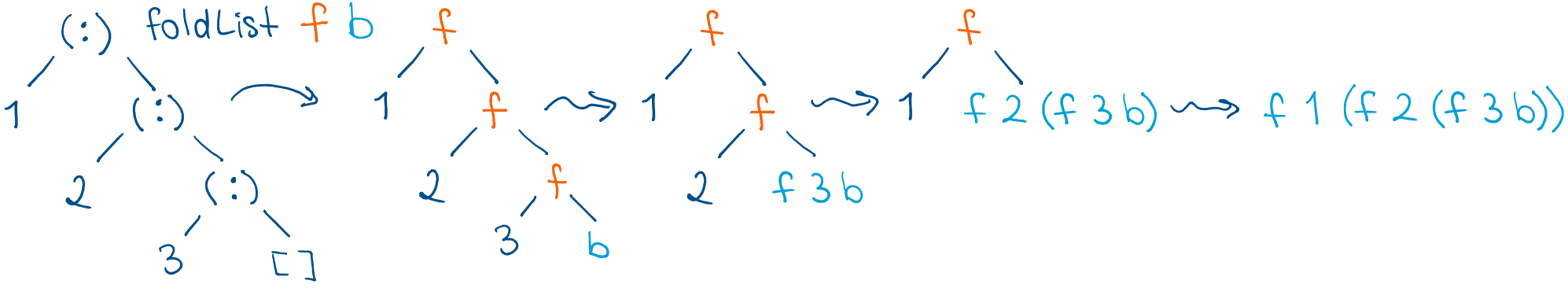
```
foldList f b [] = b
foldList f b (x:xs) = f x (foldList f b xs)
```



Fold

- For an inductive data type, replace constructors with given functions to accumulate data into a single return value.
- For example, `foldList::(X -> Y -> Y) -> Y -> [X] -> Y`

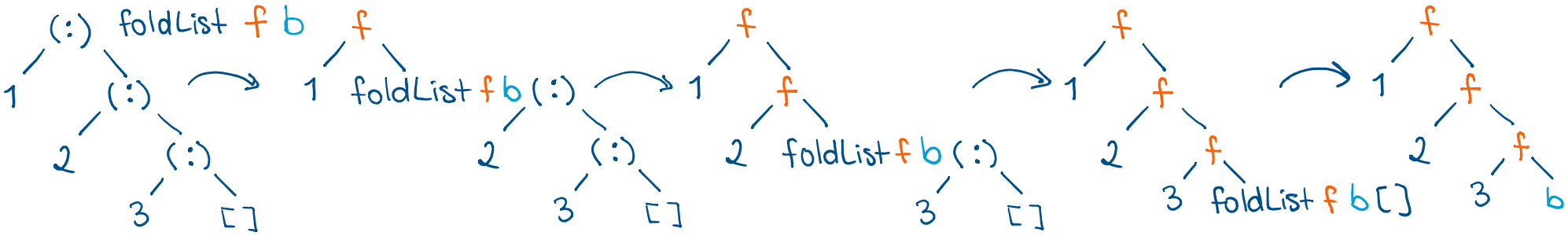
```
foldList f b [] = b
foldList f b (x:xs) = f x (foldList f b xs)
```



Fold

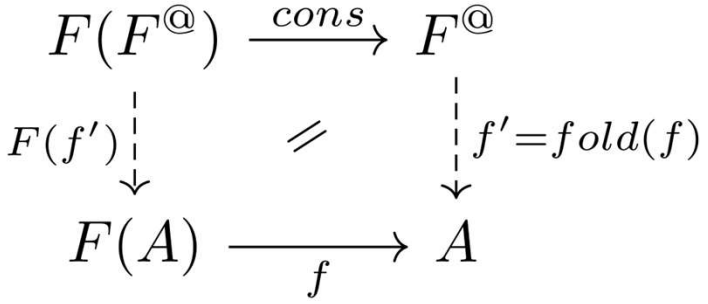
- For an inductive data type, replace constructors with given functions to accumulate data into a single return value.
- For example, `foldList::(X -> Y -> Y) -> Y -> [X] -> Y`

```
foldList f b [] = b
foldList f b (x:xs) = f x (foldList f b xs)
```



