Clock domains in a reactive functional language

Cédric Pasteur

INRIA Team PARKAS
Laboratoire d'Informatique de l'École normale supérieure

28 nov. 2011

A reactive extension to ML

- ML: higher order functional language
- Synchronous primitives a la Esterel
  - Logical instants, processes, signals (first-class values)
  - Causality by construction
- Efficient sequential implementation
  - Dynamic scheduling (dynamic creation)
- Discrete simulation (sensor networks)

A simple example

- The factorial in n steps

```
let rec process pfact n =
  pause;
  if n <= 1 then 1
  else
    let v = run (pfact (n-1)) in
    print_int (n*v)

run (pfact 2) || run (pfact 3)
```

A simple example

- The factorial in n steps

```
let rec process pfact n =
  pause;
  if n <= 1 then 1
  else
    let v = run (pfact (n-1)) in
    print_int (n*v)

run (pfact 2) || run (pfact 3)
```

A simple example

- The factorial in n steps

```
let rec process pfact n =
  pause;
  if n <= 1 then 1
  else
    let v = run (pfact (n-1)) in
    print_int (n*v)

run (pfact 2) || run (pfact 3)
```

A simple example

- The factorial in n steps

```
let rec process pfact n =
  pause;
  if n <= 1 then 1
  else
    let v = run (pfact (n-1)) in
    print_int (n*v)

run (pfact 2) || run (pfact 3)
```

A simple example

- The factorial in n steps

```
let rec process pfact n =
  pause;
  if n <= 1 then 1
  else
    let v = run (pfact (n-1)) in
    print_int (n*v)

run (pfact 2) || run (pfact 3)
```

Communication through broadcast signals

- Multi-emission
- One instant to get the value of a signal

```
signal s default 0 gather (+) in
    emit s(1)
 || emit s(-1)
 || await s(v) in print_int v
```

Communication through broadcast signals

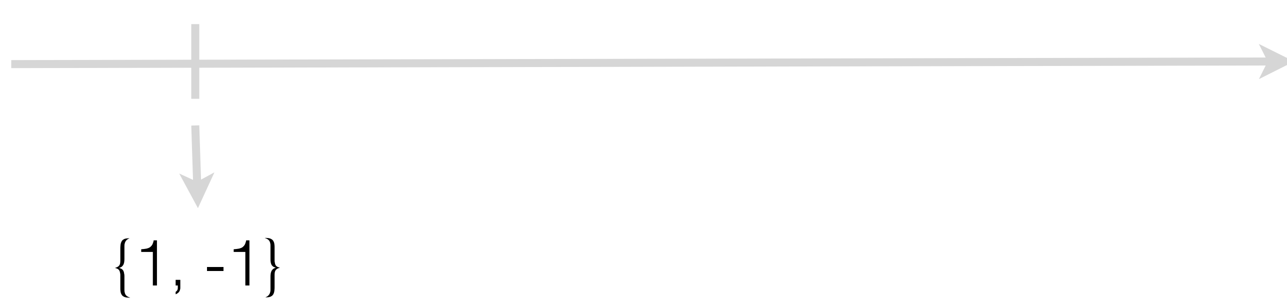
- Multi-emission
- One instant to get the value of a signal

```
signal s default 0 gather (+) in
    emit s(1)
 || emit s(-1)
 || await s(v) in print_int v
```

Communication through broadcast signals

- Multi-emission
- One instant to get the value of a signal

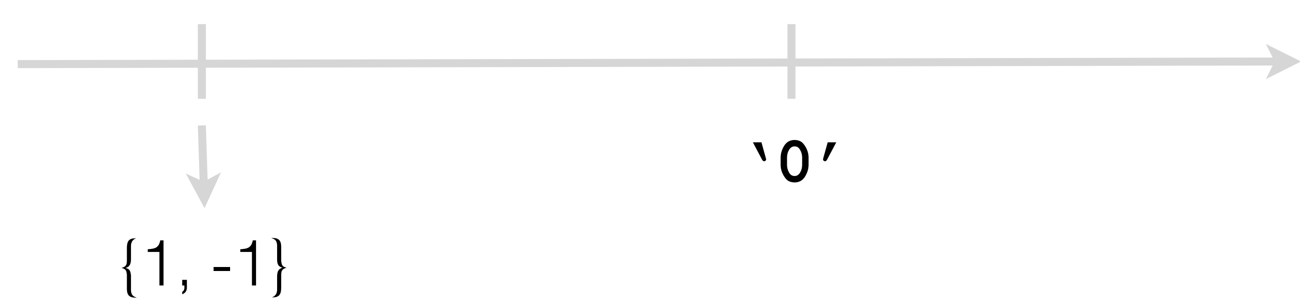
```
signal s default 0 gather (+) in
     emit s(1)
  || emit s(-1)
  || await s(v) in print_int v
```



Communication through broadcast signals

- Multi-emission
- One instant to get the value of a signal

```
signal s default 0 gather (+) in
      emit s(1)
  ||  emit s(-1)
  ||  await s(v) in print_int v
```



Demo: n-body

Simulate a solar system

- Simulate the gravitational interactions of n bodies
- Equation:

$$m_i \vec{a}_i = \vec{f}_i = \sum_j \vec{F}_{i,j}$$

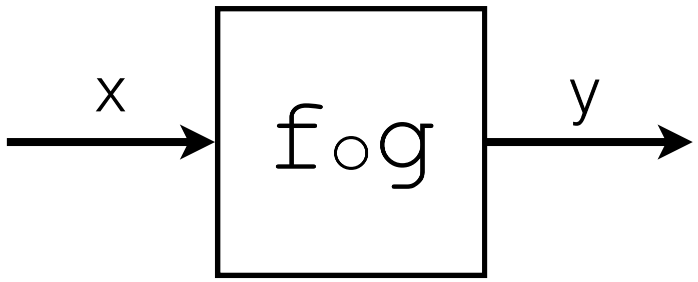
- Fixed-step numerical methods

Demo: n-body

```
type state =
 { mutable b_pos : vector; mutable b_vel : vector;
   b_weight : float; }

let dt = 0.1
signal env default (fun _ -> zero_vector) gather add_force

let compute_euler st f =
  st.b_pos <- add_v st.b_pos (sc_mult dt st.b_vel);
  st.b_vel <- add_v st.b_vel (sc_mult dt f)

let process body =
  let st = new_state () in
  loop
    emit env (force st);
    await env(f) in
    compute_euler st (f st.b_pos)
  end
```

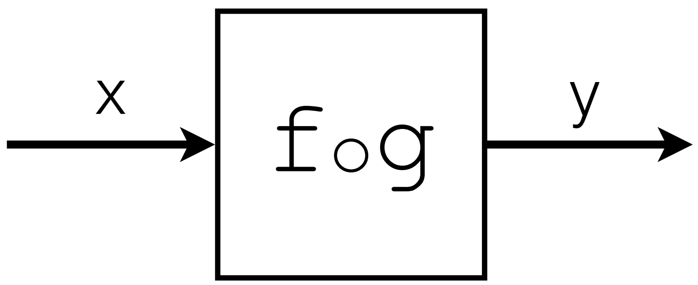
Modularity problem

Modularity problem

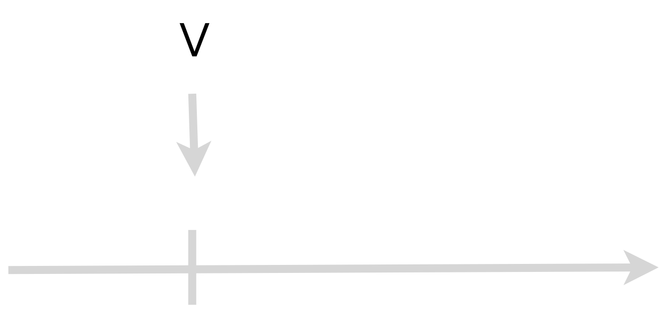



```
await x(v) in
  emit y (f (g v))
```

