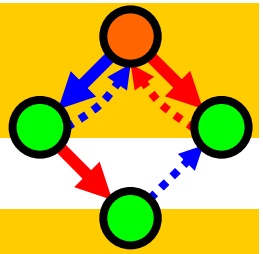
The purpose of models is not to fit the data but to sharpen the questions. Samuel Karlin.

Artificial Biochemistry



Complexation

Luca Cardelli

Microsoft Research

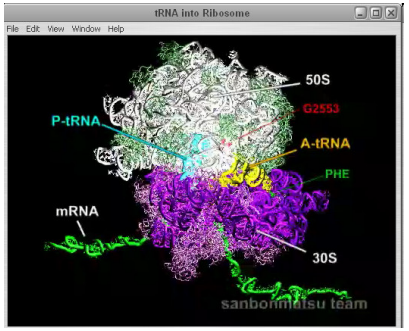
The Microsoft Research - University of Trento
Centre for Computational and Systems Biology

Trento, 2006-05-22..26

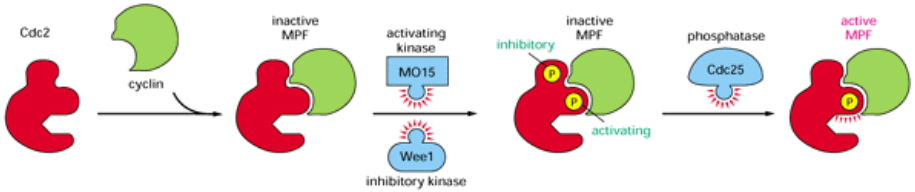
www.luca.demon.co.uk/ArtificialBiochemistry.htm

Complexation in Biochemistry

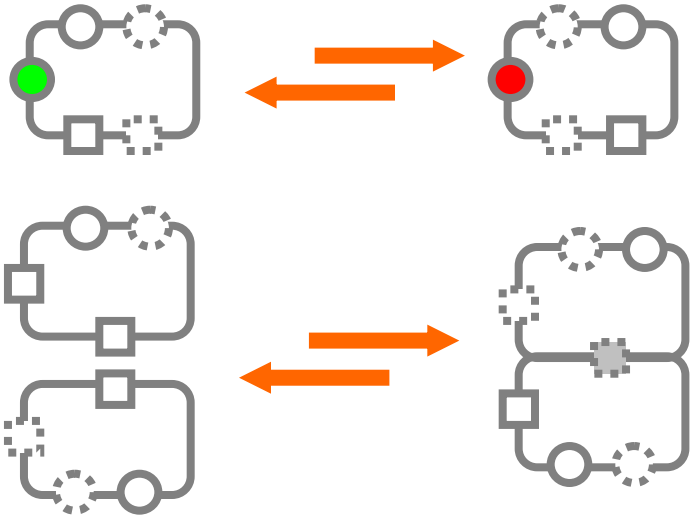
- Complexation
  - Proteins form complexes
  - Enzymes work by complexation
  - Biological machines are often made of complexes of dozens of proteins



- Abstraction
  - Complexation is a fundamental modeling abstraction



- Processes
  - We can easily handle phosphorylation (state) and solutions (composition)
  - But there is no complexation in process algebra
  - How are we going to make "processes stick together" (so they each have their *local state*)

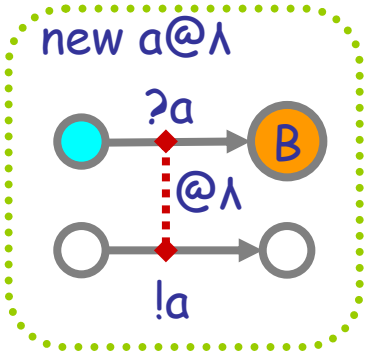


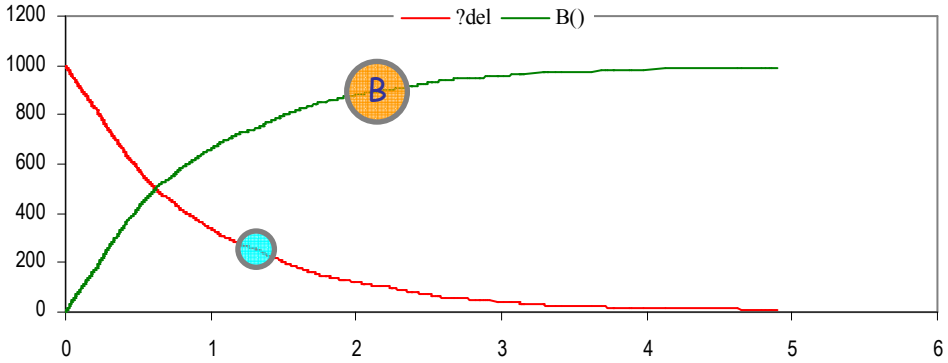
Encapsulating Interaction

Decay = Private Interaction

$@\lambda;B = \text{new } a@\lambda\ (!a \mid ?a;B)$ a not occurring in B

$=_{def}$





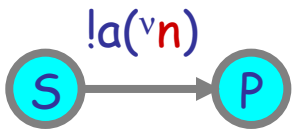

```
directive sample 5.0 10000
directive plot ?del; B()
new del@1.0:chan

let Delay(r:float, P:proc()) =
  (new a@r:chan
   run (!a | do ?a; P() or ?del))

let A() = Delay(1.0, B)
and B() = ()

run 1000 of A()
```

Private interaction, in mass, obeys the same exponential decay law as degradation.

(Because each private interaction is a single event sampled from an exponential distribution.)

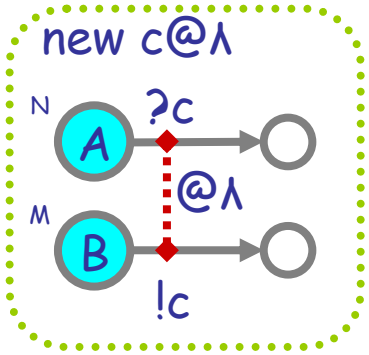
Delay(r,P) = delay@r;P

Graphical Notation: *bound output*



The ν in front of n indicates that this is a *new n* that is being sent as output. That n is a *binding occurrence* (since the *new* is a binder) and may be colored red as such.

Shared Private Interaction



We want two processes A(-) and B(-) that, no matter how many times invoked, talk on a single shared *private* channel between themselves.

A and B are closures that share c in their scope.

A and B are "returned"

A and B are invoked 100 times

```
directive sample 1.0 1000
directive plot ?c

let Share(Continue:proc(proc(),proc())) =
  (new c@1.0:chan
   let A() = ?c
   and B() = !c
   run Continue(A,B))

let Continue(A:proc(),B:proc()) =
  (run 100 of (A() | B()))

run Share(Continue)
```

"Client code"

Or, how to make functional closures in SPiM.

Now we abstract out the bodies of the procedures A(-),B(-) to pull them out of the Question boilerplate code.

"Library" code for channel sharing

Ac and Bc are closures that have c as parameter.