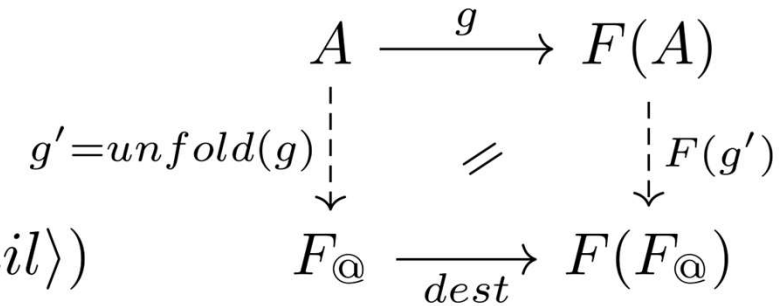
Functor algebras (Lambek 1968, Hagino 1987, Barr 1990)

- An F-algebra, for a functor F, is a pair $(A, f : F(A) \to A)$.
- An initial F-algebra $(F^@, cons)$ has a unique map to every F-algebra.
- An inductive data type and its constructor form an initial algebra, and the unique map is given by fold.
- Natural numbers $(\mathbb{N}, \langle succ | zero \rangle)$ are initial for $F(A) = A + 1$.
- Lists $(L(X), \langle cons | nil \rangle)$ are initial for $F(A) = (X \times A) + 1$, where X can be any data type.



Duals: codata, unfold, and functor coalgebras

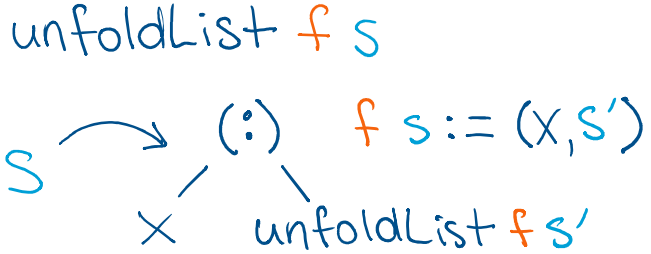
- Codata can represent possibly infinite data.
- An F-coalgebra, for a functor F, is a pair $(A, g : A \to F(A))$.
- A final F-coalgebra $(F_@, dest)$ has a unique map from every F-coalgebra.
- A coinductive codata type and its destructor form a final coalgebra, and the unique map is given by unfold.
- Conatural numbers $(\mathbb{N}^\infty, pred)$ are final for $F(A) = A + 1$.
- Possibly infinite lists $(IL(X), \langle head, tail \rangle)$ are final for $F(A) = (X \times A) + 1$.



Unfold

- For a coinductive codata type, generate an instance of the structure using a given function and seed value.
- In object-oriented programming, creating an instance of an object.
- For example, lazy evaluation in Haskell allows for possibly infinite lists which means lists are also a codata type with an unfold:

```
unfoldList::(Y -> Maybe(X,Y)) -> Y -> [X]
unfoldList f s = case (f s) of
     Just(x,s') -> x:(unfoldList f s')
     Nothing -> [ ]
```

