Type Classes in CaMPL

Melika Norouzbeygi
Supervisor: Prof. Robin Cockett
July 2024

1

What are Type Classes and Why Are They Important?

We first need to talk about overloading

Overloading

- An operator is overloaded if it has two (or more) implementations, distinguished by type of its arguments and its output.
- In many languages, arithmetic operators (like ‘+’) have multiple distinct implementations for different types such as Int or Double.
- The decision on what implementation to use is made at compile time.
- In a language which has type inference and polymorphism, implementing "overloading" is more complicated:
    - When one defines a function, one should be able to use overloaded operators in the definition. However, this can cause the type of the overloaded operators to be polymorphic.

Type Classes: Haskell's Solution to Overloading

- Haskell uses type classes for implementing overloading.
- A type class declares a set of operations based on a type variable.
- An instance of a type class provides an implementation for the type class operations.
- If the type class operator is used in a function definition, the type need to be tagged with the type class name to indicate that the compiler must convert the operator into an instance.

Haskell

```haskell
class Eq a where
    (==) :: a -> a -> Bool

instance Eq Int where
    (==) :: Int -> Int -> Bool
    a == b = eq_int a b
```

```haskell
instance Eq a => Eq [a] where
    (==) :: Eq a => [a] -> [a] -> Bool
    [] == [] = True
    (x:xs) == (y:ys) = (x == y) && (xs == ys)
    _ == _ = False
```

Type Classes: Not Only Overloading!

- The benefit of type classes is not only for providing overloading.
- Type Classes can allow succinct and more understandable programs.
- One can make type classes depend on type constructors: this adds further power to type classes ...

Haskell

```haskell
class Functor f where
    fmap :: (a → b) → f a → f b
```

```haskell
instance Functor [] where
    fmap :: (a → b) → [a] → [b]
    fmap f [] = []
    fmap f (x:xs) = f x : fmap f xs
```

More Examples of Higher Order Type Classes

- One can define the Monad type class. In order for a type constructor to be a Monad it needs to be a Functor.
- One can make a type constructor (e.g List) an instance of Monad type class by implementing the return and sequence operations for it.

Haskell

```
class Functor m ⇒ Monad m where
    return :: a → m a
    (>>=) :: m a → (a → m b) → m b
```

```
instance Monad [] where
    return :: a → [a]
    return x = [x]

    (>>=) :: [a] → (a → [b]) → [b]
    xs >>= f = concatMap f xs
```

How are Type Classes Implemented?

- At compile time, after type inference, any usage of type class operators is translated to the core Haskell.
- Each type class operator is an input function to the function that uses it, and type class tags are also converted to the type of the function instance.

Haskell

```haskell
member :: Eq a ⇒ [a] → a → Bool
member list x = case list of
    [] → False
    (y:ys) → y == x || member ys x

r = member [1,2,3] 1
```

```haskell
--translation
member' :: (a → a → Bool) → [a] → a → Bool
member' = \eq_a list x → case list of
    [] → False
    (y:ys) → eq_a y x || member' eq_a ys x

r' = member' eq_int [1,2,3] 1
```

2

What is CaMPL?

Categorical Message Passing Language

CaMPL: Categorical Message Passing Language

- CaMPL is a polymorphic concurrent language with type inference.
- Its sequential tier is an implementation of lambda-calculus with data types.
- Its concurrent tier is an implementation of a linear actegory (a linear actegory is given by a monoidal category acting on a linearly distributive category).
- CaMPL was implemented by Robin Cockett, Prashant Kumar and Jared Pon at the University of Calgary.

Say “Hello World!” in CaMPL

- The helloworld process has an output channel of type StringTerminal.
- It puts the string “Hello World” on the terminal
- At the end it halts the terminal.

CaMPL

```
proc helloworld :: | => StringTerminal =
    | => terminal -> do
       hput StringTerminalPut on terminal
       put "Hello World! press any key to exit..." on terminal
       hput StringTerminalGet on terminal
       get input on terminal
       hput StringTerminalClose on terminal
       halt terminal
```

```
proc run :: | => StringTerminal =
     | => terminal -> helloworld( | => terminal)
```

Categorical Semantics of the Concurrent Part

The categorical semantics of the concurrent side of CaMPL is defined by a linearly distributive category:

- Objects are concurrent channel types.
- Maps are concurrent processes (from input polarity to output polarity channels).
- Identity is a channel.
- Composition is given by plugging two processes to each other.
- The $\otimes$ and $\oplus$ functors allow bundling the channels together.

Categorical Semantics of the Sequential Part

The categorical semantics of the sequential side of CaMPL is defined by a cartesian closed category with data types:

- Objects are sequential types
- Maps are sequential functions.
- Identity is an identity function
- Composition is given by composition of functions

As for a functional language

Categorical Semantics of Message Passing

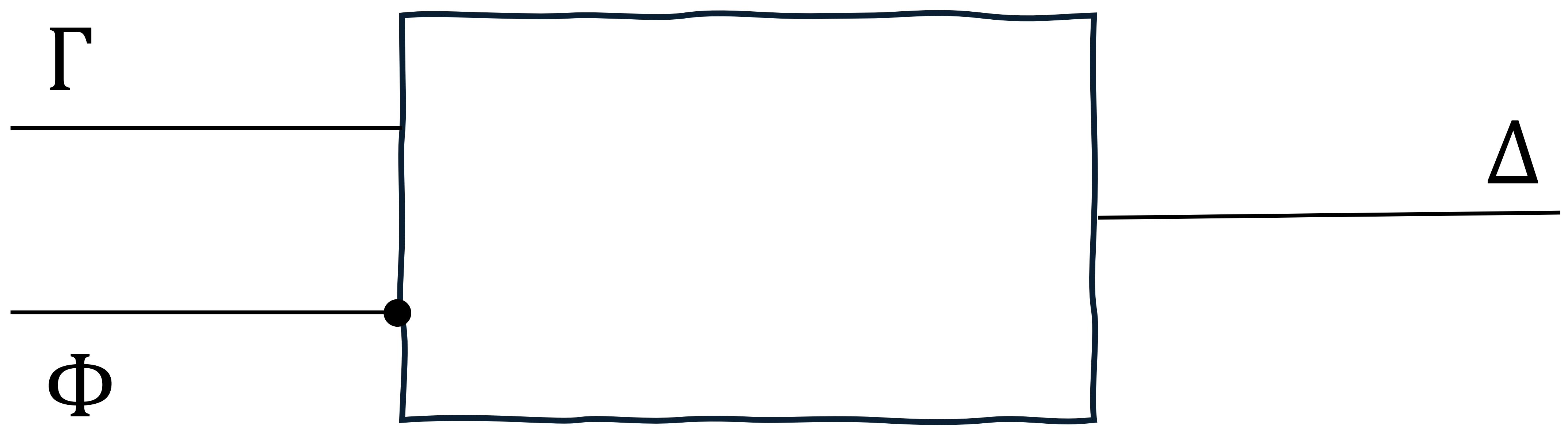
The categorical semantics of the message passing is a linear actegory:

- It consists of a cartesian closed category $\mathbb{S}$ (sequential part) acting on a linearly distributive category $\mathbb{C}$ (concurrent part).
- There are two functors for $\mathbb{S}$ acting on $\mathbb{C}$:
  - ○ (:= Put) : $\mathbb{S} \times \mathbb{C} \to \mathbb{C}$
  - ● (:= Get) : $\mathbb{S}^{op} \times \mathbb{C} \to \mathbb{C}$