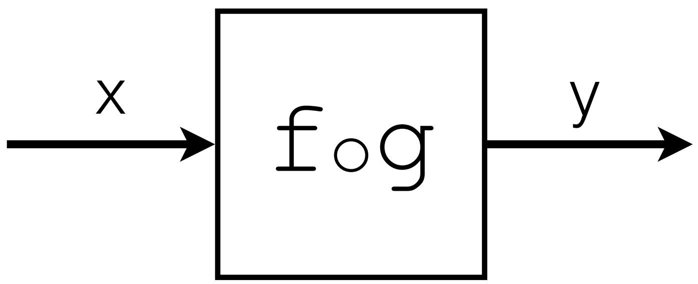
Modularity problem



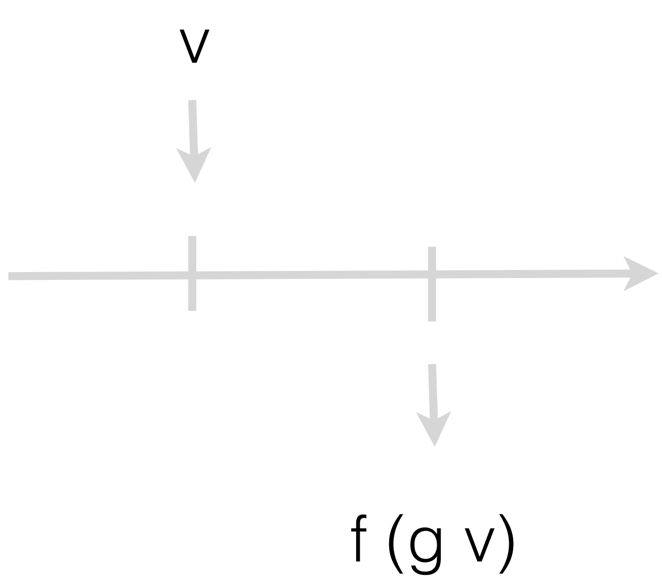

```
await x(v) in
  emit y (f (g v))
```



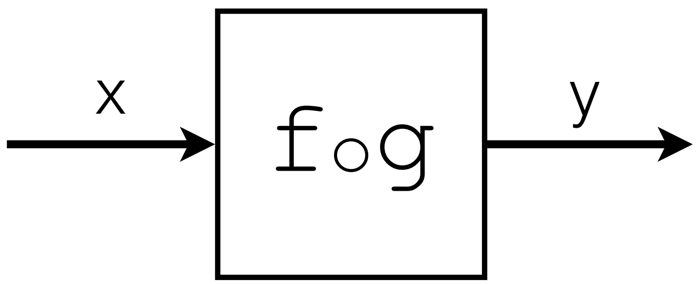
Modularity problem



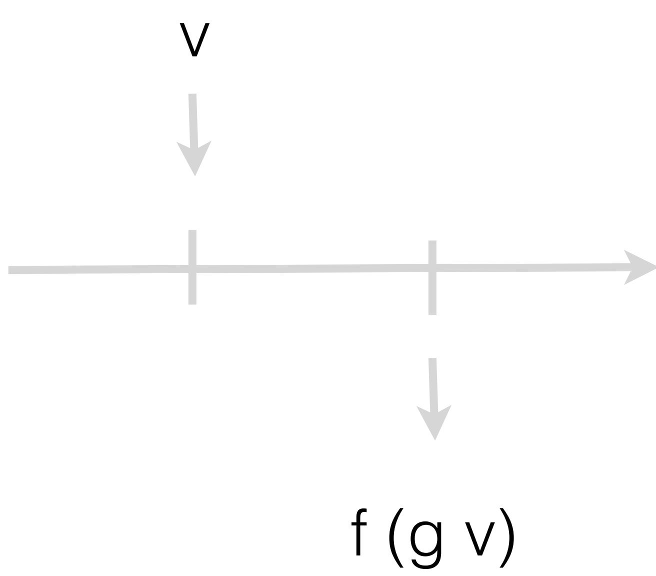

```
await x(v) in
  emit y (f (g v))
```

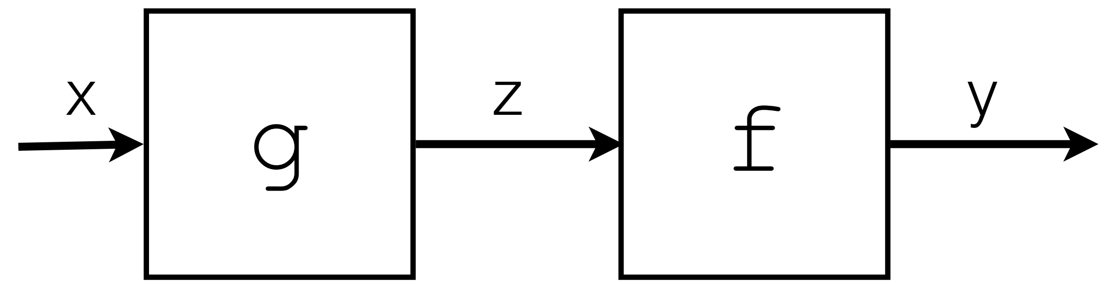


Modularity problem



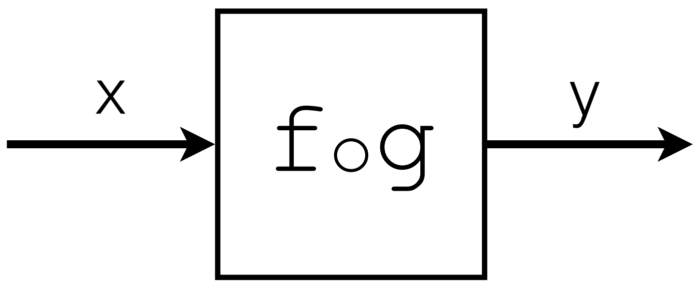

```
await x(v) in
  emit y (f (g v))
```



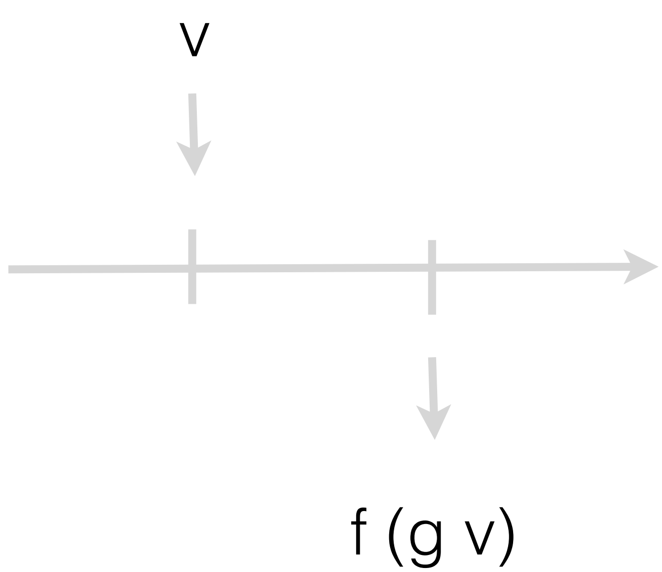


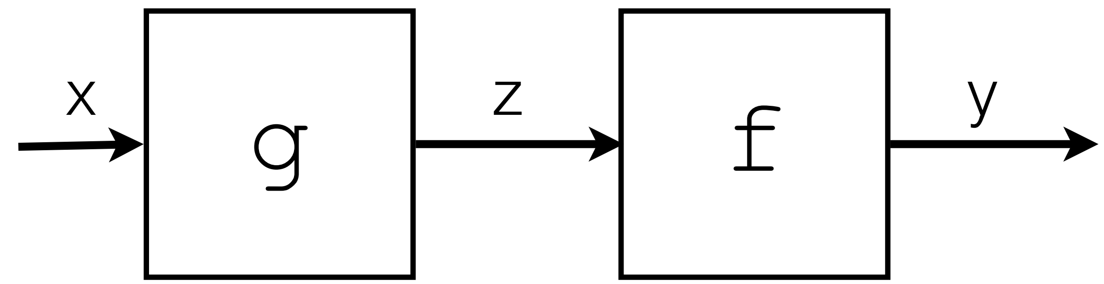

```
signal z in
     await x(v) in
      emit z(g v)
  || await z(v) in
      emit y(f v)
```