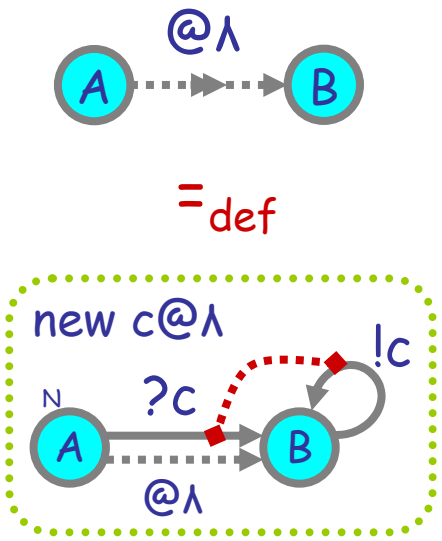
Ac and Bc are invoked here

Ac and Bc are "passed" to the library

```
directive sample 1.0 1000
directive plot ?c

let Share(Ac:proc(chan),Bc:proc(chan),
          Continue:proc(proc(),proc())) =
  (new c@1.0:chan
   let A() = Ac(c)
   and B() = Bc(c)
   run Continue(A,B) )

let Ac(c:chan) = ?c
and Bc(c:chan) = !c

let Continue(A:proc(),B:proc()) =
  (run 100 of (A() | B()))

run Share(Ac,Bc,Continue)
```

"Client code"

Fast Decay as Shared Private Interaction

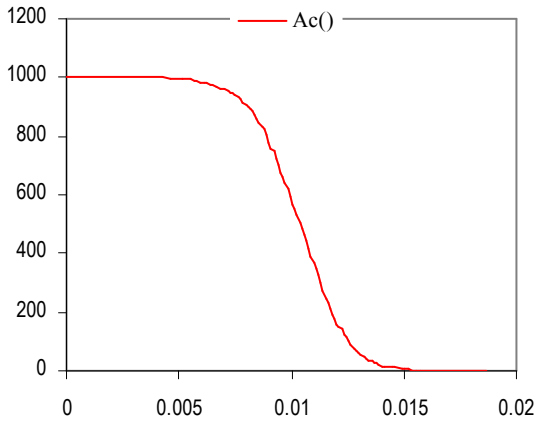


Here we want to define a fast decay A->B process, and only *later* decide how many copies of A there should be; note that all those copies must share the same private channel.

```
directive sample 0.1 1000
directive plot Ac()

let Share(Ac:proc(chan), Continue:proc(proc())) =
  (new c@1.0:chan
   let P() = Ac(c)
   run Continue(P) )

let Ac(c:chan) = do delay@1.0;Bc(c) or ?c;Bc(c)
and Bc(c:chan) = !c;Bc(c)

let Continue(A:proc()) =
  (run 1000 of A())

run Share(Ac,Continue)
```

All N fast decay processes must share the same private c!

Because the Bs *collectively* help drive the fast decay.



Complexation Modeling Techniques

Complexation

$A + B \rightarrow A{:}B$ (binary reaction)

$A{:}B \rightarrow A + B$ (unary reaction)

new a@μ red=binders

$A_{free} = !a(\nu n_\lambda);\ A_{bound}(n)$

$A_{bound}(n) = !n;\ A_{free}$

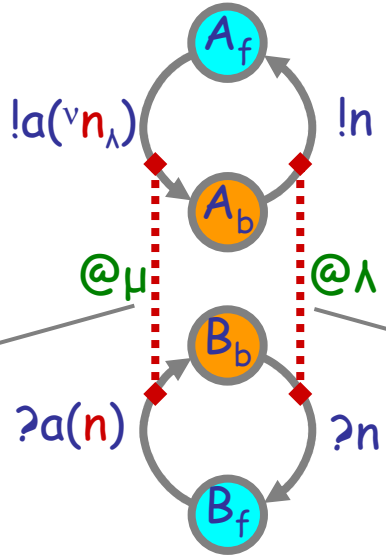
$B_{free} = ?a(n);\ B_{bound}(n)$

$B_{bound}(n) = ?n;\ B_{free}$

Complexation is modeled by a shared private channel.

```
directive sample 1.0 1000
directive plot Afree(); Abound(); Bfree(); Bbound()

val mu = 1.0    val lam = 1000.0
new a@mu:chan(chan)

let Afree() = (new n@lam:chan run !a(n); Abound(n))
and Abound(n:chan) = !n; Afree()

let Bfree() = ?a(n); Bbound(n)
and Bbound(n:chan) = ?n; Bfree()

run (150 of Afree() | 200 of Bfree())
```

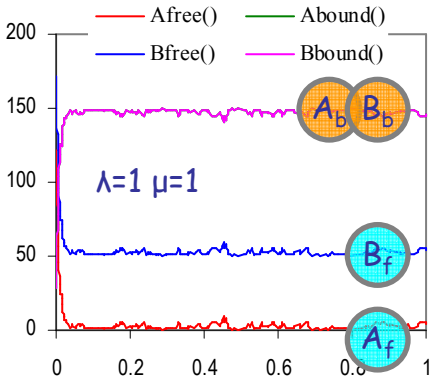
Bound output:
$!a(\nu n_\lambda);P = \text{new } n@\lambda\ (!a(n);P)$

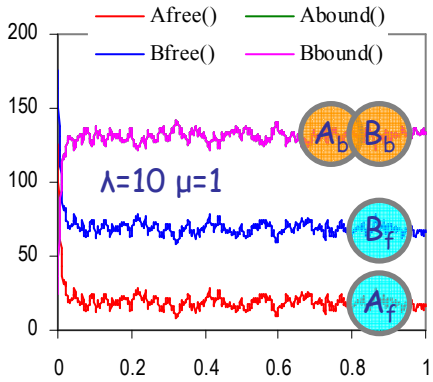


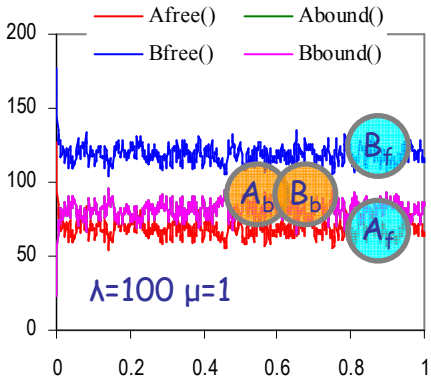
complexation (mass interactions on global a)

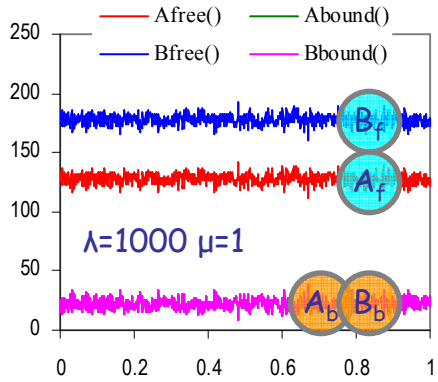
decomplexation (private interactions on each separate n)

150xA 200xB









Complexation: π-reductions

new $a@\mu$ red=binders

$A_{free} = !a(\nu n_{\lambda}); A_{bound}(n)$
$A_{bound}(n) = !n; A_{free}$

$B_{free} = ?a(n); B_{bound}(n)$
$B_{bound}(n) = ?n; B_{free}$

[Regev & Shapiro]

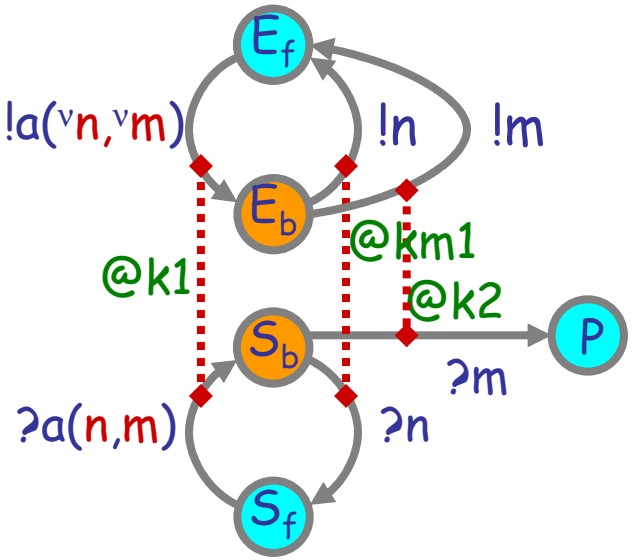
$A_{free} \mid B_{free}$ ← decomplexed state
$= !a(\nu n_{\lambda}); A_{bound}(n) \mid ?a(n); B_{bound}(n)$