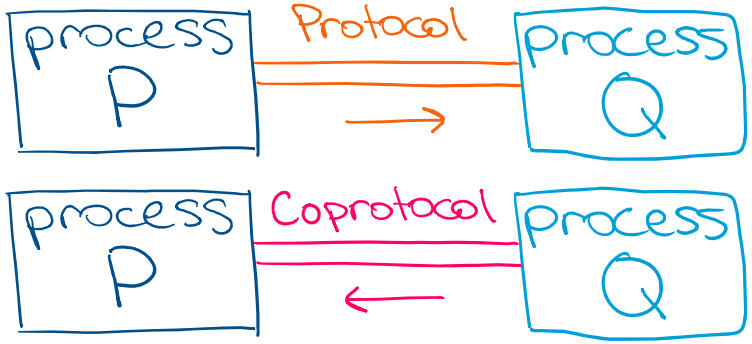


Concurrent analogues (Cockett, Pastro 2009, Yeasin 2012, Cockett, Hashimoto, Little Berg, Srinivasan 2026)

- Categorical Message Passing Language (CaMPL) is a functional-style concurrent programming language in which the primary communication method between concurrent processes is by passing messages along two-way communication channels.
- **Concurrent types are channel types**, rather than data types, and they **define a structure on channels**.
- The built-in channel types in CaMPL define the basic interactions that processes are permitted to have along a channel.
- User-defined channel types are defined by combining built-in types.

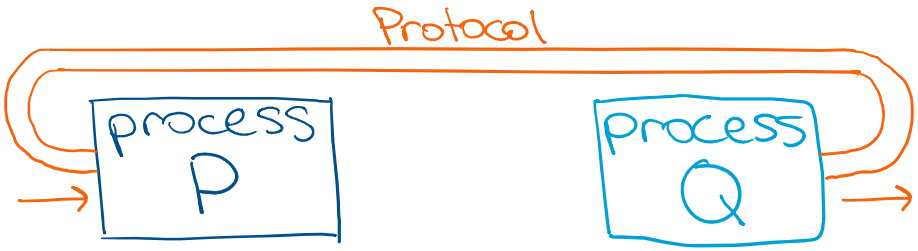
Protocols and coprotocols

- The concurrent analogue of data and codata types are protocol and coprotocol channel types.
- However, unlike sequential types, **both protocols and coprotocols** can **define possibly infinite channel structures**.
- **Protocols and coprotocols are symmetric**: a protocol can be turned into a coprotocol by negating it, i.e. by reversing its direction.



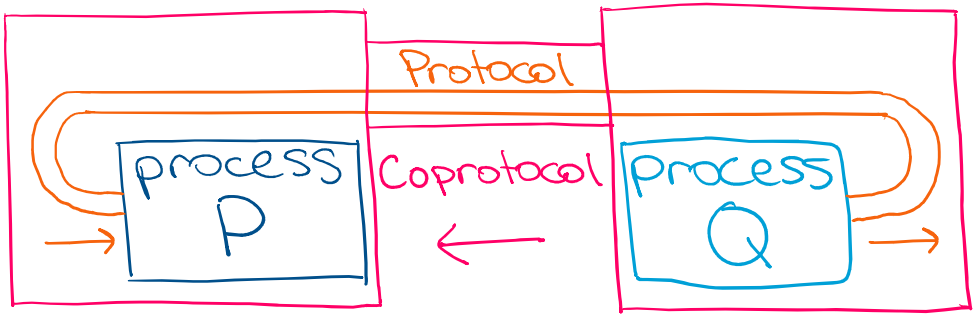
Protocols and coprotocols

- The concurrent analogue of data and codata types are protocol and coprotocol channel types.
- However, unlike sequential types, **both protocols and coprotocols** can **define possibly infinite channel structures**.
- **Protocols and coprotocols are symmetric**: a protocol can be turned into a coprotocol by negating it, i.e. by reversing its direction.



Protocols and coprotocols

- The concurrent analogue of data and codata types are protocol and coprotocol channel types.
- However, unlike sequential types, **both protocols and coprotocols** can **define possibly infinite channel structures**.
- **Protocols and coprotocols are symmetric**: a protocol can be turned into a coprotocol by negating it, i.e. by reversing its direction.