Modularity problem



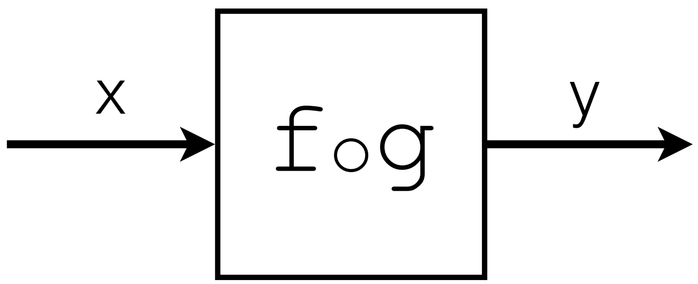

```
await x(v) in
  emit y (f (g v))
```

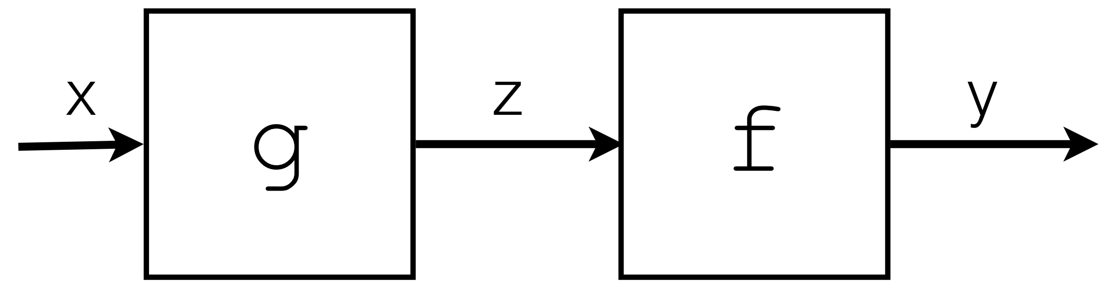





```
signal z in
     await x(v) in
      emit z(g v)
  || await z(v) in
      emit y(f v)
```

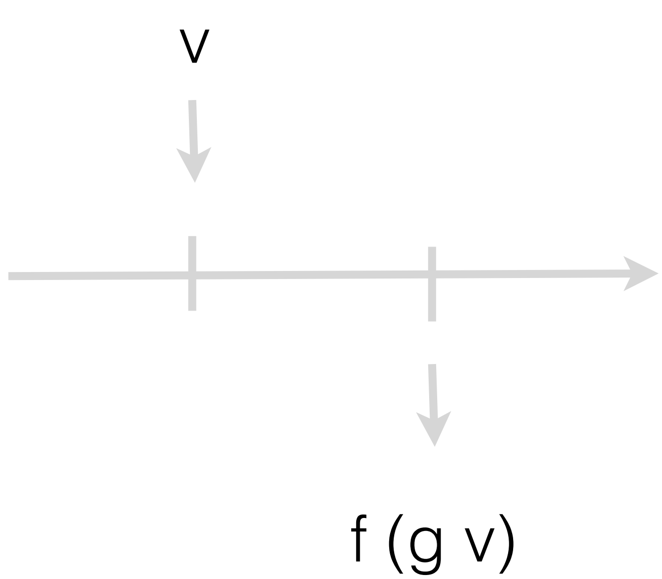
Modularity problem



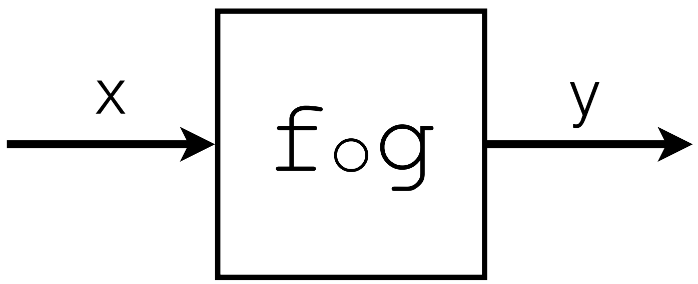


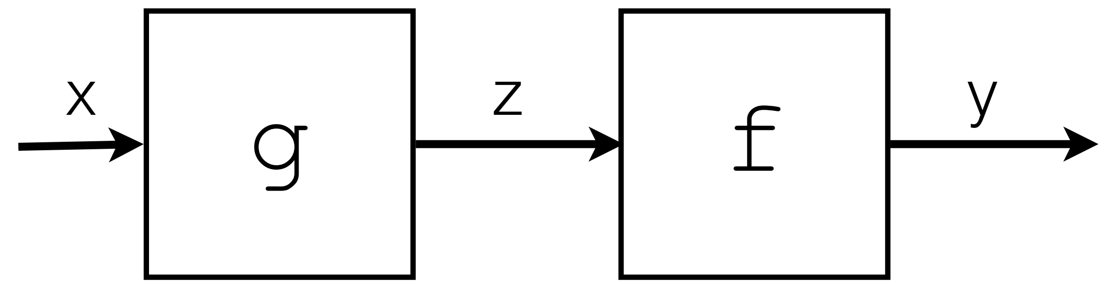

```
await x(v) in
  emit y (f (g v))
```

```
signal z in
     await x(v) in
      emit z(g v)
  || await z(v) in
      emit y(f v)
```



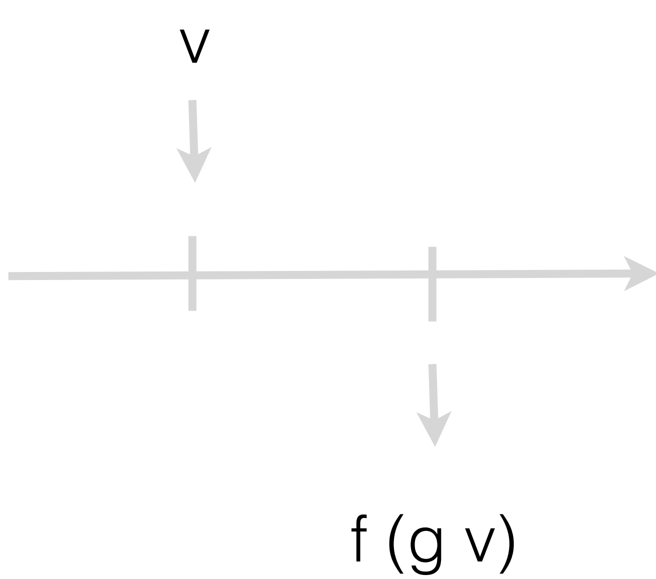
Modularity problem

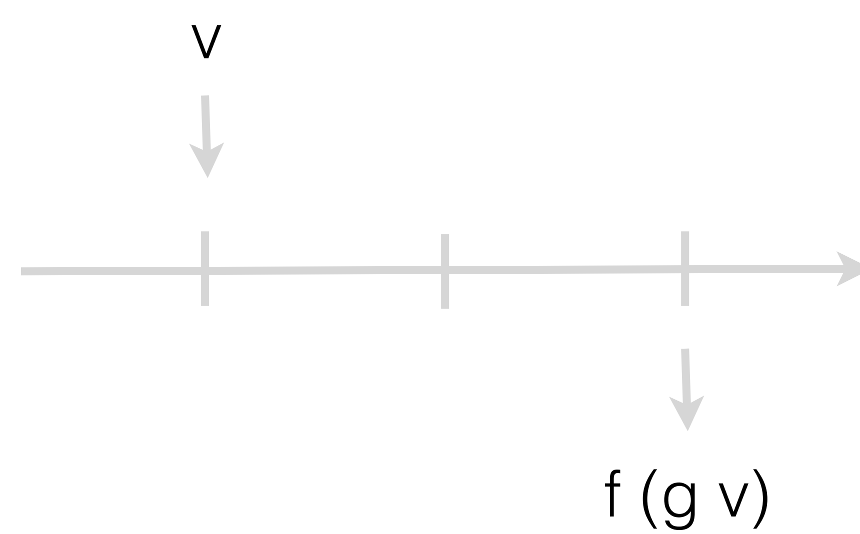




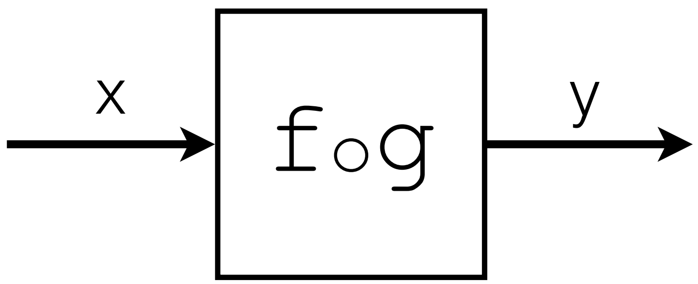

```
await x(v) in
  emit y (f (g v))
```