Proof Theory of CaMPL

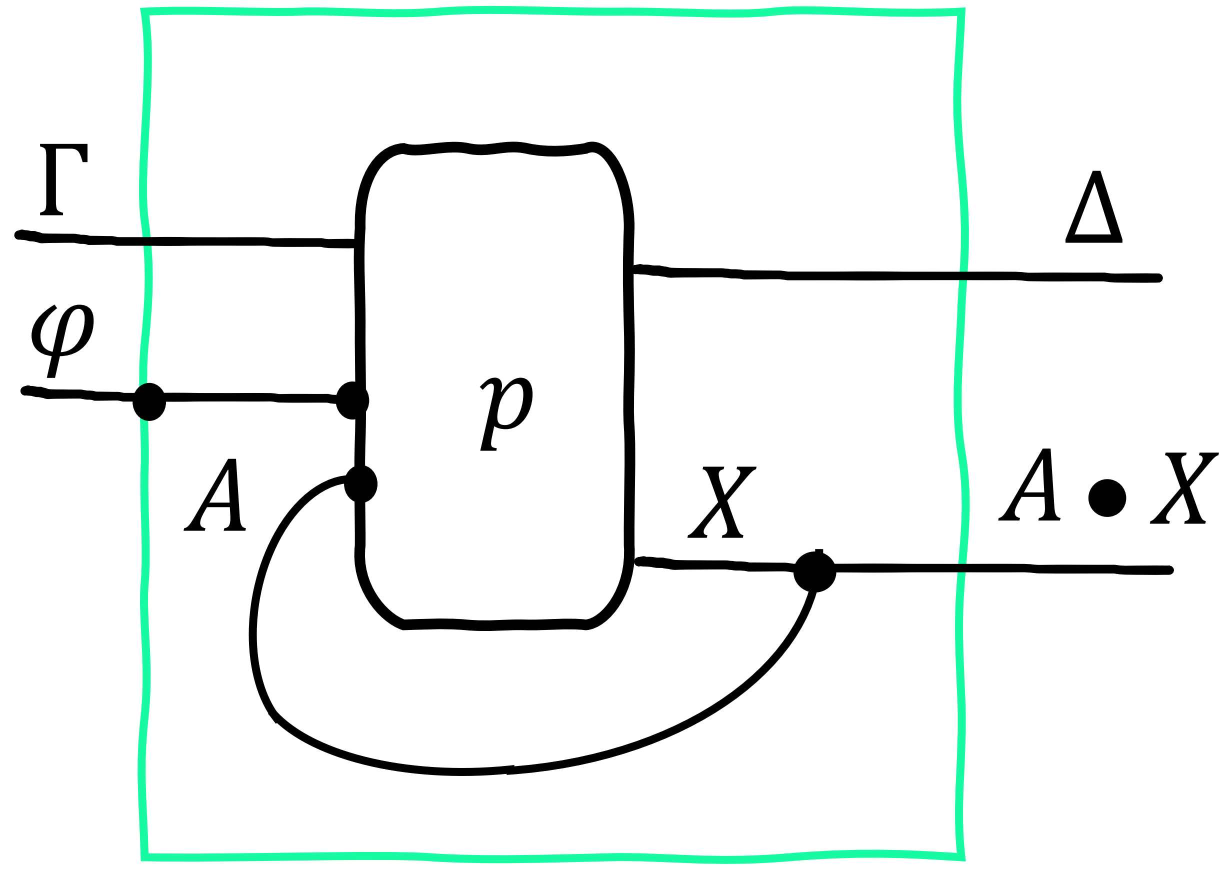
- The concurrent side of CaMPL is specified by the proof theory of linear actegories.
- It is specified by inference rules for concurrent sequents.
- Programming features of CaMPL can also be described equivalently by circuit diagrams.

$$\Phi \mid \Gamma \Vdash \Delta$$



Summary of CaMPL Concurrent Constructs

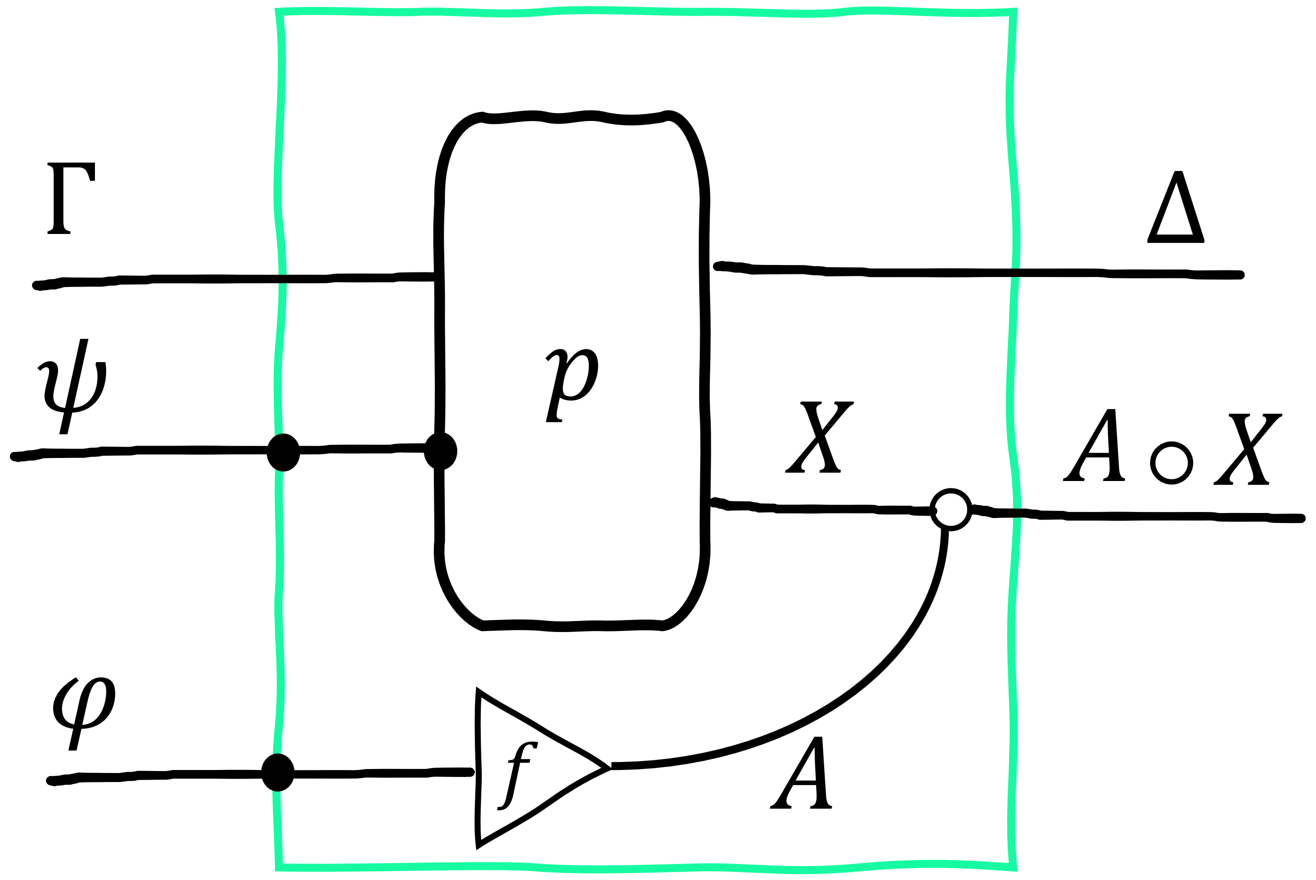
$$\frac{\Phi, A \mid \Gamma \Vdash X, \Delta}{\Phi \mid \Gamma \Vdash A \bullet X, \Delta} \bullet_r$$



CaMPL

```
proc q ::
   Phi | Gamma => Delta, Get(A|X) =
     phi | gamma => delta, alpha -> do
        get a on alpha
        p(phi, a | gamma => delta, alpha)
```

$$\frac{\Phi \vdash A \quad \Psi \mid \Gamma \Vdash X, \Delta}{\Phi, \Psi \mid \Gamma \Vdash A \circ X, \Delta} \circ_r$$