$\rightarrow$ new $n@\lambda$ $(A_{bound}(n) \mid B_{bound}(n))$ ← complexed state (connected by "fresh" n)
$=$ new $n@\lambda$ $(!n; A_{free} \mid ?n; B_{free})$

$\rightarrow A_{free} \mid B_{free}$ ← decomplexed state (previous n is "forgotten")

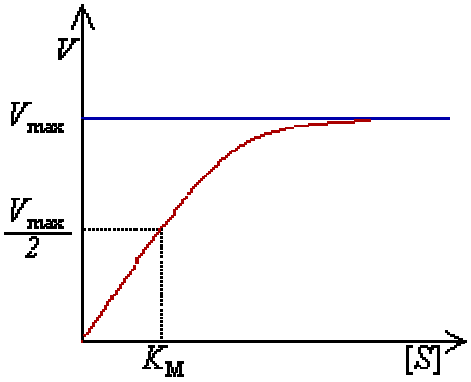
Enzymes

$$E + S \underset{k_{-1}}{\overset{k_1}{\rightleftarrows}} ES \xrightarrow{k_2} E + P$$



Michaelis-Menten *steady-state* approximation

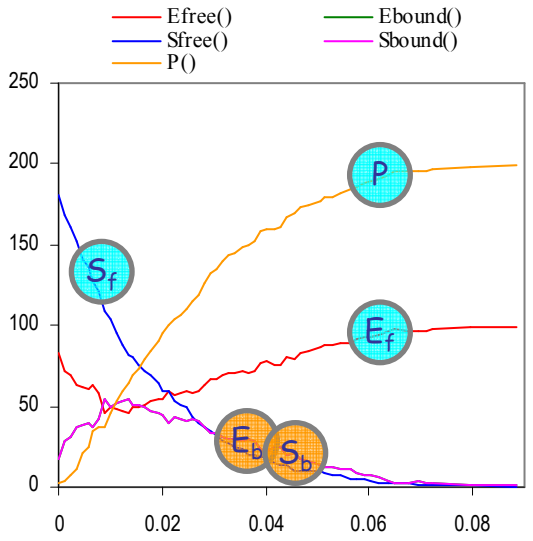
(derived by assuming [ES]• = 0)



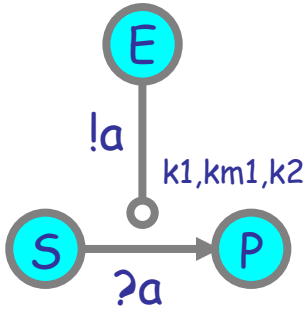
$$K_m = \frac{k_{-1} + k_2}{k_1} \qquad [E_0] = [E] + [ES]$$

$$\frac{d[P]}{dt} = k_2[E_0]\frac{[S]}{K_m + [S]} = V_{max}\frac{[S]}{K_m + [S]}$$

http://en.wikipedia.org/wiki/Enzyme

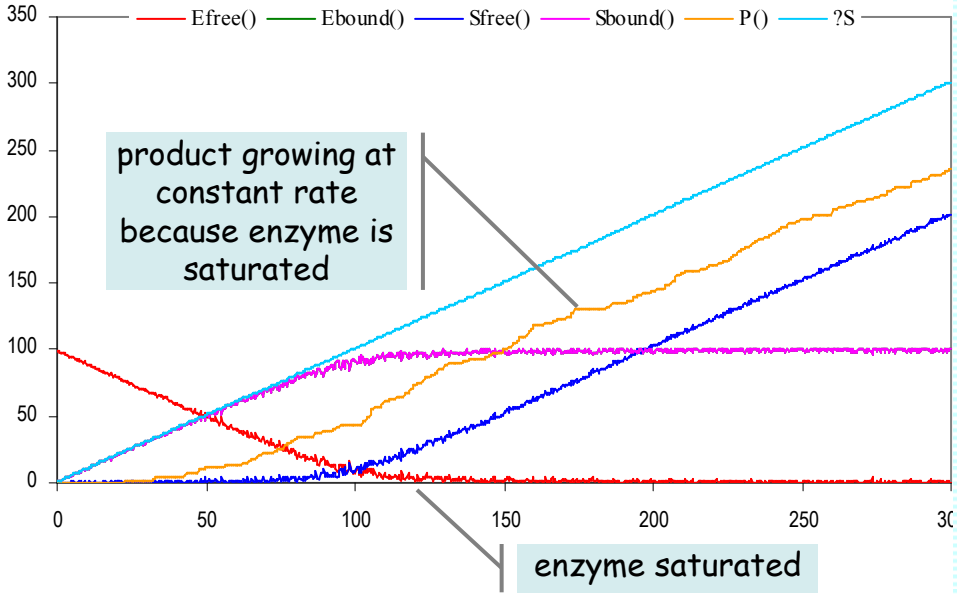


Notation




```
directive sample 1.0 1000
directive plot Efree(); Ebound(); Sfree(); Sbound(); P()

val k1 = 1.0   val km1 = 1.0   val k2 = 100.0
new a@k1:chan(chan,chan)   new stop@1.0:chan

let P() = ?stop

let Efree() =
  (new n@km1:chan new m@k2:chan
   run !a(n,m); Ebound(n,m))
and Ebound(n:chan,m:chan) =
  do !n; Efree() or !m; Efree()

let Sfree() = ?a(n,m); Sbound(n,m)
and Sbound(n:chan,m:chan) =
  do ?n; Sfree() or ?m; P()

run (100 of Efree() | 200 of Sfree())
```

Enzyme Equilibrium



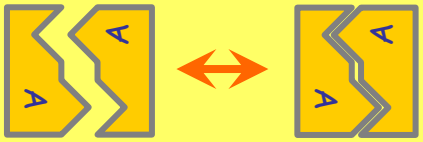
Total S is made to grow linearly.

E gets saturated at t=100..150.

After that, rate of production of P reaches a steady state.

Ebound is hidden behind Sbound in the plot because they are identical.

```
directive sample 300.0 1000
directive plot Efree(); Ebound(); Sfree(); Sbound(); P(); ?S

val k1 = 1.0   val km1 = 1.0   val k2 = 0.01
new a@k1:chan(chan,chan)   new S@1.0:chan new stop@1.0:chan

let P() = ?stop

let Efree() =
  (new n@km1:chan new m@k2:chan
   run !a(n,m); Ebound(n,m))
and Ebound(n:chan,m:chan) =
  do !n; Efree() or !m; Efree()

let Sfree() =
  do ?a(n,m); Sbound(n,m)
  or ?S; ()  (* plotting total S *)
and Sbound(n:chan,m:chan) =
  do ?n; Sfree()
  or ?m; (P() | Sfree())  (* Holding S concentration constant *)
  or ?S; ()  (* plotting total S *)

run 100 of Efree()

let clock(t:float, tick:chan) =        (* sends a tick every t time *)
  (val ti = t/100.0 val d = 1.0/ti     (* by 100-step erlang timers *)
   let step(n:int) = if n<=0 then !tick; clock(t,tick) else delay@d; step(n-1)
   run step(100))
let S(p:proc(), tick:chan) = (p() | ?tick; S(p,tick))
let raising(p:proc(), t:float) =
  (new tick:chan run (clock(t,tick) | S(p,tick)))

run raising(Sfree,1.0)
```