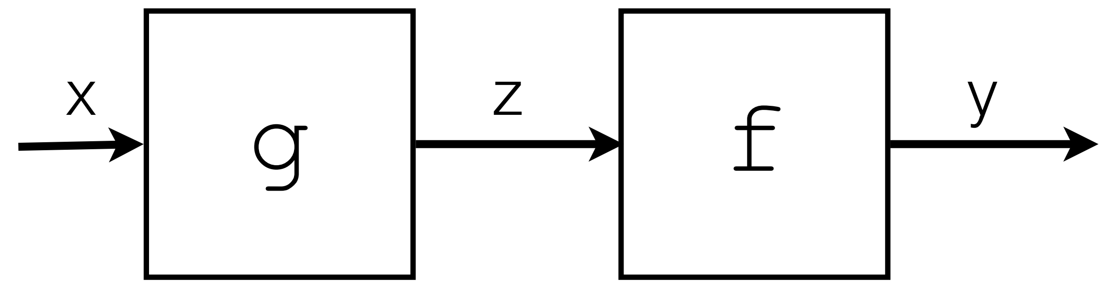
```
signal z in
     await x(v) in
      emit z(g v)
  || await z(v) in
      emit y(f v)
```





Modularity problem

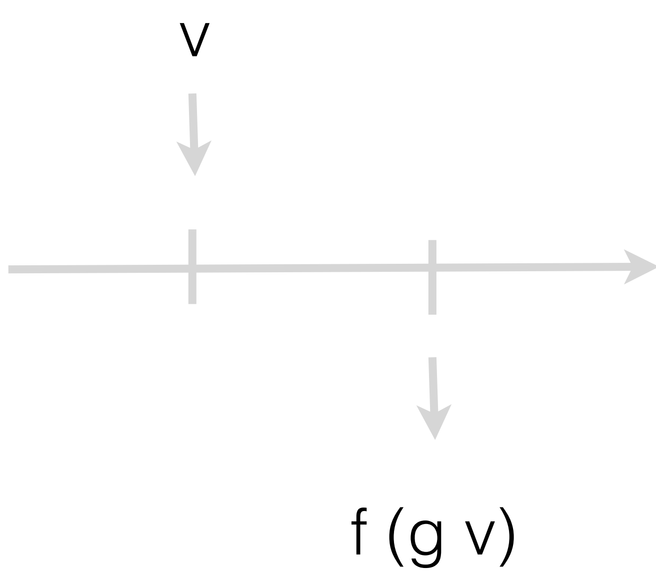


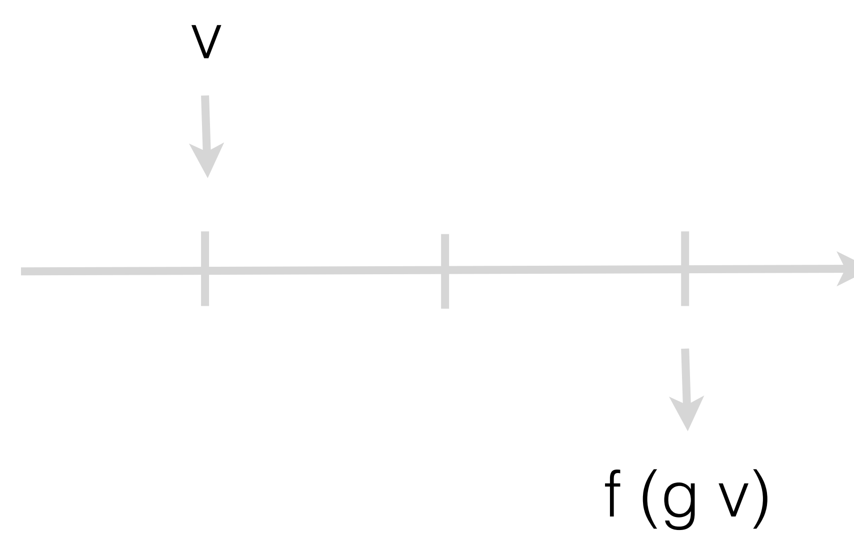



```
await x(v) in
  emit y (f (g v))
```

$\neq$

```
signal z in
     await x(v) in
      emit z(g v)
   || await z(v) in
      emit y(f v)
```





Modularity problem

Communicating takes time

- No causality analysis
- Enables dynamic creation of processes

Problems of compositionality

- It is hard to put processes with different rates in parallel
- Time refinement is hard

The proposed solution

Clock domains *(aka 'Clock refinement', Gemünde et al, Synchron 2009)*

- Local notion of instant
- Hides internal steps from the outside

The proposed solution

Clock domains *(aka 'Clock refinement', Gemünde et al, Synchron 2009)*

- Local notion of instant
- Hides internal steps from the outside

```
signal z in
     await x(v) in
      emit z(g v)
  || await z(v) in
      emit y(f v)
```

The proposed solution

Clock domains *(aka 'Clock refinement', Gemünde et al, Synchron 2009)*

- Local notion of instant
- Hides internal steps from the outside

```
signal z in
     await x(v) in
      emit z(g v)
  || await z(v) in
      emit y(f v)
```

The proposed solution

Clock domains *(aka 'Clock refinement', Gemünde et al, Synchron 2009)*

- Local notion of instant
- Hides internal steps from the outside

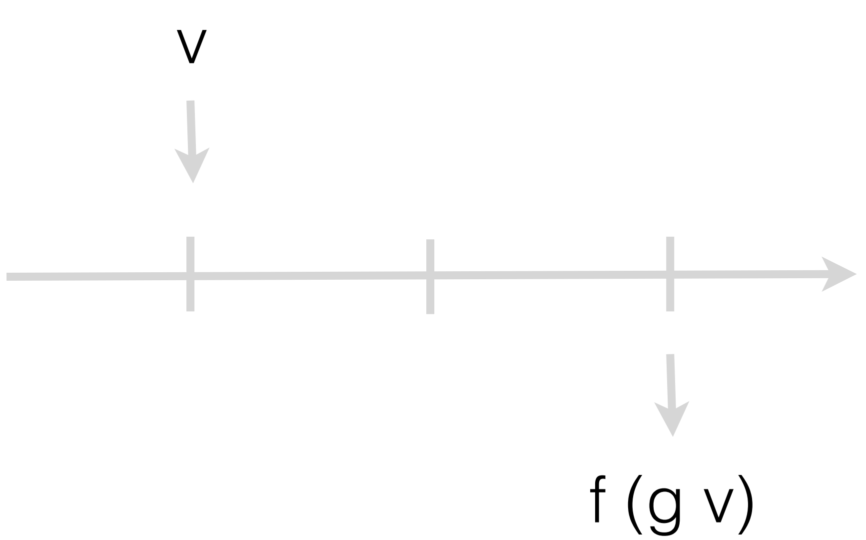
```
signal z in
     await x(v) in
      emit z(g v)
  || await z(v) in
      emit y(f v)
```

The proposed solution

Clock domains *(aka 'Clock refinement', Gemünde et al, Synchron 2009)*

- Local notion of instant
- Hides internal steps from the outside

```
signal z in
     await x(v) in
      emit z(g v)
  || await z(v) in
      emit y(f v)
```