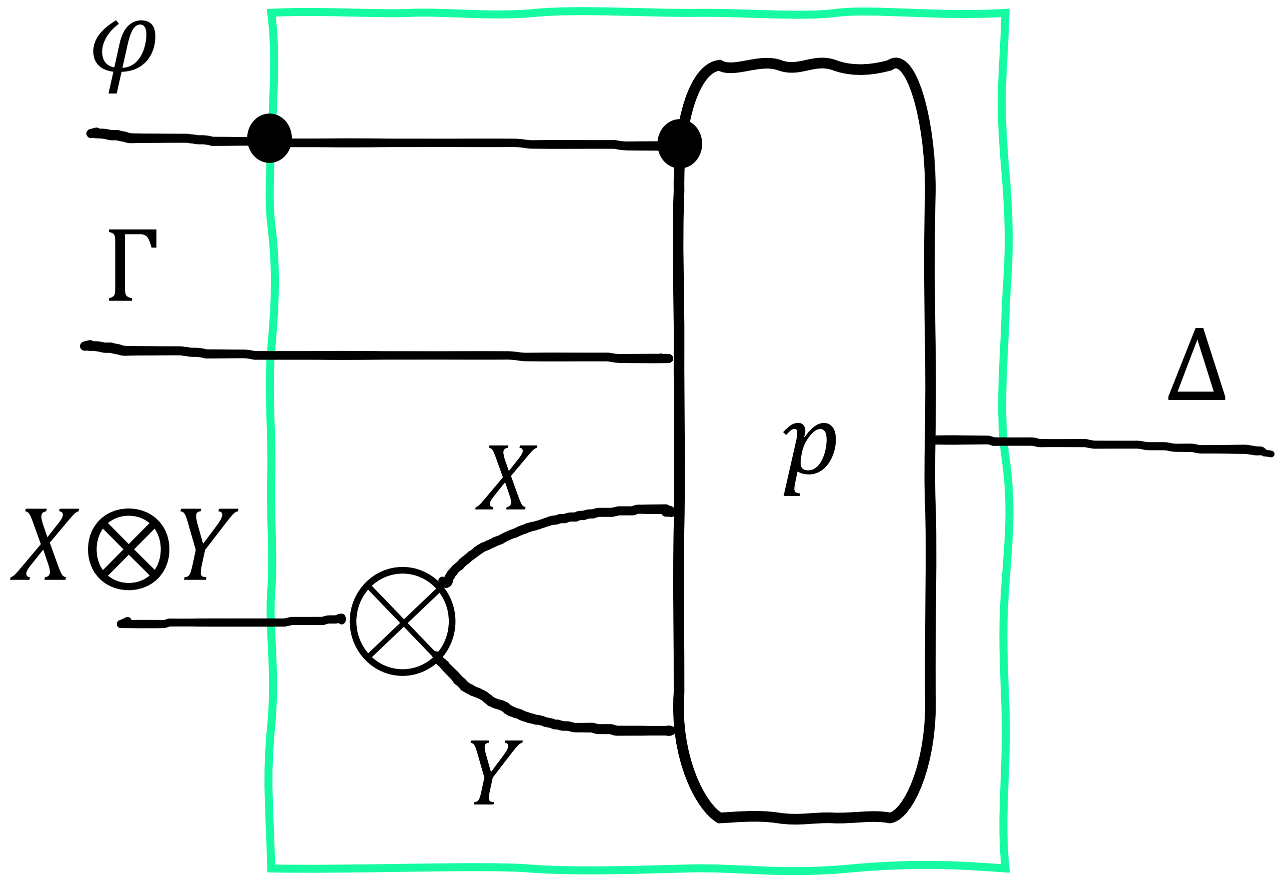


CaMPL

```
proc q ::
   A, Phi | Gamma => Delta, Put(A|X) =
     a, phi | gamma => delta, alpha -> do
        put a on alpha
        p(phi | gamma => delta, alpha)
```

Summary of CaMPL Concurrent Constructs

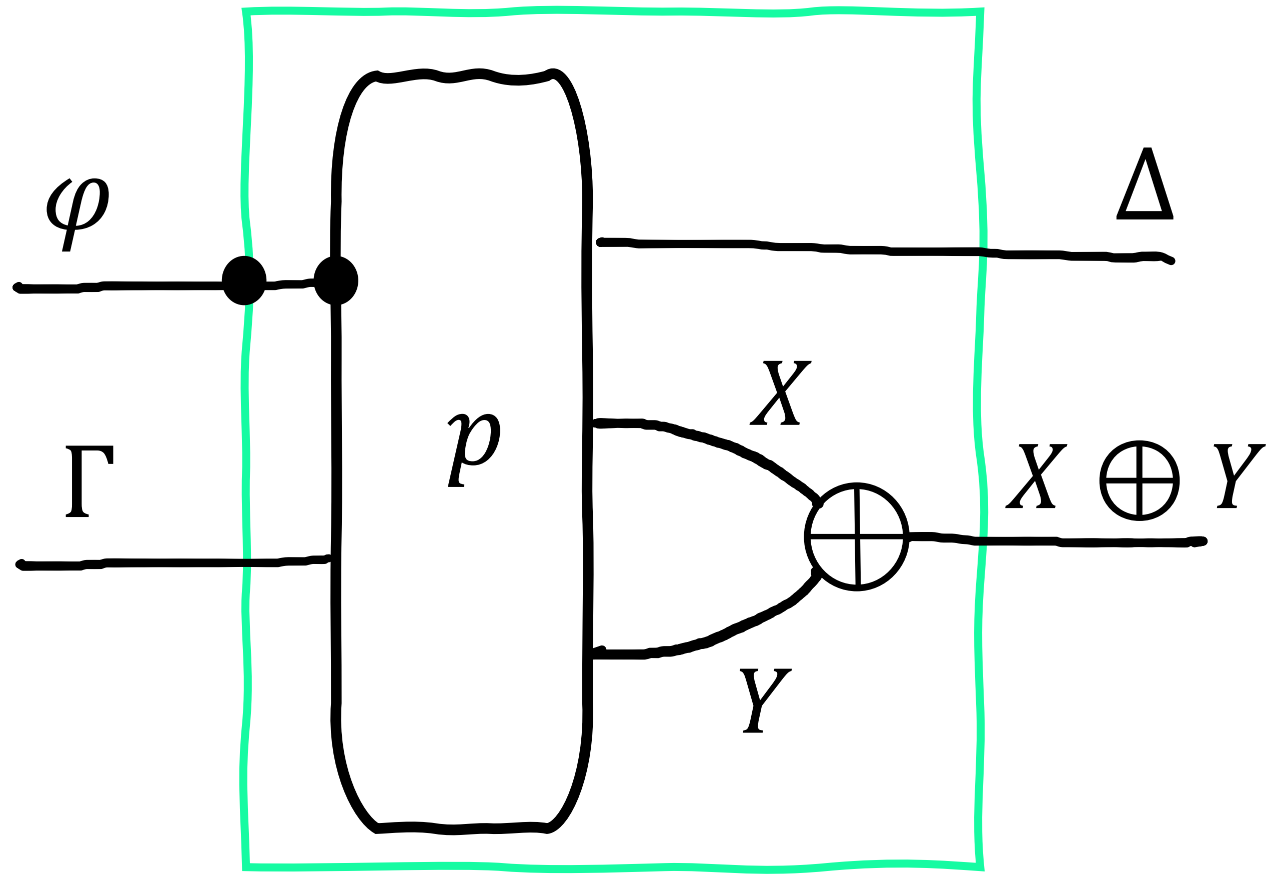
$$\frac{\Phi \mid \Gamma, X, Y \Vdash \Delta}{\Phi \mid \Gamma, X \otimes Y \Vdash \Delta} \otimes_{\ell}$$



CaMPL

```
proc q ::
  Phi | Gamma, X (*) Y => Delta =
   phi | gamma, alpha => delta -> do
     split alpha into aplpha1, alpha2
     p(phi| gamma, alpha1, alpha2 => delta)
```

$$\frac{\Phi \mid \Gamma \Vdash X, Y, \Delta}{\Phi \mid \Gamma \Vdash X \oplus Y \Delta} \oplus_{r}$$