Homodimerization

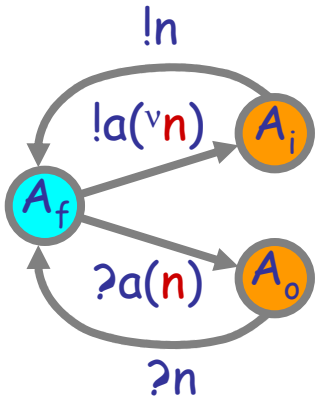
new a@μ

red=binders

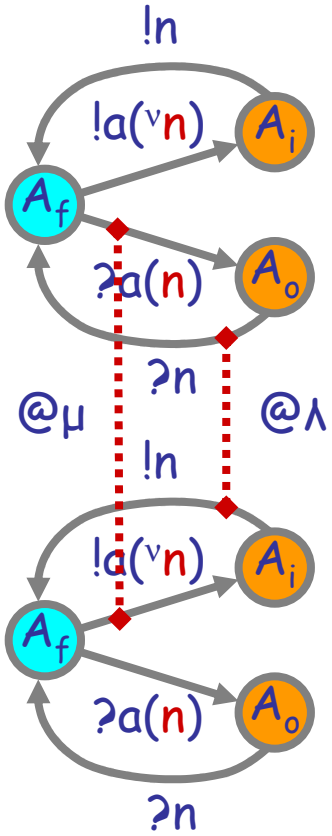
A_{free} = ?a(n); $A_{in}(n)$ + !a($^{\nu}n_{\lambda}$); $A_{out}(n)$

$A_{in}(n)$ = ?n; A_{free}

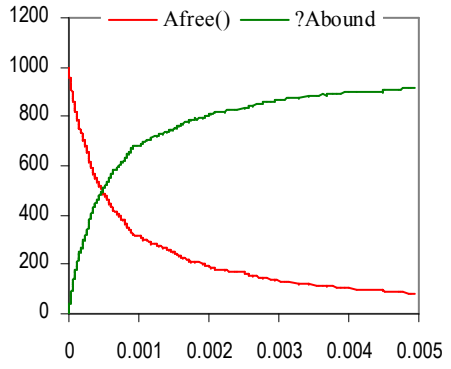
$A_{out}(n)$ = !n; A_{free}



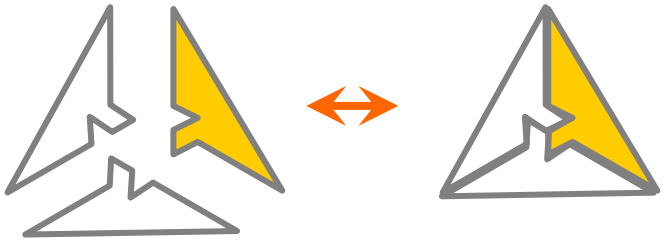
Homodimerization is symmetric complexation




```
directive sample 0.005 10000
directive plot Afree(); ?Abound
new Abound@1.0:chan

val mu = 1.0
val lam = 1.0
new a@mu:chan(chan)

let Afree() =
  (new n@lam:chan
   run do ?a(m); Ain(m)
      or !a(n); Aout(n))

and Ain(n:chan) =
  do ?n; Afree() or ?Abound

and Aout(n:chan) =
  do !n; Afree() or ?Abound

run 1000 of Afree()
```



Exercise (Open): Homotrimerization



Swap Complexation

red=binders

new b@μb new d@μd

$A_B(nb) = !d(^{\nu}nd_{\lambda d}); !nb; A_D(nd)$
$A_D(nd) = !b(^{\nu}nb_{\lambda b}); !nd; A_B(nb)$

$B_{free} = ?b(nb); B_A(nb)$
$B_A(nb) = ?nb; B_{free}$

$D_{free} = ?d(nd); D_A(nd)$
$D_A(nd) = ?nd; D_{free}$

$A_B(nb)$: A connected to B via nb
$A_D(nd)$: A connected to D via nd
$B_A(nb)$: B connected to A via nb
$D_A(nd)$: D connected to A via nd

$D_{free} \mid A_B(nb) \mid B_A(nb)$
$\xrightarrow{d} D_A(nd) \mid !nb; A_D(nd) \mid B_A(nb)$ *for new* nd
$\xrightarrow{nb} D_A(nd) \mid A_D(nd) \mid B_{free}$

Swap Complexation

new b@μb new b'@μb'
new d@μd new d'@μd'

$A_B(n)$ = ?d'; !n; !d(n); $A_D(n)$
$A_D(n)$ = ?b'; !n; !b(n); $A_B(n)$

B_{free} = !b'; ?b(n); $B_A(n)$
$B_A(n)$ = ?n; B_{free}

D_{free} = !d'; ?d(n); $D_A(n)$
$D_A(n)$ = ?n; D_{free}

new n@λn ($A_B(n)$ | $B_A(n)$ | D_{free})

the unique channel used in all the complexations of one A with any B or D

Idea: *reuse* a private channel, instead of always creating new ones.

Needs a little handshake on d',b' channels to properly serialize the use of the private channel.

(Assumes that release rates of B and D are the same, or else assumes using different weighted actions on release)

This kind of technique is important, e.g., if one wants to have any chance of generating a *finite* CTMC.

	D_{free}	| $A_B(n)$	| $B_A(n)$
$\xrightarrow{d'}$	?d(n); $D_A(n)$	| !n; !d(n); $A_D(n)$	| $B_A(n)$
$\xrightarrow{n}$	?d(n); $D_A(n)$	| !d(n); $A_D(n)$	| B_{free}
$\xrightarrow{d}$	$D_A(n)$	| $A_D(n)$	| B_{free}

Recombination

Idea: *reuse* the private channels!

red=binders

```
type P = chan(Q)
and Q = chan(P)

new pp@λpp:Q new qq@λqq:P

A_B(p:P)   = !pp(p); ?p(q); A_D(q)
A_D(q:Q)   = !qq(q); ?q(p); A_B(p)

C_D(q:Q)   = ?pp(p); !p(q); C_B(p)
C_B(p:P)   = ?qq(q); !q(p); C_D(q)

B(p:P)     = ...
D(q:Q)     = ...

new p:P@λp new q:Q@λq
  (A_B(p) | B(p) | C_D(q) | D(q))
```

A_B is connected to B by a private p:P
A_D is connected to D by a private q:Q
C_D is connected to D by a private q:Q
C_B is connected to B by a private p:P

pp:chan(P) is a global channel used by A_B to find a C_D to swap private channels with; A_B begins by offering its p on pp, then receives its q on p.

qq:chan(Q) is a global channel used by A_D to find a C_B to swap private channels with; A_D begins by offering its q on qq, then receives its p on q.

the unique two channels reused on each recombination

$$
\begin{array}{lllll}
 & A_B(p) & \mid C_D(q) & \mid B(p) \mid D(q) & \\
\xrightarrow{pp} & ?p(q);\ A_D(q) & \mid\ !p(q);\ C_B(p) & \mid B(p) \mid D(q) & \text{(A gives p to C over pp)} \\
\xrightarrow{p} & A_D(q) & \mid C_B(p) & \mid B(p) \mid D(q) & \text{(C gives q to A over p)}
\end{array}
$$

Swap Interaction and Molecule Identities

$!?c(n,x).P \mid ?!c(y,m).Q \rightarrow P\{x<-m\} \mid Q\{y<-n\}$ red=binders

$!?c(n,x).P = \text{new } p\ (!c(n,p); ?p(x); P$ (p not in P)

$?!c(y,m).Q = ?c(y,p); !p(m); Q$ (p not in Q)

types: n:N, m:M, p:chan(M), c:chan(N,chan(M))

First, define the notion of *swap interaction*.