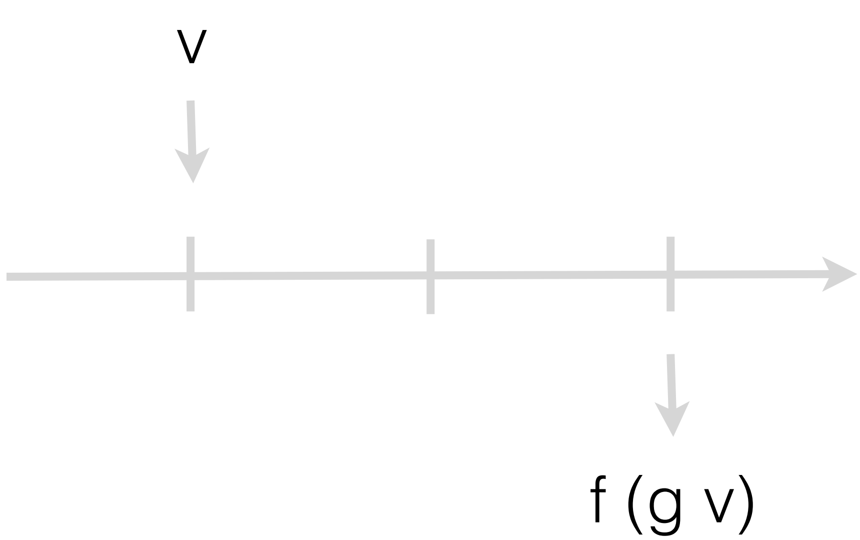
The proposed solution

Clock domains *(aka 'Clock refinement', Gemünde et al, Synchron 2009)*

- Local notion of instant
- Hides internal steps from the outside

```
signal z in
     await x(v) in
      emit z(g v)
  || await z(v) in
      emit y(f v)
```

```
newclock ck in
  signal z at ck in
       await x(v) in
        emit z(g v)
    || await z(v) in
        emit y(f v)
```

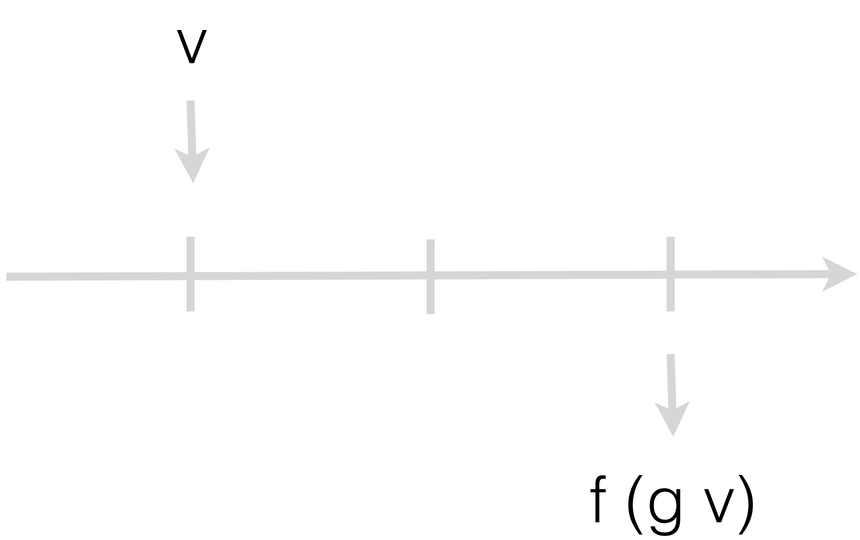


The proposed solution

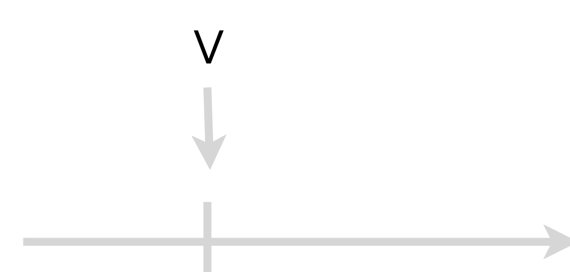
Clock domains *(aka 'Clock refinement', Gemünde et al, Synchron 2009)*

- Local notion of instant
- Hides internal steps from the outside

```
signal z in
     await x(v) in
      emit z(g v)
  || await z(v) in
      emit y(f v)
```




```
newclock ck in
  signal z at ck in
       await x(v) in
        emit z(g v)
    || await z(v) in
        emit y(f v)
```

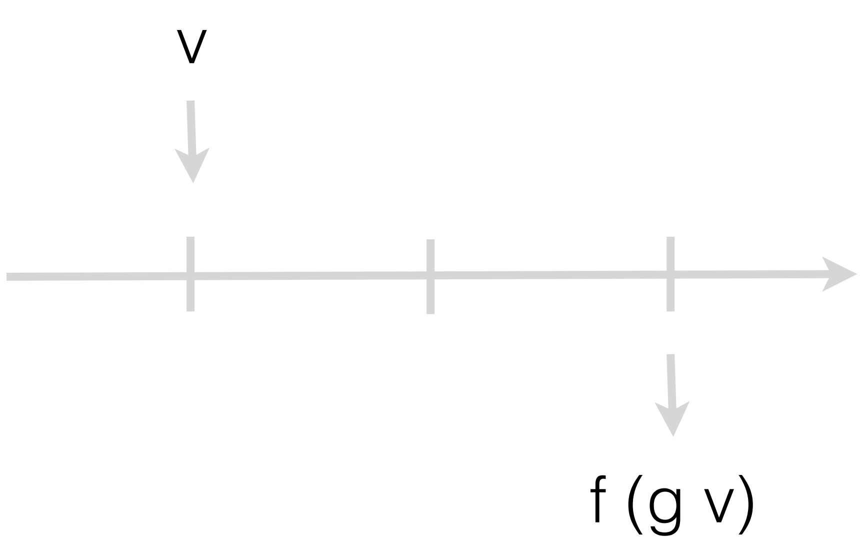


The proposed solution

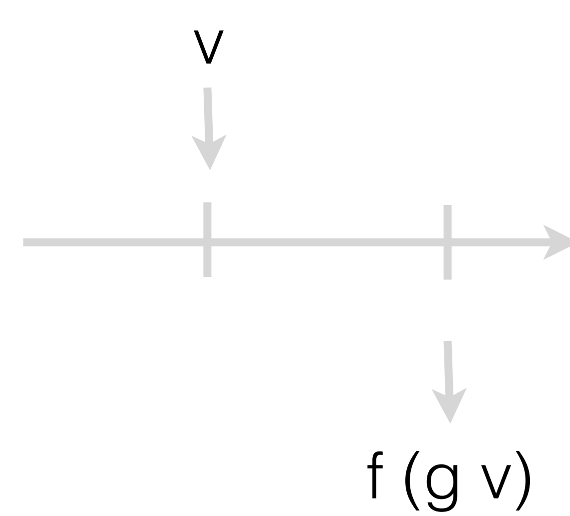
Clock domains *(aka 'Clock refinement', Gemünde et al, Synchron 2009)*

- Local notion of instant
- Hides internal steps from the outside

```
signal z in
     await x(v) in
      emit z(g v)
  || await z(v) in
      emit y(f v)
```




```
newclock ck in
  signal z at ck in
       await x(v) in
        emit z(g v)
    || await z(v) in
        emit y(f v)
```