CaMPL

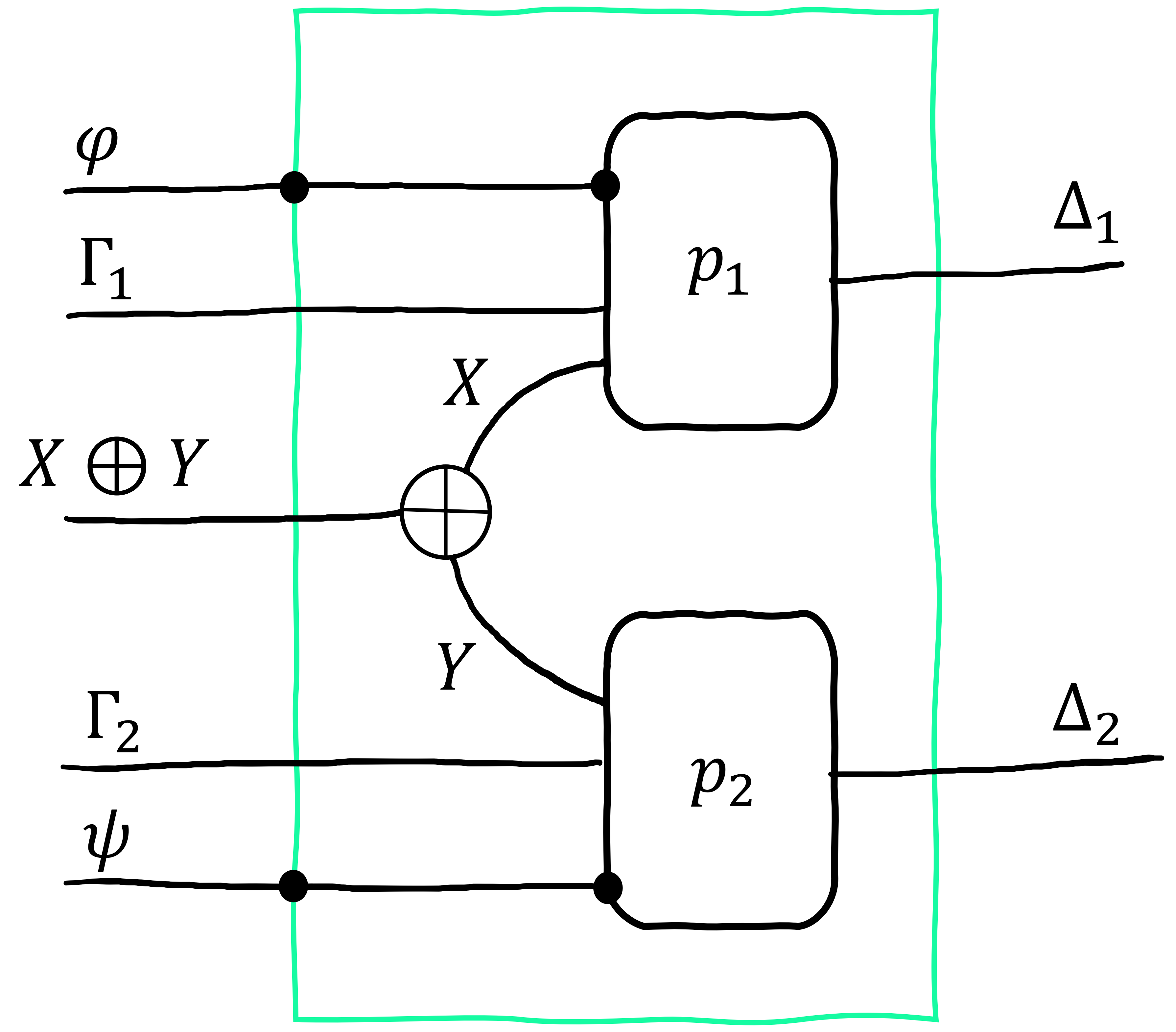
```
proc q ::
  Phi | Gamma => Delta, X (+) Y =
   phi | gamma => delta, alpha -> do
     split alpha into aplpha1, alpha2
     p(phi| gamma => delta, alpha1, alpha2)
```

Summary of CaMPL Concurrent Constructs

$$\frac{\Phi \mid \Gamma_1, X \Vdash \Delta_1 \quad \Psi \mid Y, \Gamma_2 \Vdash \Delta_2}{\Phi, \Psi \mid \Gamma_1, X \oplus Y, \Gamma_2 \Vdash \Delta_1, \Delta_2} \oplus_\ell$$

CaMPL

```
proc q ::
  Phi, Psi | Gamma1, Gamma2 => X (*) Y, Delta1, Delta2 =
   phi | gamma1, gamma2 => alpha, delta1, delta2 -> do
     fork alpha as
        alpha1 -> p1(psi|gamma1 => delta1, alpha1)
        alpha2 -> p2(phi|gamma2 => delta2, alpha2)
```

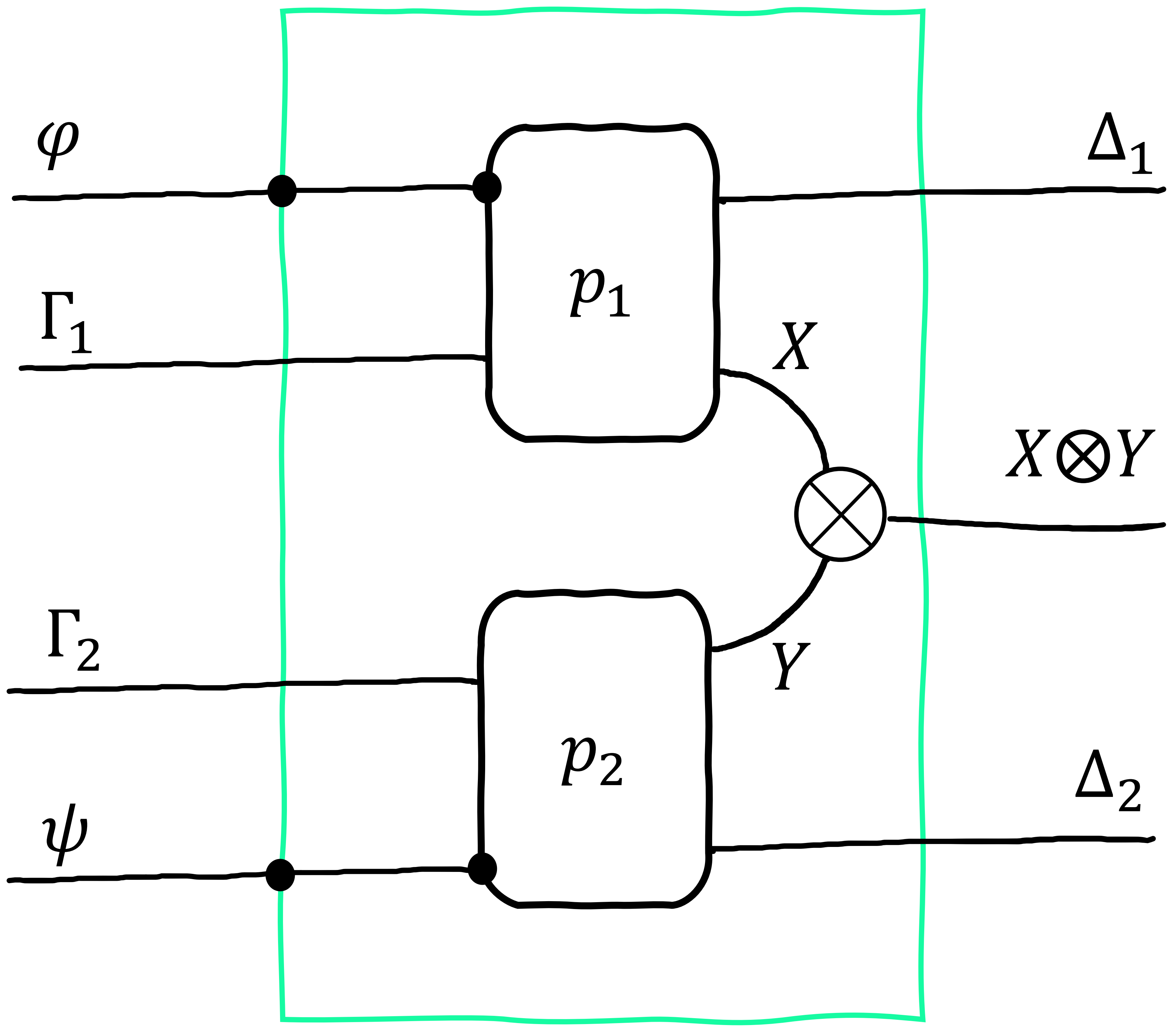


Summary of CaMPL Concurrent Constructs

$$\frac{\Phi \mid \Gamma_1 \Vdash \Delta_1, X \qquad \Psi \mid \Gamma_2 \Vdash Y, \Delta_2}{\Phi, \Psi \mid \Gamma_1, \Gamma_2 \Vdash \Delta_1, X \otimes Y, \Delta_2} \otimes_r$$

