Scope

Every system needs to have local variables /channels that only its members can interact with. At the same time, the channel can be passed to an external system, thus the scope of the channel changes, it encompasses this external system as well. In π-calculus, this is expressed by the structural rule:

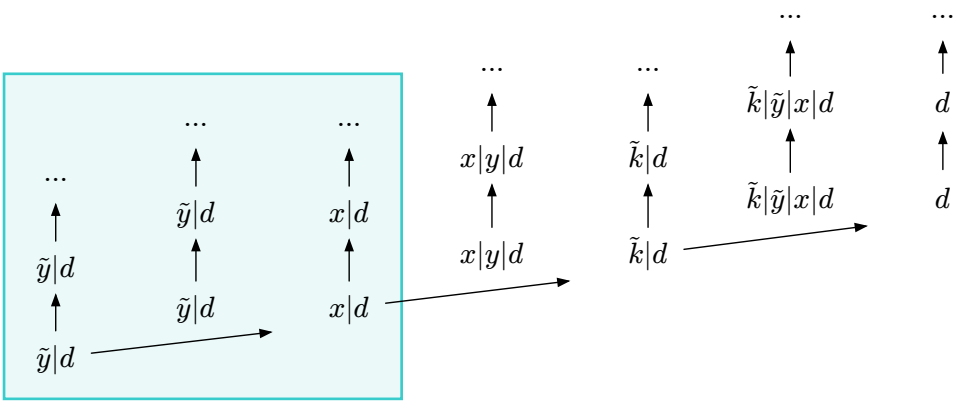
$(\nu x)(P \mid Q) \equiv (\nu x)P \mid Q$ when x is not a free variable in Q.

Let us now have an example where we show how we handle scope in our system. Consider these systems:

$A \coloneqq (\nu x)\big(\tilde{y}\langle x\rangle.0 | x(k).\tilde{k}\langle q\rangle.0 | d(q).A\big)$

$B \coloneqq y(e).\tilde{e}\langle z\rangle.\tilde{d}\langle k\rangle.0 | z(q).B$

We have discussed that our systems are described by sets of potentialities. Here though, we have the simplest systems. Let us describe system A:



Each horizontal line represents the function of change after system A receives/sends a msg from a specific channel. For the first case, there are two possibilities, either it sent a message on channel $\tilde{y}$ or it received a message on channel d. For this reason, there are two potentialities.

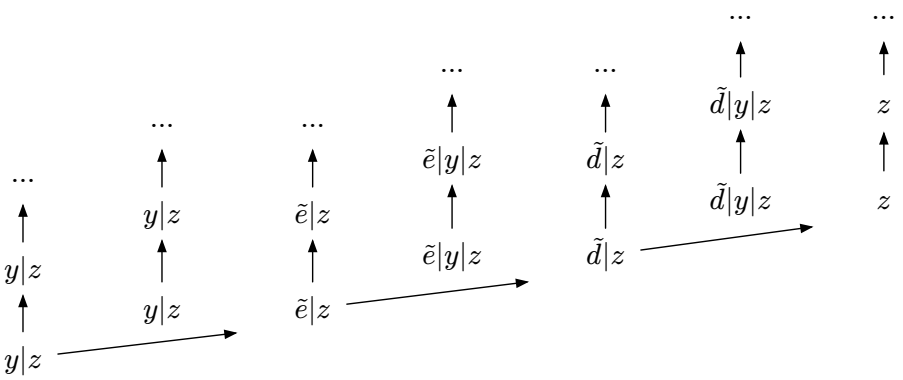
k is a variable, it is the secret to be received by channel x.

Keep in mind that the type does not track the number of **actors** that are present. It only cares whether there is at least one actor that accepts/sends a specific channel. It is idempotent. It is for this reason that the second column is not $\tilde{y}|\tilde{y}|d$

Also, the diagram is incomplete, since we do not describe the functions of change of the other potentialities.