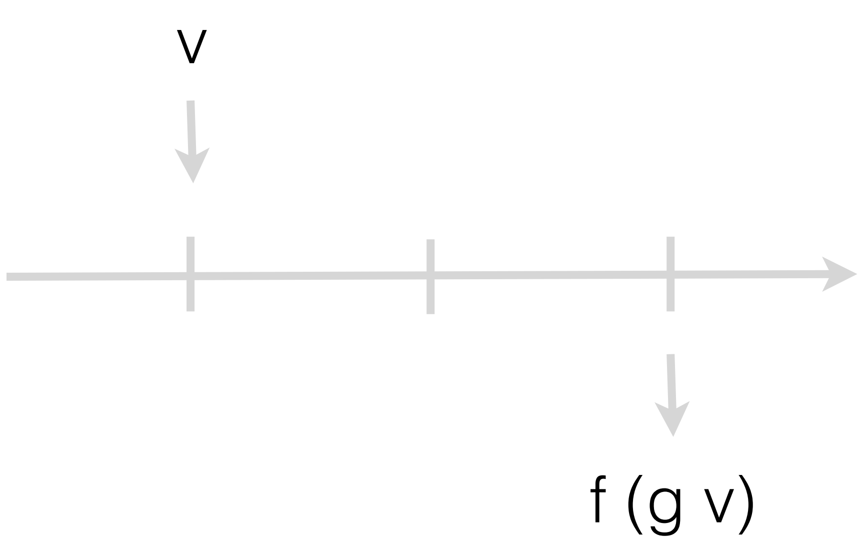


The proposed solution

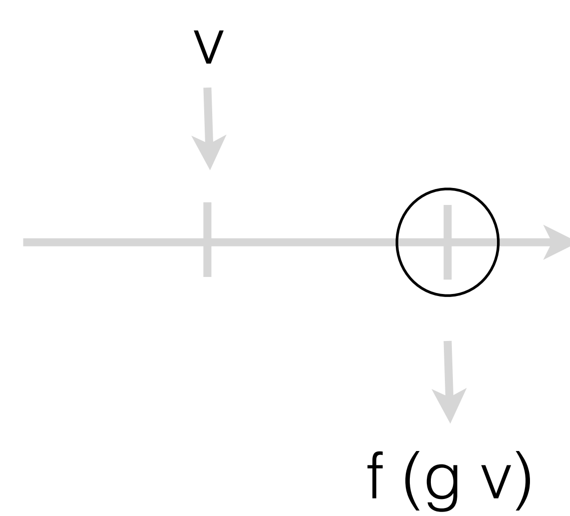
Clock domains *(aka 'Clock refinement', Gemünde et al, Synchron 2009)*

- Local notion of instant
- Hides internal steps from the outside

```
signal z in
     await x(v) in
      emit z(g v)
  || await z(v) in
      emit y(f v)
```




```
newclock ck in
  signal z at ck in
       await x(v) in
        emit z(g v)
    || await z(v) in
        emit y(f v)
```



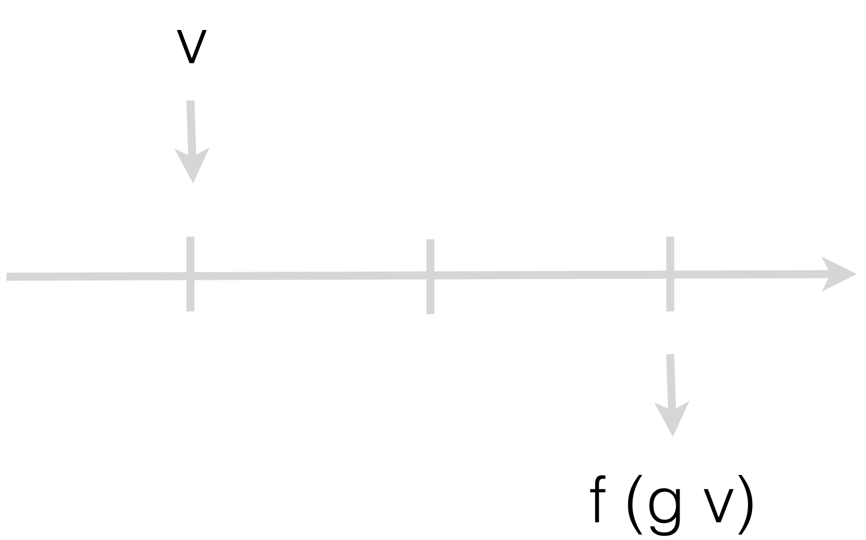
The proposed solution

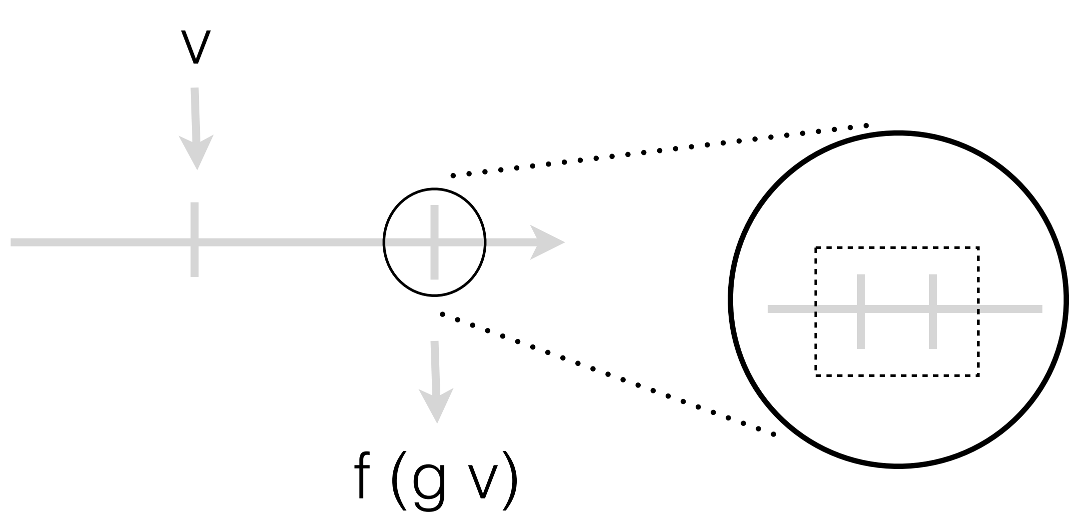
Clock domains *(aka 'Clock refinement', Gemünde et al, Synchron 2009)*

- Local notion of instant
- Hides internal steps from the outside

```
signal z in
      await x(v) in
       emit z(g v)
   || await z(v) in
       emit y(f v)
```

```
newclock ck in
  signal z at ck in
        await x(v) in
         emit z(g v)
     || await z(v) in
         emit y(f v)
```





Some examples

```
let process f s_out =
 newclock ck in
  loop
   emit s_out 1; pause ck;
   emit s_out 2; pause ck;
   emit s_out 3; pause topck
  end
```

A periodic clock domain

- Each instant of the top clock, **`ck`** does three steps

Some examples

```
let process f s_out =
 newclock ck in
  loop
   emit s_out 1; pause ck;
   emit s_out 2; pause ck;
   emit s_out 3; pause topck
  end
```

A periodic clock domain

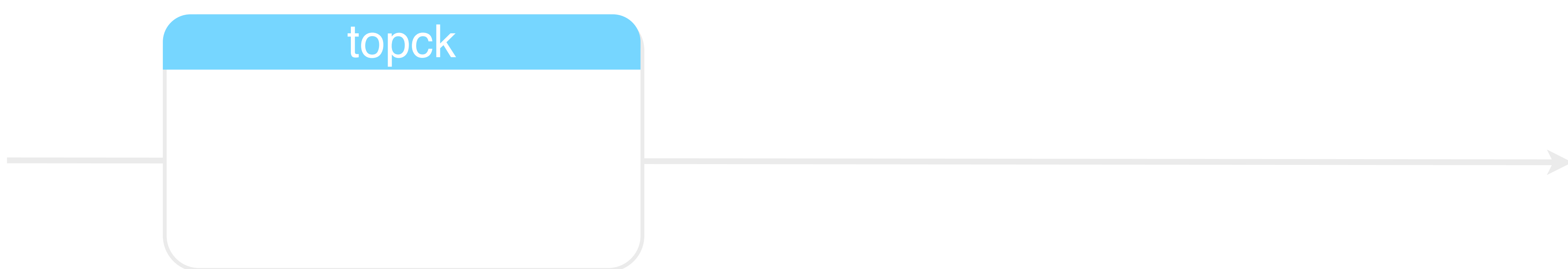
- Each instant of the top clock, **ck** does three steps

Some examples

```
let process f s_out =
 newclock ck in
  loop
   emit s_out 1; pause ck;
   emit s_out 2; pause ck;
   emit s_out 3; pause topck
  end
```

A periodic clock domain

- Each instant of the top clock, **ck** does three steps



Some examples

```
let process f s_out =
 newclock ck in
  loop
   emit s_out 1; pause ck;
   emit s_out 2; pause ck;
   emit s_out 3; pause topck
  end
```

A periodic clock domain

- Each instant of the top clock, **ck** does three steps

Some examples

```
let process f s_out =
 newclock ck in
  loop
   emit s_out 1; pause ck;
   emit s_out 2; pause ck;
   emit s_out 3; pause topck
  end
```