Example: PList protocol

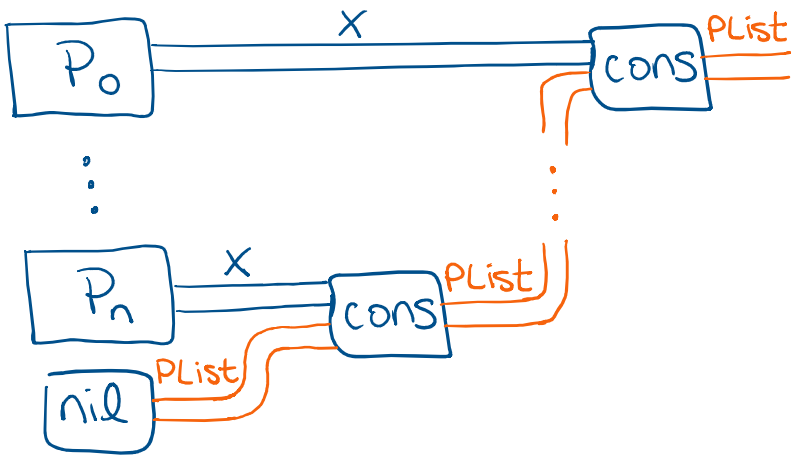
- The `PList` protocol constructs a tensor of arbitrarily many channels:

```
protocol PList( |X) => S =
    PListNil :: TopBot => S
    PListCons :: X(*)S => S
```

- Constructors are processes:

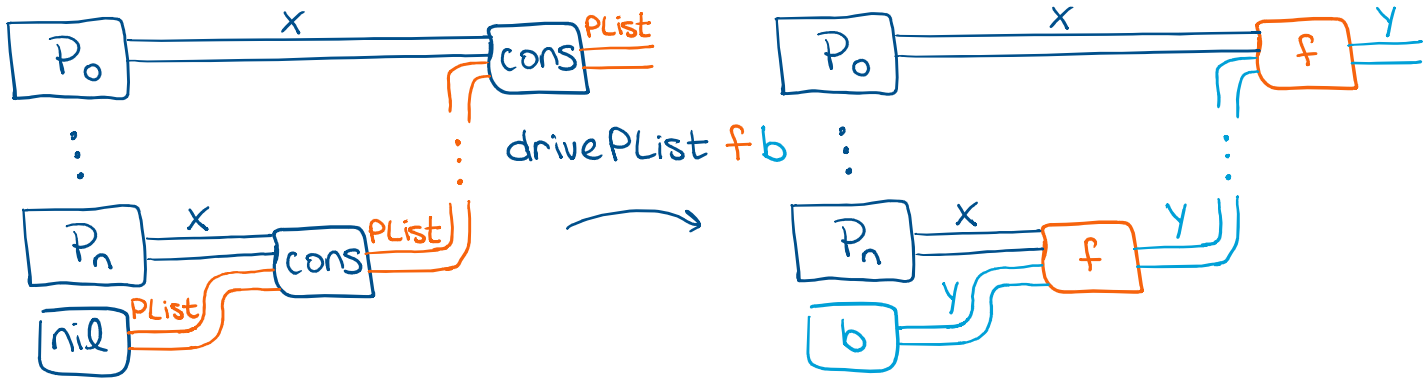
```
nil::| =>PList( |X)
cons::|X,PList( |X)=>PList( |X)
```

- The constructor processes create a linked list process network.



What is the "fold" for a protocol?