CaMPL

```
proc q ::
 Phi, Psi | Gamma1, Gamma2, X (+) Y => Delta1, Delta2 =
   phi | gamma1, gamma2, alpha => delta1, delta2 -> do
     fork alpha as
        alpha1 -> p1(psi|gamma1 => delta1, alpha1)
        alpha2 -> p2(phi|gamma2 => delta2, alpha2)
```

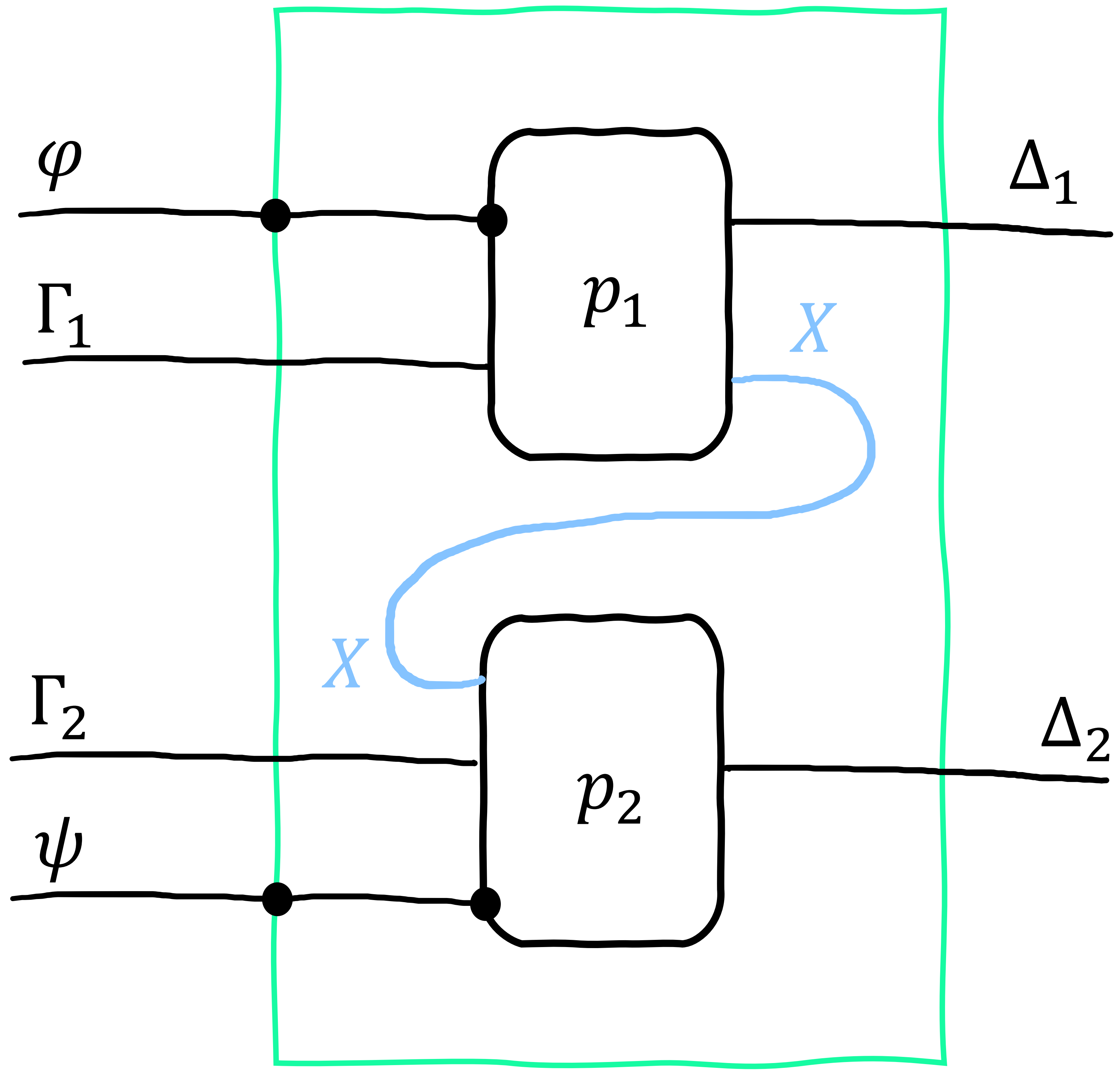


Summary of CaMPL Concurrent Constructs

$$\frac{\Phi \mid \Gamma_1 \Vdash \Delta_1, X \qquad \Psi \mid X, \Gamma_2 \Vdash \Delta_2}{\Phi, \Psi \mid \Gamma_1, \Gamma_2 \Vdash \Delta_1, \Delta_2} \text{ cut}$$

CaMPL

```
proc q ::
   Phi, Psi | Gamma1, Gamma2 => Delta1, Delta2 =
       phi, psi | gamma1, gamma2 => delta1, delta2 ->
           plug
              p1(phi | gamma1 => delta1, x)
              p2(psi | gamma2, x => delta2)
```



3.1

How to add type classes to the sequential CaMPL?

Inspired by Haskell

Type Classes in Sequential CaMPL

For the sequential side of CaMPL as it is a functional-style language one can use the same approach to type classes as Haskell.

CaMPL

```
proc getFromTerminal ::
  Parse A ⇒ | Get(A | TopBot) ⇒ StringTerminal =
    | ch ⇒ strterm → do
      hput StringTerminalPut on strterm
      put "enter something" on strterm
      hput StringTerminalGet on strterm
      get input on strterm

      case parse(input) of
       Just(a_val) → do
           put a_val on ch
       Nothing → getFromTerminal(|ch ⇒ strterm)
```

CaMPL

```
-- translation
proc getFromTerminal ::
  Fun([Char],Maybe(A))| Get(A | T) ⇒ StringTerminal =
    parseA |ch ⇒ strterm → do
      hput StringTerminalPut on strterm
      put "enter something" on strterm
      hput StringTerminalGet on strterm
      get input on strterm

      case App(input, parseA) of
        Just(a_val) → do
          put a_val on ch
        Nothing → getFromTerminal(parseA|ch ⇒ strterm)
```