A periodic clock domain

- Each instant of the top clock, **`ck`** does three steps

Some examples

```
let process f s_out =
 newclock ck in
  loop
   emit s_out 1; pause ck;
   emit s_out 2; pause ck;
   emit s_out 3; pause topck
  end
```

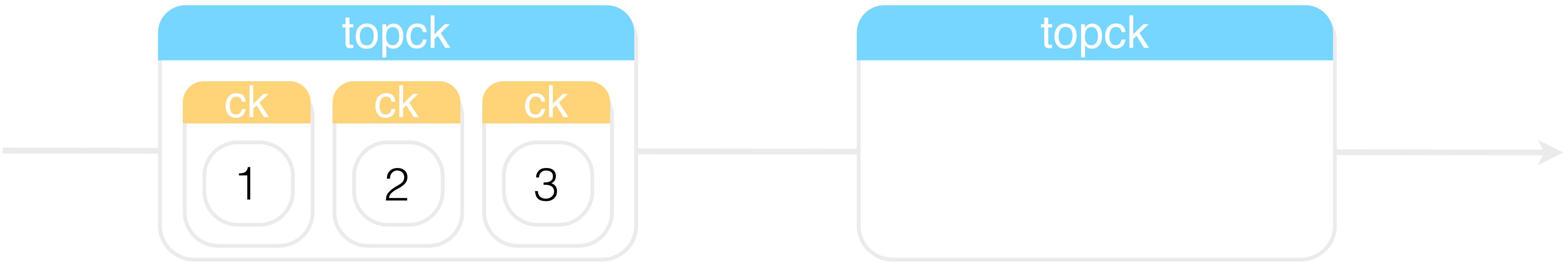
A periodic clock domain

- Each instant of the top clock, **ck** does three steps

```
let process f s_out =
 newclock ck in
  loop
   emit s_out 1; pause ck;
   emit s_out 2; pause ck;
   emit s_out 3; pause topck
  end
```

A periodic clock domain

- Each instant of the top clock, **`ck`** does three steps

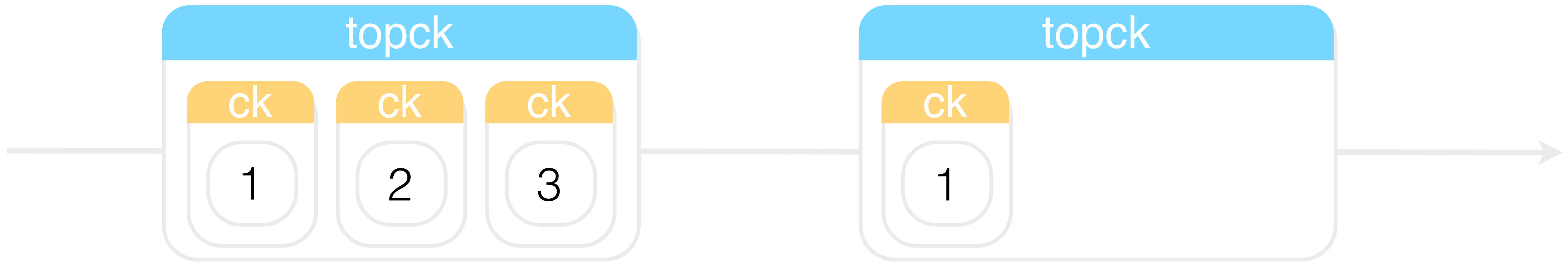


Some examples

```
let process f s_out =
 newclock ck in
  loop
   emit s_out 1; pause ck;
   emit s_out 2; pause ck;
   emit s_out 3; pause topck
  end
```

A periodic clock domain

- Each instant of the top clock, **ck** does three steps



A periodic clock domain

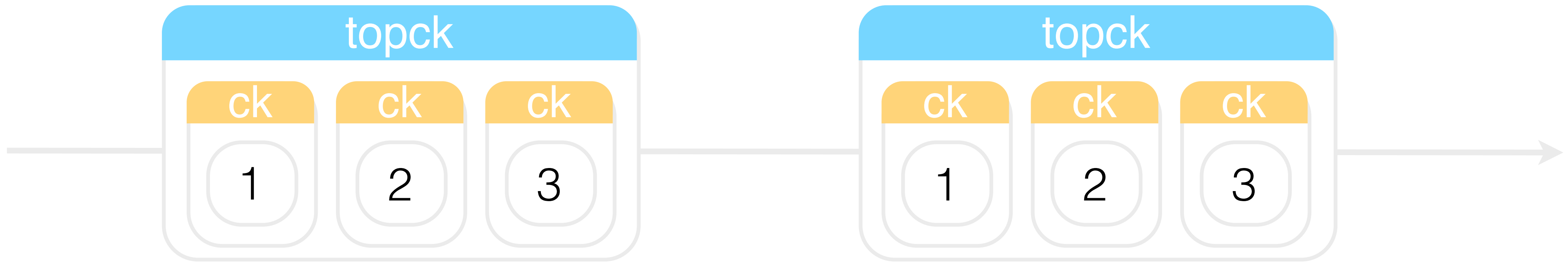
- Each instant of the top clock, `ck` does three steps

Some examples

```
let process f s_out =
 newclock ck in
  loop
   emit s_out 1; pause ck;
   emit s_out 2; pause ck;
   emit s_out 3; pause topck
  end
```

A periodic clock domain

- Each instant of the top clock, **`ck`** does three steps

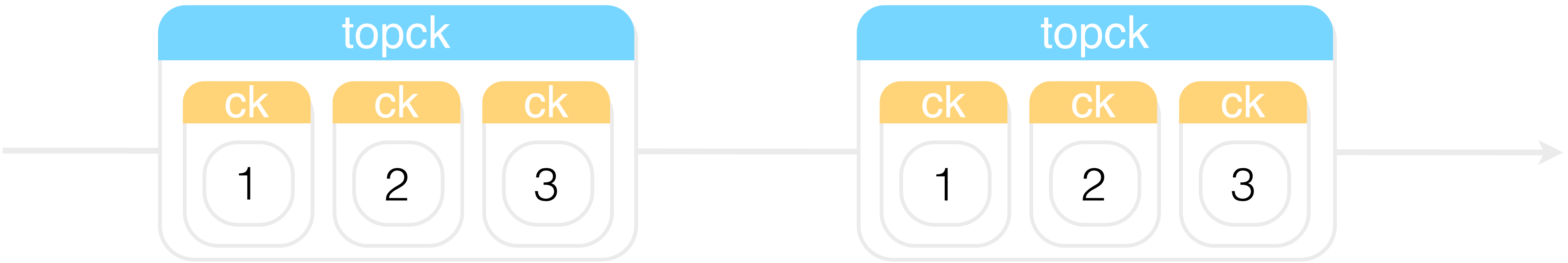


Some examples

```
let process f s_out =
 newclock ck in
  loop
   emit s_out 1; pause ck;
   emit s_out 2; pause ck;
   emit s_out 3; pause topck
  end
```

A periodic clock domain

- Each instant of the top clock, **`ck`** does three steps



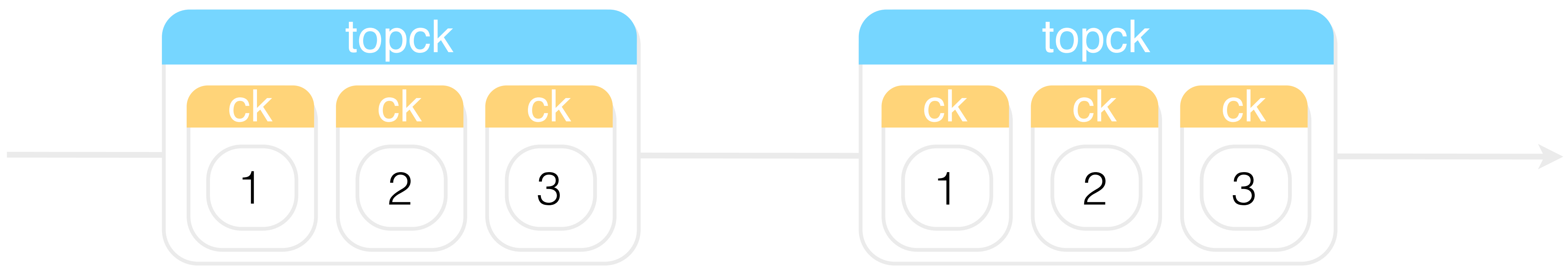
Some examples

```
let process f s_out =
 newclock ck in
  loop
   emit s_out 1; pause ck;
   emit s_out 2; pause ck;
   emit s_out 3; pause topck
  end
```

```
let process g s_out =
  loop
    emit s_out 1;
    emit s_out 2;
    emit s_out 3; pause
  end
```