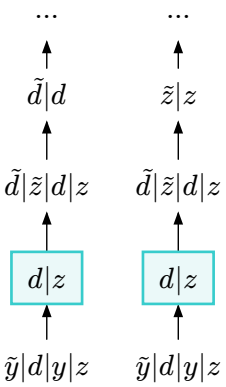
Now, if you look closer, at the first type, it is $\tilde{y}|d$ when channel x is ready to receive new messages. Of course, this is not possible since x is a local variable. In column 3 though, x is part of the type, the reason for that is that channel x has been transmitted to the outside world.

System B:



Both systems A and B are static, meaning that they cannot progress any further without external help. For this reason each column has a constant type. In general though, systems progress on their own.

In the next diagram, I will only describe the initial potentialities of the system A&B, when it is not pertrubed by an external system.



The interesting thing happens at the second "state" of the potentialities. Here, it should have been $x|d|\tilde{x}|z$. The reason it is not is because channel x has been sent inside the system, thus we know that noone else has channel x, thus it is impossible that the external world communicate with A&B through channel x.

This is how I handle scope at the moment, by limiting the type of the system to remove channels that cannot be accessed by the exterior world.

In this 'library', we do not have channels, by predicates that require the knowledge of a list of secrets. Thus, any predicates that can only be fulfilled from inside the system are removed. This is the functionality of the **limit&** function.

```
restr : ∀{𝒰 𝒱} → {A : 𝒰 ̇} → (P : A → 𝒱 ̇) → Σ P → A
restr P x =  x .pr₁

_$₂_ : ∀{𝒰 𝒱} → {A : 𝒰 ̇} → {B : 𝒱 ̇} → (A → B) → A × A → B × B
f $₂ (a , b) = f a , f b

+→𝟚 : ∀{𝒰 𝒱} → {X : 𝒰 ̇} → {Y : 𝒱 ̇} → X + Y → 𝟚
+→𝟚 (inl x) = ₀
+→𝟚 (inr x) = ₁

scope-l1 : (x : Secret) → (ls : List Secret) → (A : 𝟚 → 𝓦 ̇)
         → is-decidable (x ∈ ls) → 𝓦 ̇
scope-l1 x ls A r = A (+→𝟚 r)
```

```
module BSet-scope (_∈?_ : ∀ s ls → is-decidable (s ∈ ls)) where

 Lim : 𝒱 ̇ → 𝟚 → Set 𝒱
 Lim P ₀ = 𝟘
 Lim P ₁ = P

 limitPr : Secret → 𝒱 ̇ → Pred S×Msg 𝒱
 limitPr s P mp@(ls , msg) = scope-l1 s ls (Lim P) (s ∈? ls)

 limit : Secret → BSet 𝒱 → BSet 𝒱
 limit s bs .pr₁ mp = limitPr s (< bs > mp) mp
 limit s bs .pr₂ = λ ascrs scrs x (a⊂s , a⊃s) → l1 ascrs scrs x a⊂s a⊃s (s ∈? ascrs)
(s ∈? scrs) , l2 ascrs scrs x a⊂s a⊃s (s ∈? scrs) (s ∈? ascrs) where
   l1 : ∀ ascrs scrs x a⊃s a⊂s → (deq : is-decidable (s ∈ ascrs)) → (deq2 : is-
decidable (s ∈  scrs)) → scope-l1 s ascrs (Lim (< bs > (ascrs , x))) deq → scope-l1 s
scrs (Lim (< bs > (scrs , x))) deq2
   l1 ascrs scrs x a⊃s a⊂s (inr neq) (inl eq2) cond = 𝟘-elim (neq (∈→∈ s scrs ascrs
a⊂s eq2))