3.2 What about concurrent type classes?

As far as we know, there is no implementation for concurrent type classes …

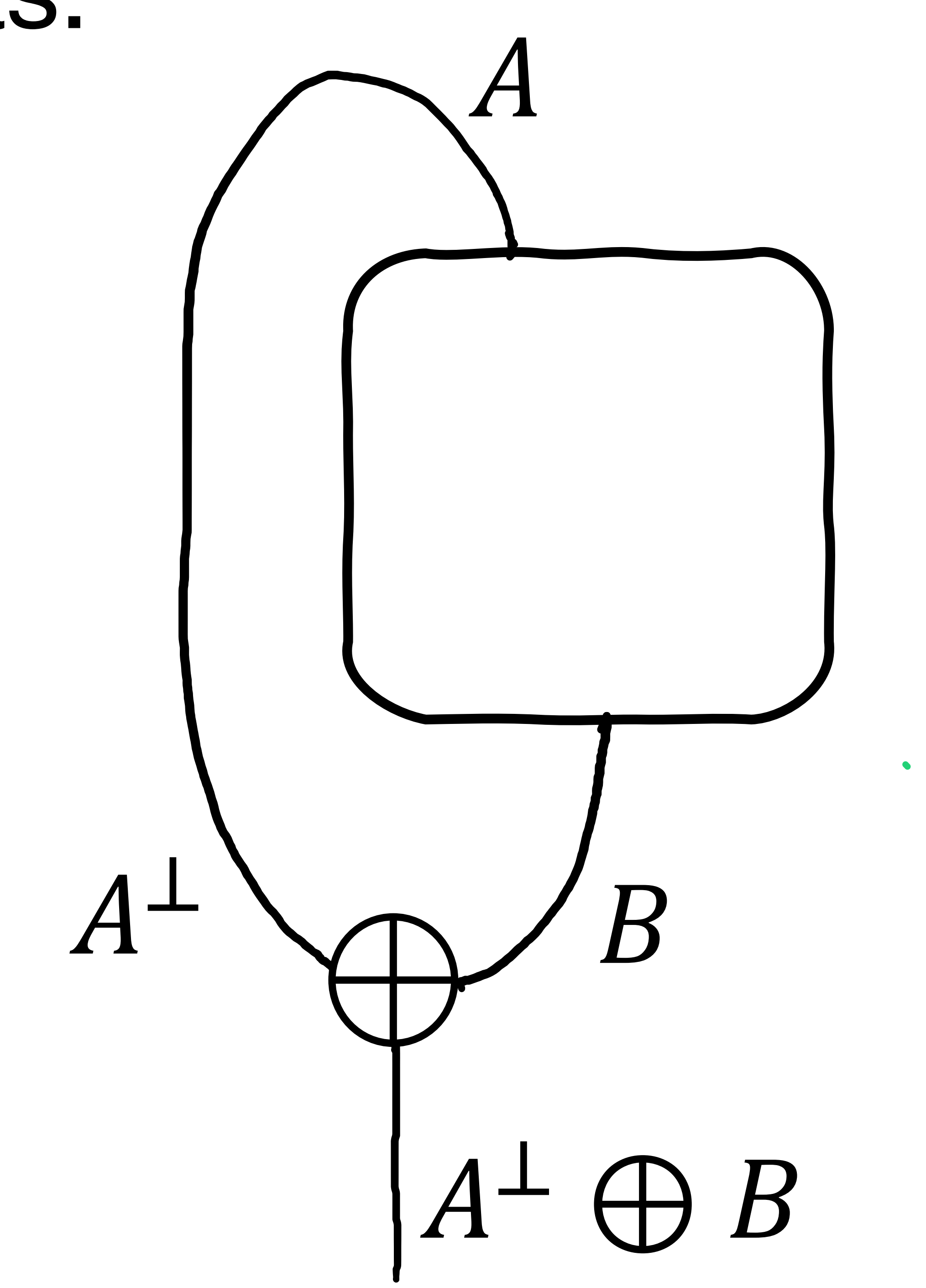
Type Classes in Concurrent CaMPL

- CaMPL is one of the first languages with a strongly typed concurrent side.
- CaMPL's rich concurrent type system is a necessary basis for investigating concurrent type classes.
- As far as we know, this is the first time that concurrent type classes have been considered for a concurrent language.

Type Classes in Concurrent CaMPL

- The concurrent side of CaMPL is already higher order as:

$$\frac{\Gamma \otimes A \vdash B}{\Gamma \vdash A \multimap B := A^{\perp} \oplus B}$$



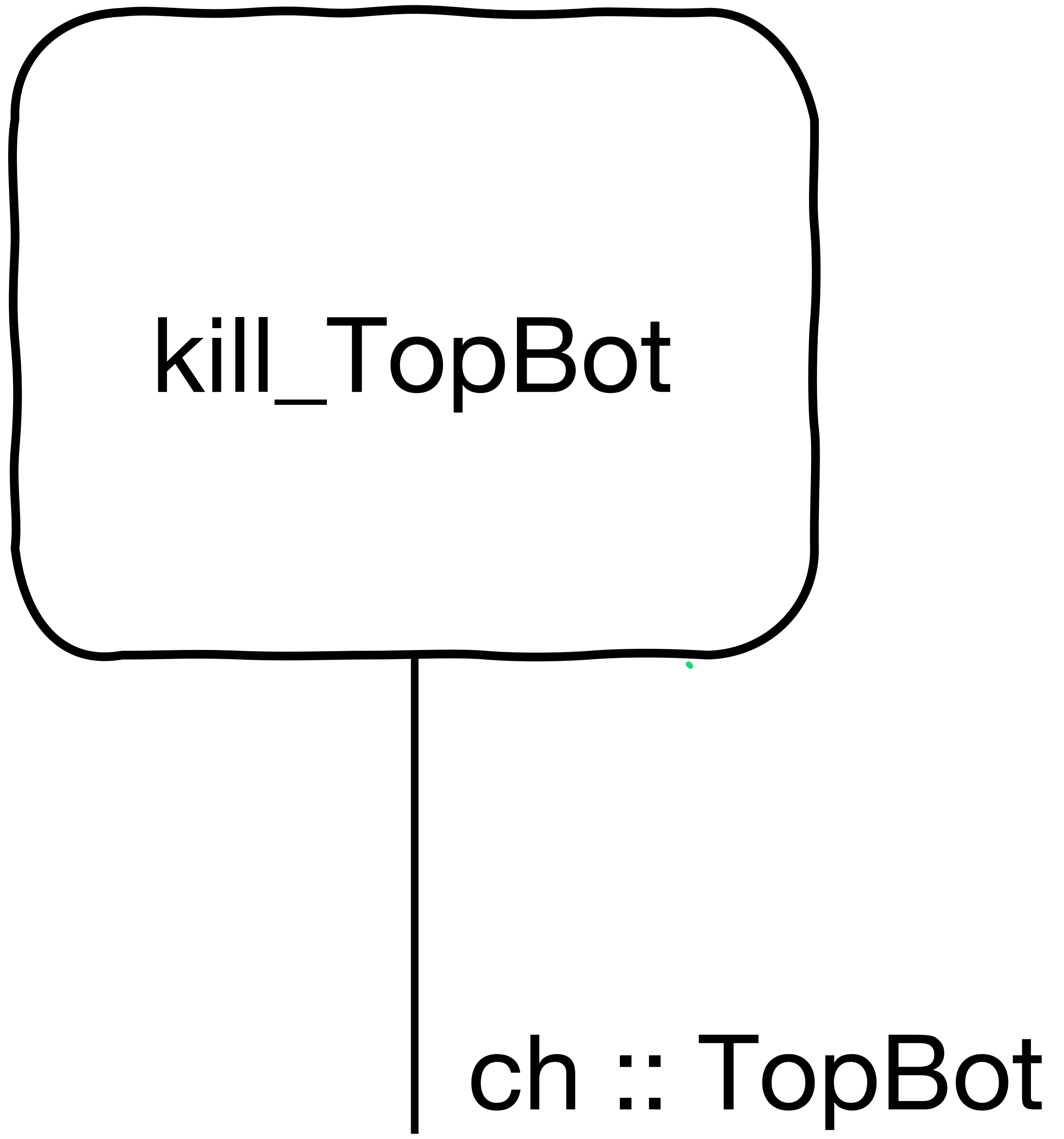
- We can pass a process with input type A and output type B, to the other process using the type Neg(A) (+) B.