$A_{id}(a) = !?ab(a,b); \ldots$

$A() = \text{new } a@\lambda\ A_{id}(a)$ generating the identity

Here is a different programming style, which scales up better to complex interactions.

Each process is parameterized by its own molecule identity (its first parameter). The first thing that happens in an interaction is then typically a swap of identities over some public channel, by the above swap interaction.

After that, the identities are used as private channels for communication between the molecules; here is complexation/decomplexation rewritten in this style. (In this case, a is not actually used.)

new ab@μ

$A_{free}(a) = !?ab(a,b); A_{bound}(a,b))$

$A_{bound}(a,b) = !b; A_{free}(a)$

$B_{free}(b) = ?!ab(a,b); B_{bound}(b,a)$

$B_{bound}(b,a) = ?b; B_{free}(b)$

$A() = \text{new } a@1\ A_{free}(a)$

$B() = \text{new } b@\lambda\ B_{free}(b)$

$$\begin{array}{ll} & A_{free}(a) \mid B_{free}(b) \\ \xrightarrow{ab} & A_{bound}(a,b) \mid B_{bound}(b,a) \\ \xrightarrow{b} & A_{free}(a) \mid B_{free}(b) \end{array}$$

Recombination

red=binders

new cd new cb

$A_B(a,b)$ = ?!cd((c,d),(a,b)); !b(c); $A_D(a,d)$
$A_D(a,d)$ = ?!cb((c,b),(a,d)); !d(c); $A_B(a,b)$

$C_D(c,d)$ = !?cd((c,d),(a,b)); !d(a); $C_B(c,b)$
$C_B(c,b)$ = !?cb((c,b),(a,d)); !b(a); $C_D(c,d)$

$B_A(b,a)$ = ?b(c); $B_C(b,c)$
$B_C(b,c)$ = ?b(a); $B_A(b,a)$

$D_C(d,c)$ = ?d(a); $D_A(d,a)$
$D_A(d,a)$ = ?d(c); $D_C(d,c)$

AB() = new a,b ($A_B(a,b)$ | $B_A(b,a)$)
CD() = new c,d ($C_D(c,d)$ | $D_C(d,c)$)
AD() = new a,d ($A_D(a,d)$ | $D_A(d,a)$)
CB() = new c,b ($C_B(c,b)$ | $B_C(b,c)$)

(AB() | CD() | AD() | CB())

N.B. it would be trivial to treat this as an X+Y=Z+W reaction, but the idea here is that each of A,B,C,D is not an isolated molecule, but may be attached to other things, e.g. it may be part of a polymer; those connections, and the identities of A,B,C,D should be preserved by the recombination.

Best idea: use molecule identities. (Try instead generalizing the Swap example by reusing connections: it's hard, and it seems to lead to recursive channels!)

$A_B(a,b)$ means "I am a connected to b" where a,b are molecule identities.

An A in state AB looks for a CD complex by communicating with a C in state CD over a public channel cd. Note "?!cd((c,d),(a,b))"; it means that A_B and C_D start the recombination protocol by swapping their identities and all the other identities they know. Then !b(c) means that B_A, through its molecule identity b, is told to disconnect (from A) and to reconnect to c.

B and D have a more passive role; they are just being told how to reconnect over their molecule identities.

a:chan; c:chan; b:chan(chan); d:chan(chan)
recomb intitiation rates are attached to cd,cb
recomb dissociation rates are attached to b,d

Exercise: Middle Out

Polymerization

Bidirectional Polymerization

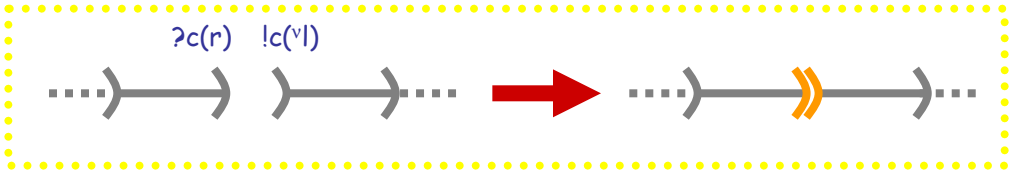
new c@μ new stop@1.0

A_{free} =
  !c(νrht_λ); A_{brht}(rht)) +
  ?c(lft); A_{blft}(lft)

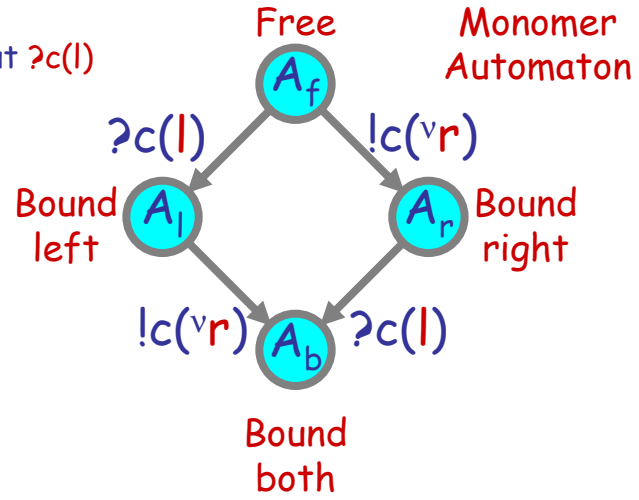
A_{blft}(lft) =
  !c(νrht_λ); A_{bound}(lft,rht))

A_{brht}(rht) =
  ?c(lft); A_{bound}(lft,rht)

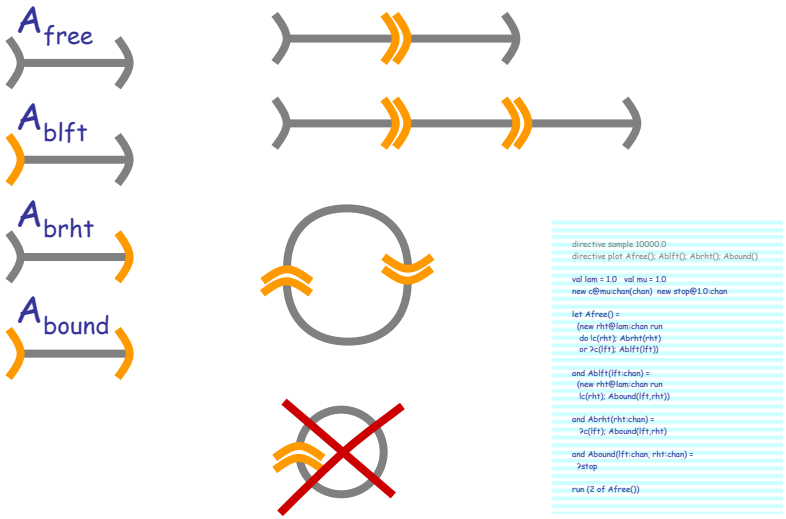
A_{bound}(lft,rht) = ?stop



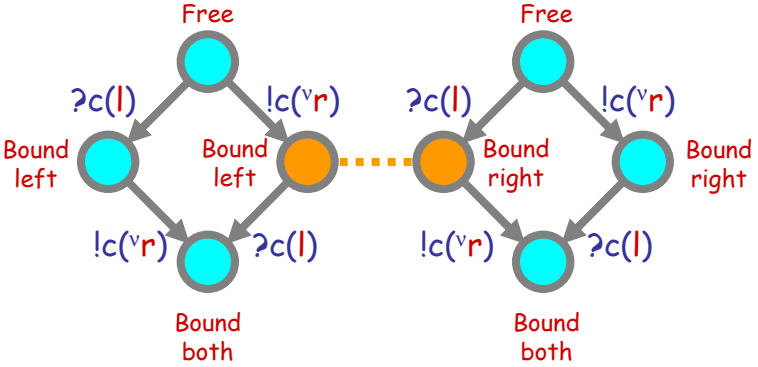
Communicating Automata
Bound output !c(νr) and input ?c(l)
on automata transitions
to model complexation



Polymerization is iterated complexation.