   l1 ascrs scrs x a⊃s a⊂s (inr neq) (inr x₁) cond = bs .pr₂ ascrs scrs x (a⊃s ,
a⊂s) .pr₁ cond

   l2 : ∀ ascrs scrs x a⊃s a⊂s → (deq : is-decidable (s ∈ scrs)) → (deq2 : is-
decidable (s ∈ ascrs)) → scope-l1 s scrs (Lim (< bs > (scrs , x))) deq → scope-l1 s
ascrs (Lim (< bs > (ascrs , x))) deq2
   l2 ascrs scrs x a⊃s a⊂s (inr neq) (inl eq2) cond = 𝟘-elim (neq (∈→∈ s ascrs scrs
a⊃s eq2))
   l2 ascrs scrs x a⊃s a⊂s (inr neq) (inr x₁) cond = bs .pr₂ ascrs scrs x (a⊃s ,
a⊂s) .pr₂ cond

 limitMPr : Secret → List Secret → 𝒱 ̇ → Pred S×Msg 𝒱
 limitMPr s [] bs mp = limitPr s bs mp
 limitMPr s (l ∷ ls) w mp = let w2 = limitPr s w mp
                                w3 = limitMPr l ls w2 mp
                            in w3

 limitPr-𝟘 : ∀ s mp → limitPr {𝒱} s 𝟘 mp = 𝟘
 limitPr-𝟘 s  mp@(scr , _) with (s ∈? scr)
 ... | inl x = refl
 ... | inr x = refl

 limitMPr-𝟘 : ∀ s ls mp → limitMPr {𝒱} s ls 𝟘 mp = 𝟘
 limitMPr-𝟘 s [] mp@(scr , _) = limitPr-𝟘 s mp
 limitMPr-𝟘 s (l ∷ ls) mp = ap (λ z → limitMPr l ls z mp) (limitPr-𝟘 s mp) •
limitMPr-𝟘 l ls mp

 limitM : Secret → List Secret → BSet 𝒱 → BSet 𝒱
 limitM s ls bs .pr₁ mp = limitMPr s ls (< bs > mp) mp
 limitM s [] bs .pr₂ = limit s bs .pr₂
 limitM s (l ∷ ls) bs .pr₂ = limitM l ls (limit s bs) .pr₂

 limitM' : List Secret → BSet 𝒱 → BSet 𝒱
 limitM' [] bs = bs
 limitM' (s ∷ ls) bs = limitM s ls bs
```

```
-- limitM is a restriction, so it fits where bs fits.
 lim-rec : ∀{𝓦} → {A : 𝓦 ̇} → ∀ s ls {bs mp} → < (limitM {𝓥} s ls bs) > mp → (< bs >
mp → A) → A
 lim-rec s [] {bs} {mp@(ws , msg)} c f = l1 (s ∈? ws) c where
  l1 : (w : (s ∈ ws) + (s ∈ ws → 𝟘)) →
       Lim (< bs > (ws , msg)) (+→𝟚 w) → _
  l1 (inr _) c = f c

 lim-rec {𝓥 = 𝓥} s (l :: ls) {bs} {mp@(ws , msg)} c f = l1 (s ∈? ws) c where
  l1 : (w : (s ∈ ws) + (s ∈ ws → 𝟘)) →
       limitMPr l ls (Lim (< bs > (ws , msg)) (+→𝟚 w)) (ws , msg) → _
  l1 (inl x) c with limitMPr {𝓥} l ls 𝟘 mp | (limitMPr-𝟘 {𝓥} l ls mp)
  l1 (inl x) () | r | refl
  l1 (inr x) c = lim-rec l ls {bs} {mp} c f


 lim-rec' : ∀{𝓦} → {A : 𝓦 ̇} → ∀ ls bs {mp} → < (limitM' {𝓥} ls bs) > mp → (< bs > mp
→ A) → A
 lim-rec' [] _ c f = f c
 lim-rec' (x :: ls) bs {mp} = lim-rec x ls {bs}


 module &PSet-scope {𝓥} where

  limit&P : Secret → &PSet 𝓥 𝓦 → &PSet 𝓥 (𝓤 ⊔ 𝓥 ⁺ ⊔ 𝓦)
  limit&P s ps .pr₁ v = v ∈image λ x → (λ (a , bs) → a , limit s bs) (restr < ps > x)
  limit&P s ps .pr₂ = cons-is-non-empty

  limit&PM : Secret → List Secret → &PSet 𝓥 𝓦 → &PSet 𝓥 (𝓤 ⊔ 𝓥 ⁺ ⊔ 𝓦)
  limit&PM s ls ps .pr₁ v = v ∈image λ x → (λ (a , bs) → a , limitM s ls bs) (restr <
ps > x)
  limit&PM s ls ps .pr₂ = cons-is-non-empty
```