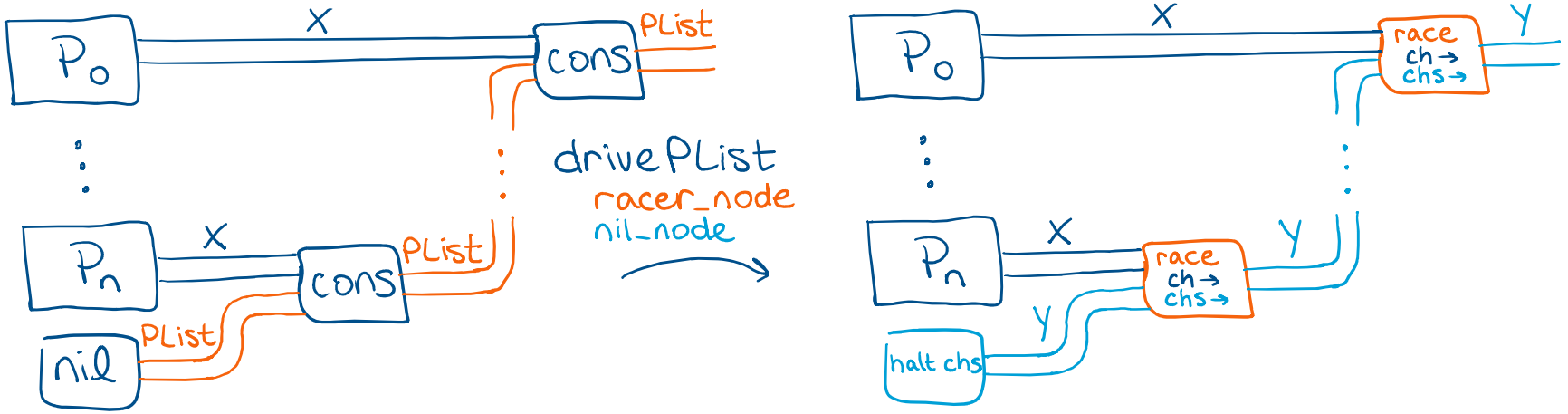
- In the sequential case, a fold is a higher-order function that replaces the constructors of an inductive data type with given functions.
- In the concurrent case, **a higher-order process**, called a **drive**, replaces the constructor processes of a protocol with given processes:



- Since coprotocols are simply protocols with the direction reversed, a codrive on a coprotocol is almost identical to a drive on a protocol.

Example: Using drivePList to race arbitrarily many channels

- In CaMPL, the `race` command provides controlled non-determinism.
- A race can only be held for a fixed number of channels.
- To race arbitrarily many channels, we can use `drivePList` to replace each `cons` in the linked list with a process that races its own channel with the rest of the list and forwards messages up the chain.



Other examples

- We have also defined a `PTree` protocol and `drivePTree` process. This constructs a process network that looks like a binary tree.
- Future work includes:
  - Developing use cases for concurrent analogues.
  - Writing programs in which process networks with arbitrarily many processes pass messages or resources to each other rather than only up the chain/tree.
  - Develop the theory to show we can guarantee properties such as deadlock and livelock freedom if programs are written using these structured recursive patterns (in the sequential case we are able to guarantee termination).

References



- Michael Barr and Charles Wells. Category Theory for Computing Science (Prentice-Hall, Englewood Cliffs, NJ, 1990).
- Robin Cockett and Craig Pastro. “The logic of message-passing.” Science of Computer Programming 74.8 (2009), pp. 498–533.
- Robin Cockett, Daniel Kiyoshi Hashimoto, Alexanna Little Berg, and Priyaa Varshinee Srinivasan. “Categorical Message Passing Language (CaMPL): Syntax and Semantics.” arXiv:2609.24436 [cs.PL] (2026).
- Tatsuya Hagino. “A Categorical Programming Language.” PhD thesis. University of Edinburgh, Sept. 1987. url: https://web.sfc.keio.ac.jp/~hagino/thesis.pdf.
- Joachim Lambek. “A fixpoint theorem for complete categories”. Mathematische Zeitschrift, 103.1 (1968), pp. 151–161.
- Masuka Yeasin. “Linear Functors and their Fixed Points.” Master’s Thesis. University of Calgary, Dec. 2012. url: https://cspages.ucalgary.ca/~robin/Theses/masuka_thesis.pdf.