Example: The Kill Type Class

CaMPL

```
class Kill T where
  proc kill :: | ⇒ T

instance Kill TopBot where
  proc kill :: | ⇒ TopBot =
    | ⇒ ch → halt ch
```

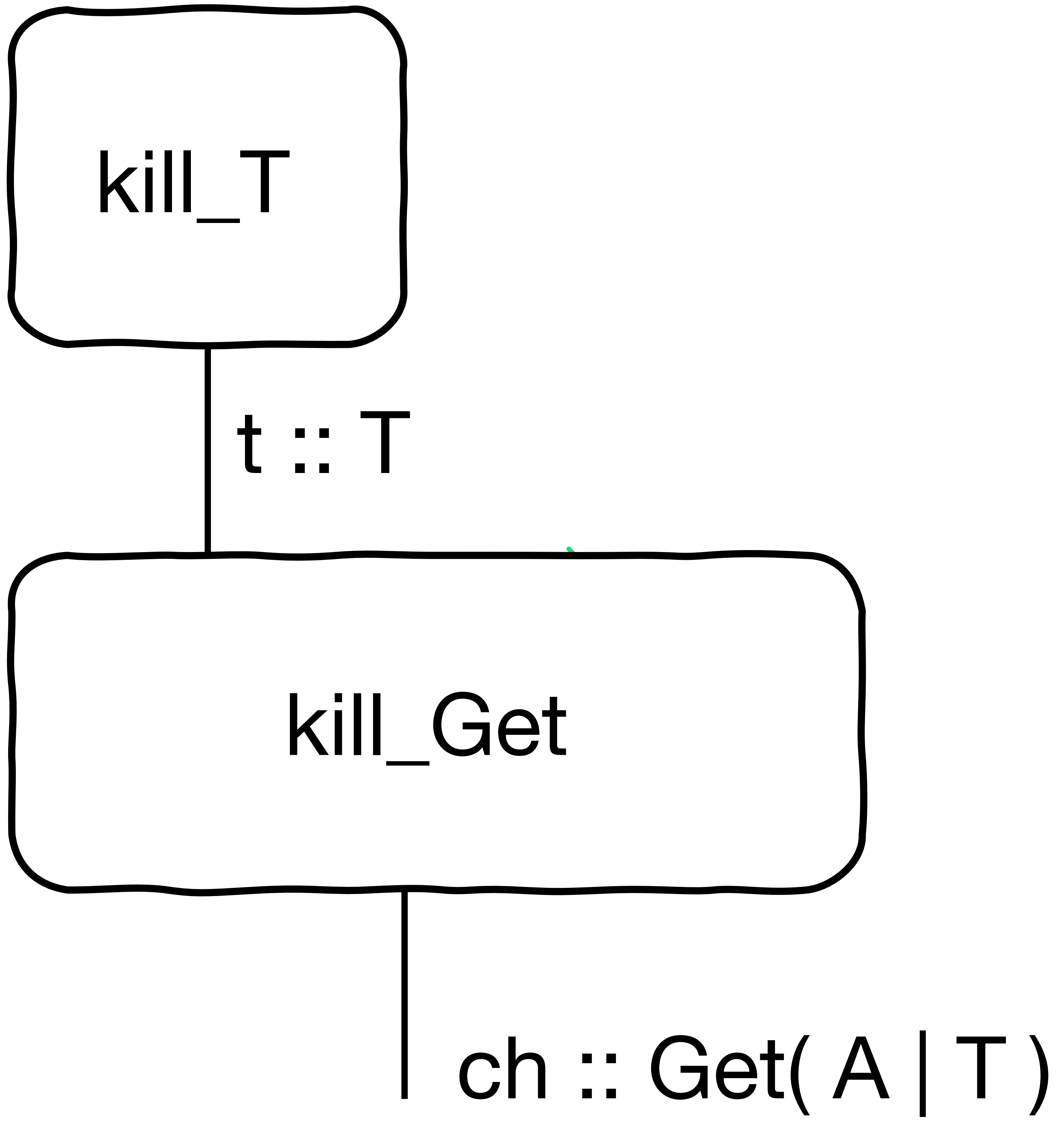


Example: The Kill Type Class

CaMPL

```
instance Kill T ⇒ Kill Get(A | T) where
  proc kill :: | ⇒ Get(A | T) =
        | ⇒ ch → do
          get a on ch
          kill( | ⇒ ch)


-- translation
proc kill_Get :: | T ⇒ Get(A | T) =
    | t ⇒ ch → do
        get a on ch
        t ⟛ ch
```

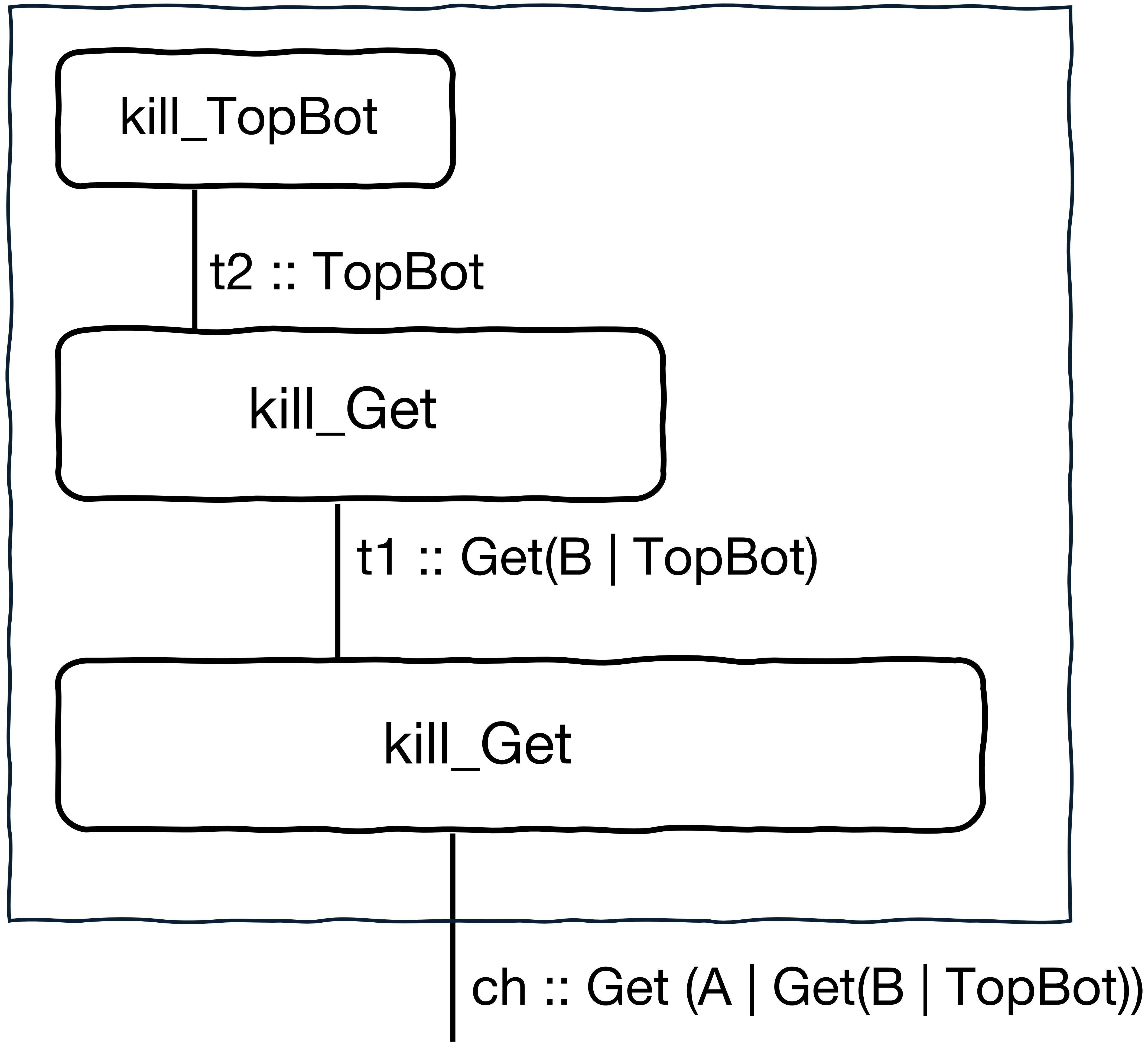


Example: The Kill Type Class

CaMPL

```
proc p :: | => Get(A | Get (B | TopBot)) =
    | => ch -> kill(ch)


-- translation:
proc p :: | => Get(A | Get (B | TopBot)) =
      | => ch -> do
        plug
          kill_Get( | t1 => ch)
          kill_Get( | t2 => t1)
          kill_TopBot( | => t2)
```



Type Classes in Concurrent CaMPL

- It is also probably useful to have higher order type classes such as Functor and Monad in the concurrent side of CaMPL.
- For example: One can define a concurrent list (list of channels) and make it a Functor.
- In our first try, we attempt to implement the Functor type class using the same way we did for first order type classes (like the Kill type class)
- Let's see if it works!