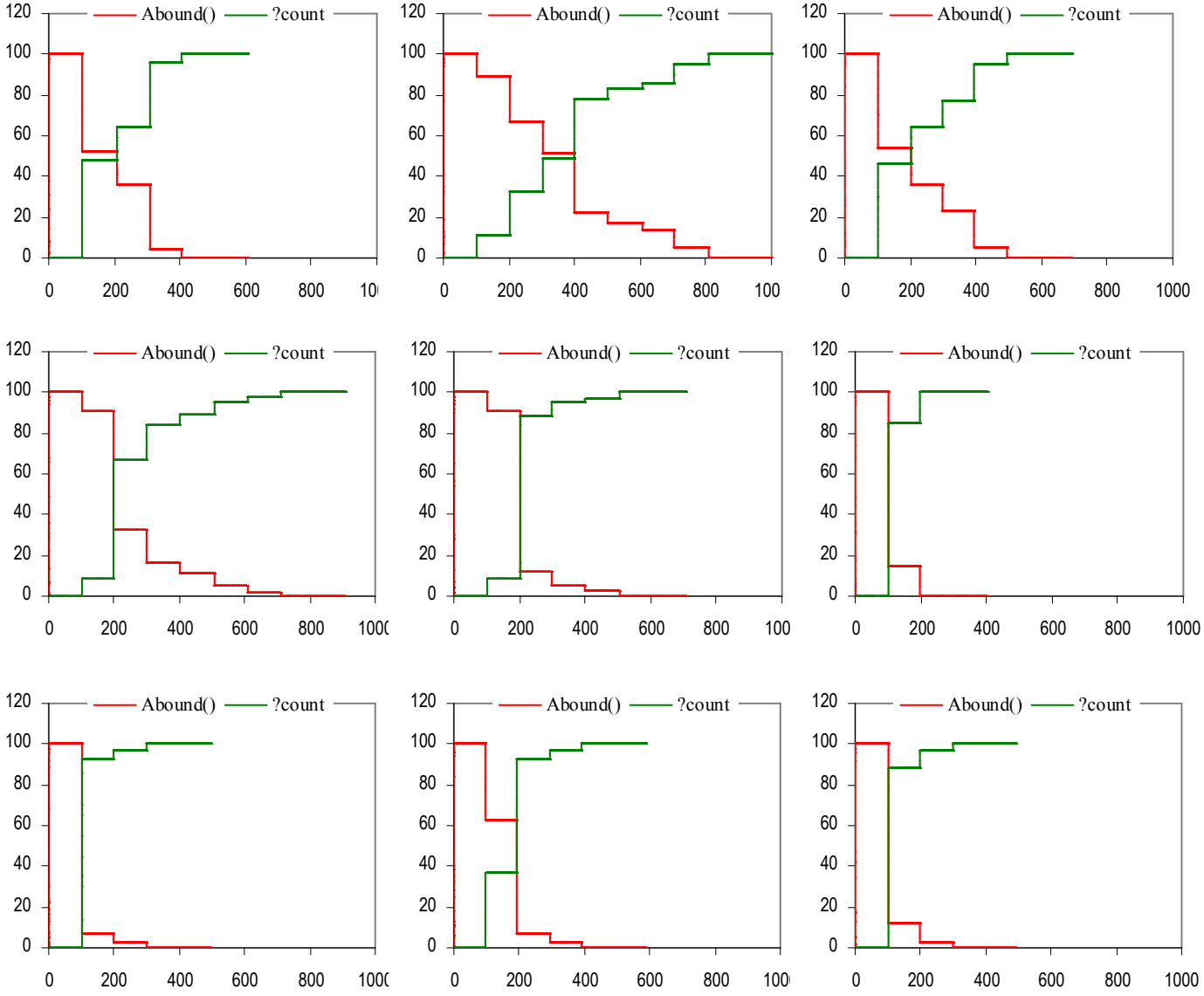

```
directive sample 10000.0
directive plot Afree(); Ablft(); Abrht(); Abound()

val lam = 1.0   val mu = 1.0
new c@mu:chan(chan)  new stop@1.0:chan

let Afree() =
 (new rht@lam:chan run
  do !c(rht); Abrht(rht)
  or ?c(lft); Ablft(lft))

and Ablft(lft:chan) =
 (new rht@lam:chan run
  !c(rht); Abound(lft,rht))

and Abrht(rht:chan) =
  ?c(lft); Abound(lft,rht)

and Abound(lft:chan, rht:chan) =
  ?stop

run (2 of Afree())
```



Bidirectional Polymerization

Circular Polymer Lengths

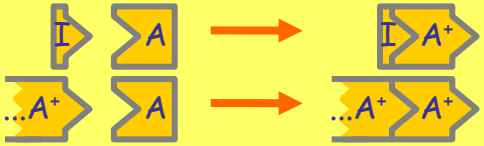
Scanning and counting the size of the circular polymers (by a cheap trick).
Polymer formation is complete within 10t; then a different polymer is scanned every 100t.




```
directive sample 1000.0
directive plot Abound(); ?count

type Link = chan(chan)
type Barb = chan

val lam = 1000.0 (* set high for better counting *)
val mu = 1.0
new c@mu:chan(Link)
new enter@lam:chan(Barb)
new count@lam:Barb

let Afree() =
  (new rht@lam:Link run
   do !c(rht); Abrht(rht)
   or ?c(lft); Ablft(lft))

and Ablft(lft:Link) =
  (new rht@lam:Link run
   !c(rht); Abound(lft,rht))

and Abrht(rht:Link) =
  ?c(lft); Abound(lft,rht)

and Abound(lft:Link, rht:Link) =
  do ?enter(barb); (?barb | !rht(barb))
  or ?lft(barb); (?barb | !rht(barb))
(* each Abound waits for a barb, exhibits it, and passes it to
   the right so we can plot number of Abound in a ring *)

let clock(t:float, tick:chan) =      (* sends a tick every t time *)
  (val ti = t/1000.0 val d = 1.0/ti
   let step(n:int) =
     if n<=0 then !tick; clock(t,tick) else delay@d; step(n-1)
   run step(1000))

new tick:chan
let Scan() = ?tick; !enter(count); Scan()

run 100 of Afree()
run (clock(100.0, tick) | Scan())
```

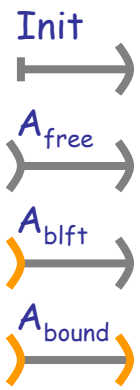
$100 \times A_{free}$, initially.

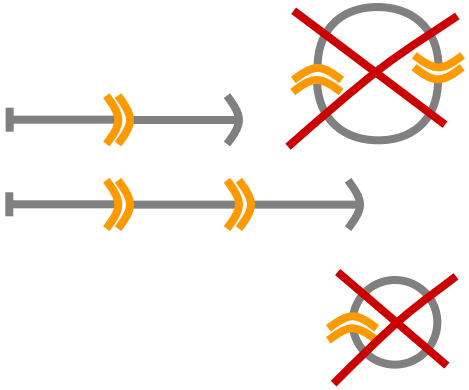
The height of each rising step is the size of a separate circular polymer. (Unbiased sample of nine consecutive runs.)