Example: Functor Type Class

CaMPL

```
class Functor \S → T( | S) where
| proc fmap :: | B (+) Neg(A), T( | A) ⇒ T( | B)

protocol List(| A) ⇒ S =
| | Cons :: A (*) S ⇒ S
| | Nil :: A ⇒ S

proc cons :: | T, List( | T) ⇒ List( | T) =
| | | t, ts ⇒ ts' → do
| | | | hput Cons on ts'
| | | | fork ts' as
| | | | | | t'' → t |=| t''
| | | | | | ts'' → ts |=| ts''
```

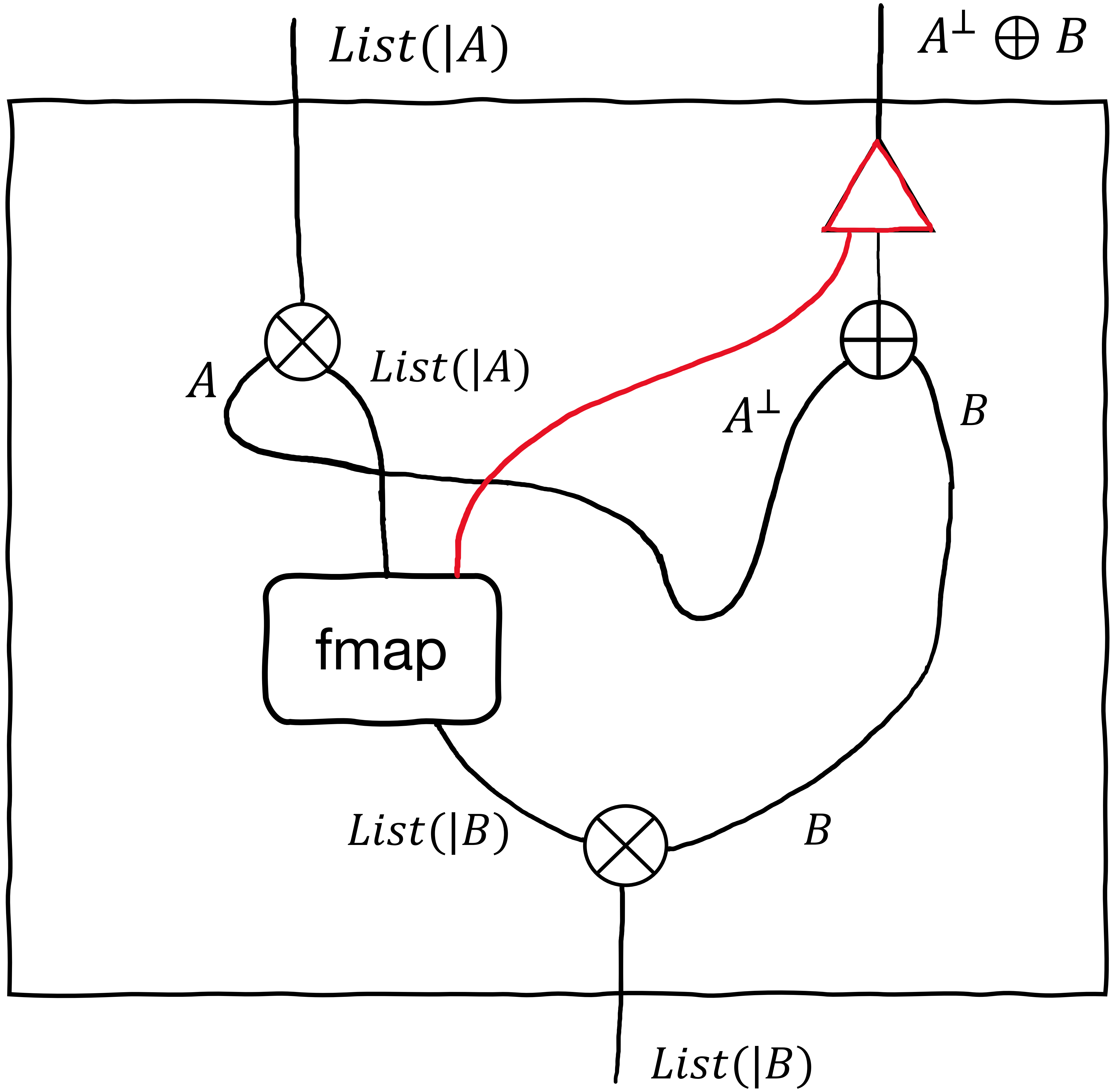
```
proc nil :: | T ⇒ List(| T) =
| | | t ⇒ ts → do
| | | | hput Nil on ts
| | | | ts |=| t

proc apply :: | Neg(A) (+) B, A ⇒ B =
| | | negAandB, a ⇒ b → do
| | | | fork negAandB as
| | | | | | nega → nega |=| neg a
| | | | | | b' → b' |=| b
```

Example: Functor Type Class

CaMPL

```
instance Functor \S → List(S) where
proc fmap :: | Neg(A) (+) B, List( | A) ⇒ List( | B) =
    | negAandB, la ⇒ lb →
        hcase la of
            Cons → do
                split la into hla, tla
                plug
                    apply(| negAandB, hla ⇒ hlb)
                    cons(| hlb, tlb ⇒ lb)
                    fmap(| negAandB, tla ⇒ tlb)
            Nil →
                plug
                    apply( | negAandB, la ⇒ b)
                    nil( | b ⇒ lb)
```



Are there any solutions?

Hmm… There might be!

Store and Use

- *Store* is a **sequential** data type that takes in a **concurrent** process type.
- One can store a process and make it sequential data, then it can behave like other sequential resources so it can be duplicated!
- One can call a stored process using the *use* command.

CaMPL

```
proc p =
    a | ⇒ b → do
        ...

proc q =
    a | b ⇒ → do
        -- store process p and
        -- pass it as sequential data to z
        z (Store(p) | b ⇒ )

proc z :: Store(A | ⇒ B ), A | b ⇒ =
    p, a | b ⇒ →
        plug
            -- use process p
            use(p)( a | ⇒ b)
            -- call z recursively,
            -- pass the stored process p to it
            z(p, a | b ⇒ )
```

Store and Use

- *Store* and *use* can help us solve the problem we had in writing fmap for concurrent lists.

- We can pass the **stored** process that we want to call on each channel in the list, to the fmap process and **use** it as needed.

- But what is the semantics of the *store* and *use*? We don't know! Although it is reminiscent of the bang of linear logic.

CaMPL

```
instance Functor \S → List(S) where
  proc fmap :: Store( | A ⇒ B) | List( | A) ⇒ List( | B) =
    p | la ⇒ lb →
      hcase la of
        Cons → do
          split la into hla, tla
          plug
            use(p)( | hla ⇒ hlb)
            cons(| hlb, tlb ⇒ lb)
            fmap(p | tla ⇒ tlb)
        Nil →
          plug
            use(p)( | la ⇒ b)
            nil( | b ⇒ lb)
```

Conclusion

- Type Classes are important and useful.
- CaMPL is a strongly typed concurrent language and it has the basis for adding type classes to its sequential and concurrent side.
- We are working on adding type classes to CaMPL: The sequential side seems to be going well but there are some challenges in concurrent side.
- Duplicating concurrent resources is not allowed, but one can store them in a sequential data and duplicate them and use them.
- We are working on providing this facility in CaMPL that enables us to implement type classes for both sequential and concurrent sides.

References

1. J. R. B. Cockett and Craig Pastro. The Logic of Message Passing. Science of Computer Programming, 74(8):498–533, 2009.
2. Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type classes in Haskell. ACM Trans. Program. Lang. Syst. 18, 2 (March 1996), 109–138. https://doi.org/10.1145/227699.227700
3. Reginald Lybbert. Progress for the Message Passing Logic. Undergraduate thesis, University of Calgary, April 2018. Provided by the author.
4. Prashant Kumar. *Implementation of Message Passing Language.* 2018. doi:10.11575/PRISM/10182. url: https://prism.ucalgary.ca/ handle/1880/106402.
5. Jared Pon. Implementation Status of CMPL. Undergraduate thesis Interim Report, University of Calgary, December 2021. Provided by the author.