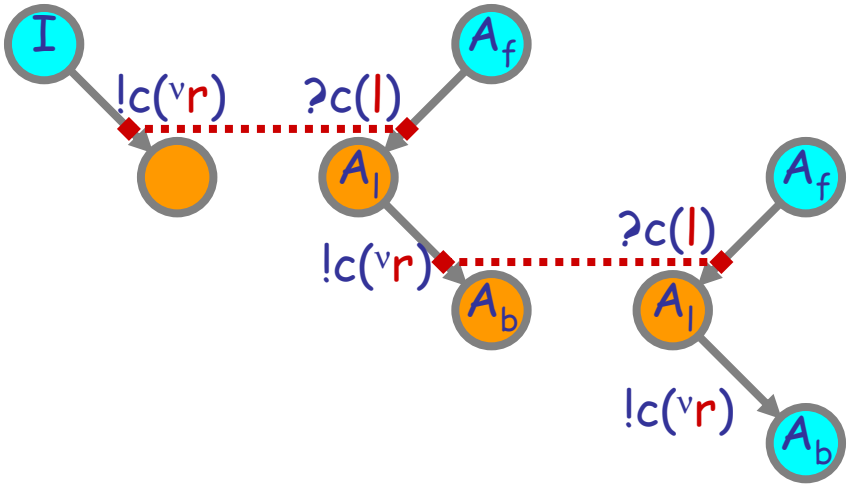


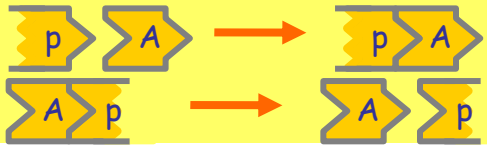
Unidirectional
Polymerization

```
new c@μ  new stop@1.0

Init =
   !c(νrht_λ); ?stop

A_free =
   ?c(lft); A_blft(lft)

A_blft(lft) =
   !c(νrht_λ); A_bound(lft,rht)

A_bound(lft,rht) =  ?stop
```









Actin-like
Poly/Depolymerization

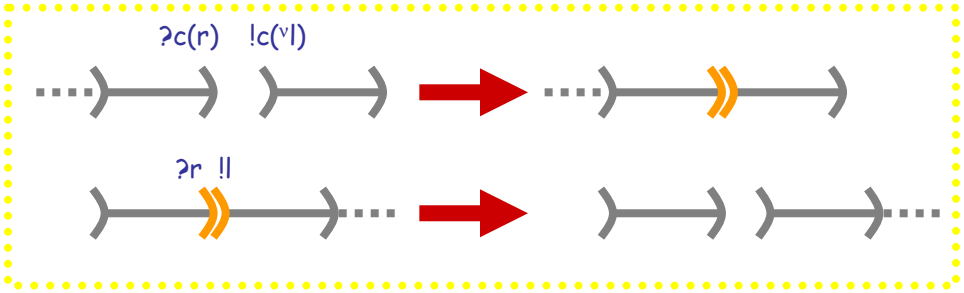
new c@μ

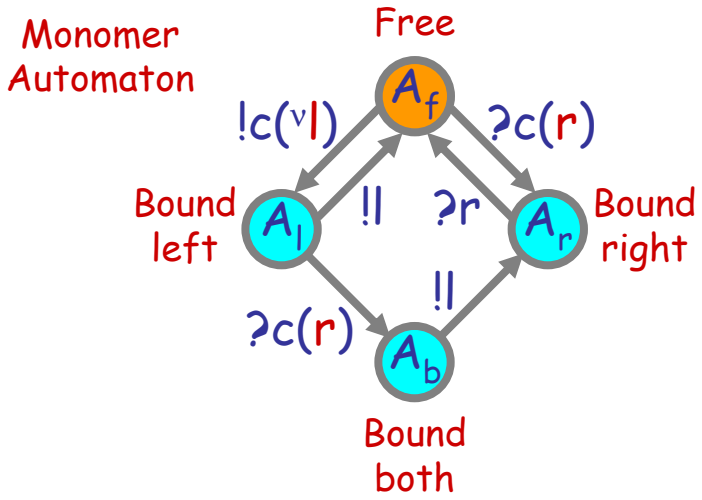
A_{free} =
  $!c(^{\nu}lft_{\lambda})$; $A_{blft}(lft)$) +
  $?c(rht)$; $A_{brht}(rht)$

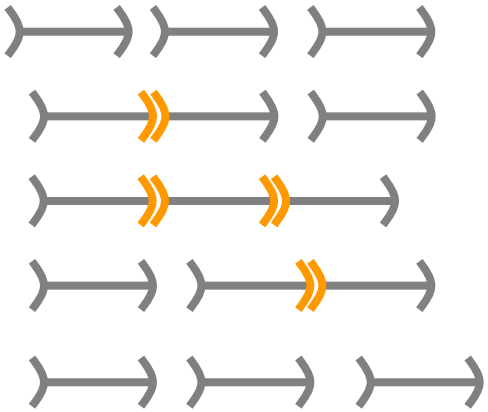
$A_{blft}(lft)$ =
  $!lft$; A_{free} +
  $?c(rht)$; $A_{bound}(lft,rht)$

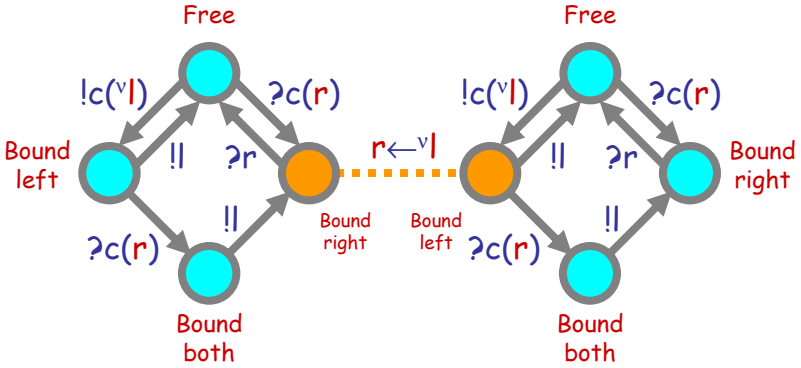
$A_{brht}(rht)$ =
  $?rht$; A_{free}

$A_{bound}(lft,rht)$ =
  $!lft$; $A_{brht}(rht)$

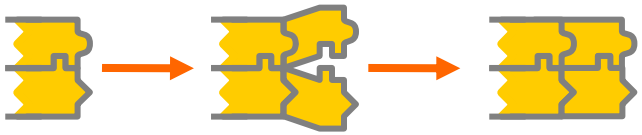








Exercise: Zipper



Complex Complexity

Complexes: The Chemical Way

n domains: A, B, C

2n domain reactions:
$A \leftrightharpoons A_p$
$B \leftrightharpoons B_p$
$C \leftrightharpoons C_p$

1 complex: ABC

2^n species:
ABC
A_pBC
AB_pC
ABC_p
A_pB_pC
A_pBC_p
AB_pC_p
$A_pB_pC_p$

$2n(2^{n-1})$ reactions (twice number of edges in n-dim hypercube):
$ABC \leftrightharpoons A_pBC$
$ABC \leftrightharpoons AB_pC$
$ABC \leftrightharpoons ABC_p$
$A_pBC \leftrightharpoons A_pB_pC$
$A_pBC \leftrightharpoons A_pBC_p$
$AB_pC \leftrightharpoons A_pB_pC$
$AB_pC \leftrightharpoons AB_pC_p$
$ABC_p \leftrightharpoons A_pBC_p$
$ABC_p \leftrightharpoons AB_pC_p$
$A_pB_pC \leftrightharpoons A_pB_pC_p$
$A_pBC_p \leftrightharpoons A_pB_pC_p$
$AB_pC_p \leftrightharpoons A_pB_pC_p$

The matrix is very sparse, so the corresponding ODE system is not dense. But it still has 2^n equations, one per species, plus conservation equations ([ABC]+[A_pBC]=constant, etc.).

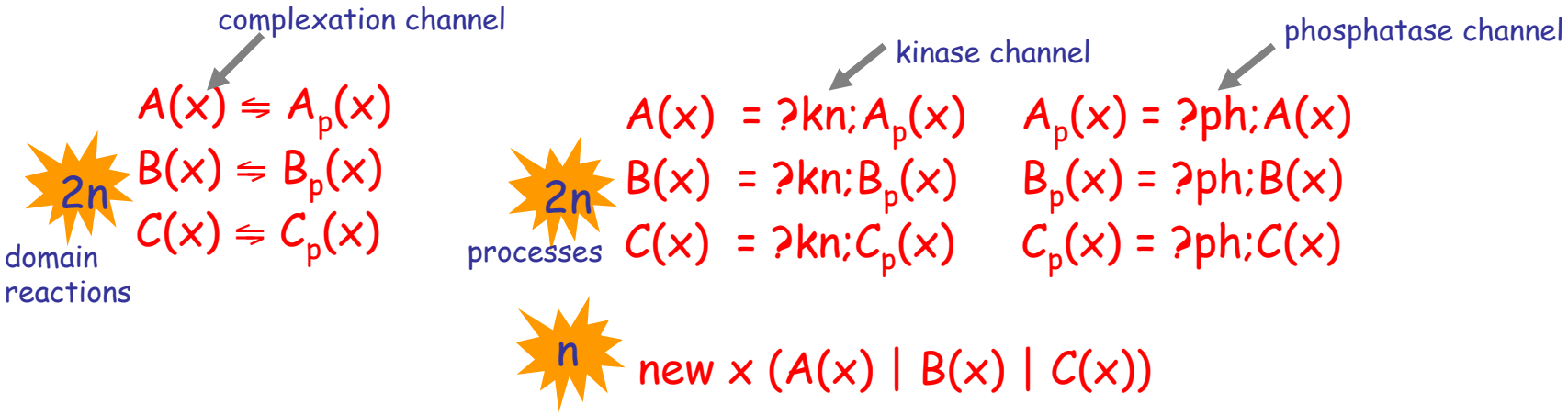
System description is exponential in the number of basic components.

Stoichiometric Matrix

N	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}	v_{16}	v_{17}	v_{18}	v_{19}	v_{20}	v_{21}	v_{22}	v_{23}	v_{24}
ABC																								
ApBC																								
ABpC																								
ABCp																								
ApBpC																								
ApBCp																								
ABpCp																								
ApBpCp																								

$2^n \times 2n(2^{n-1})$

Complexes: The Process Way



When the local domain reactions are not independent, we can use lateral communication so that each component is aware of the relevant others.

System description is linear in the number of basic components.

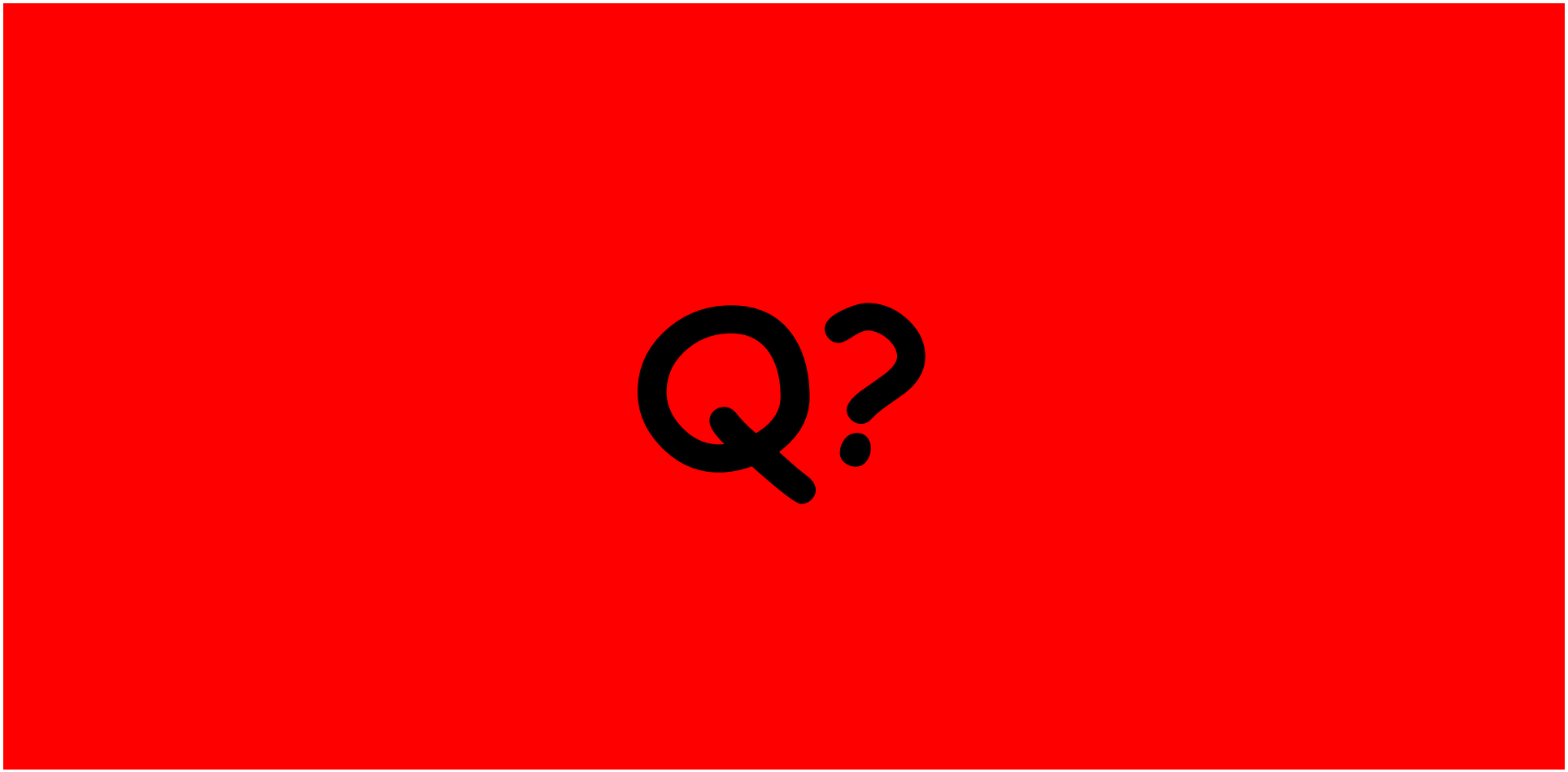
(Its "run-time" behavior or analysis potentially blows-up just as in the previous case.)

Summary

- Complexation
  - Requires the "full power" of π-calculus.
  - Or possibly an "interesting" finite subset of it (Cf. history-dependent automata).

- Polymerization
  - Automata that stick together.
  - Easily done in π-calculus, but beyond standard automata theory.

- Compositionality
  - Complexation leads to exponential blowup of state space (and of chemical and ODE based descriptions).


Q?