A periodic clock domain

- Each instant of the top clock, **ck** does three steps



Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```

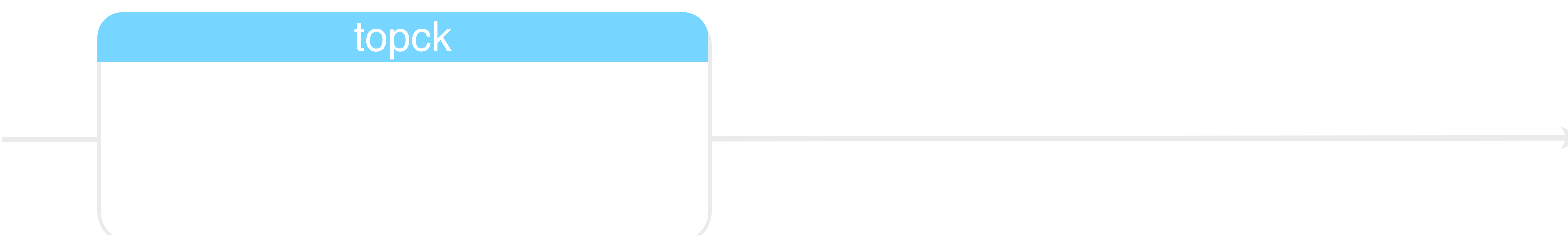
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```

||

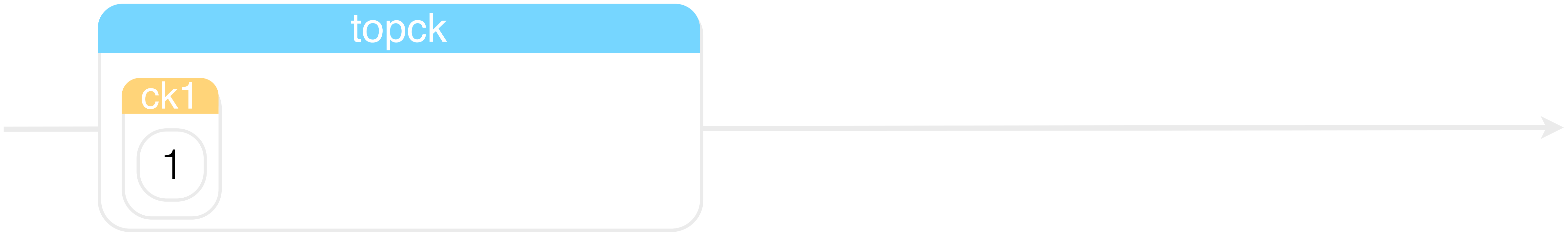
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



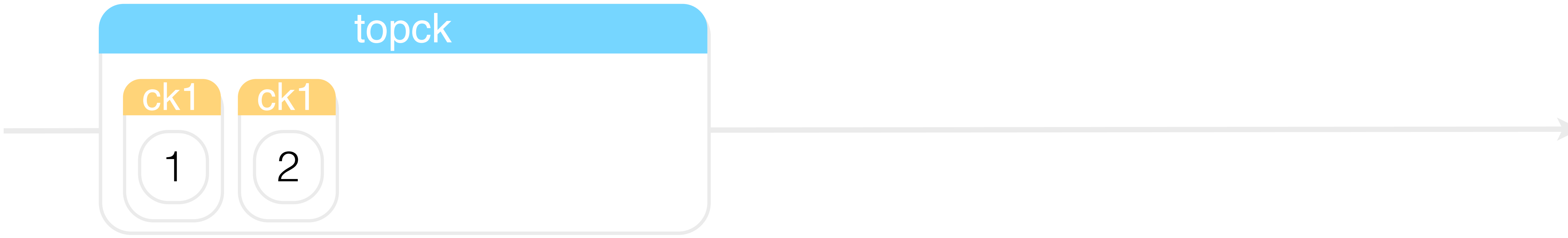
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



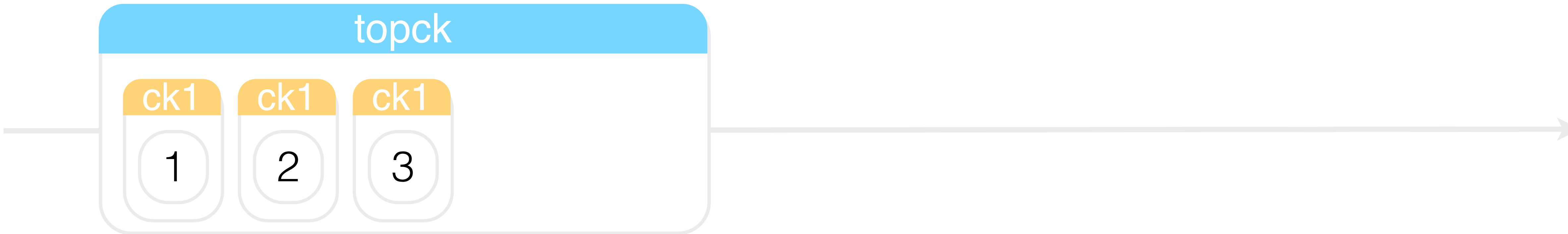
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



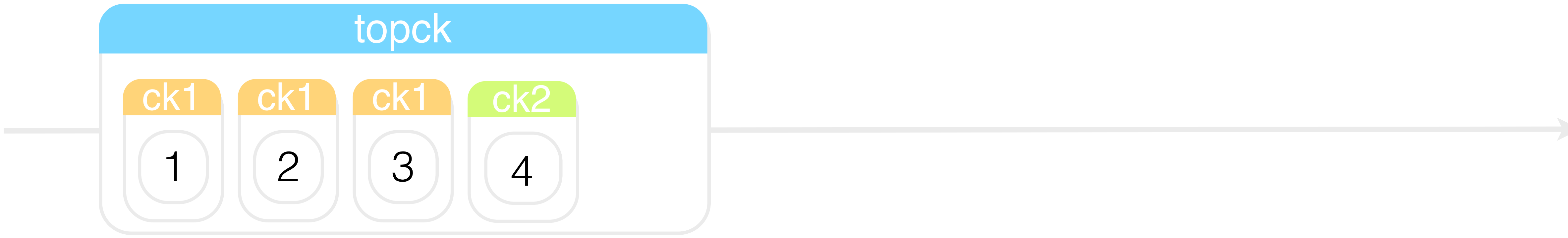
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



Some examples

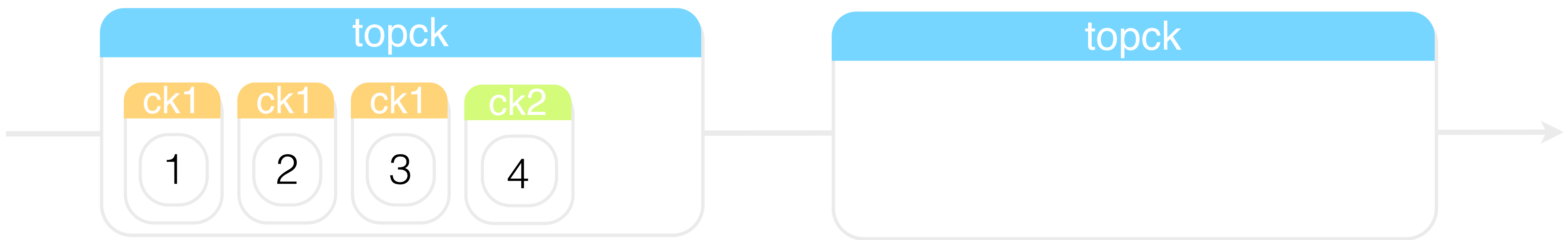
```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



lundi 28 novembre 2011

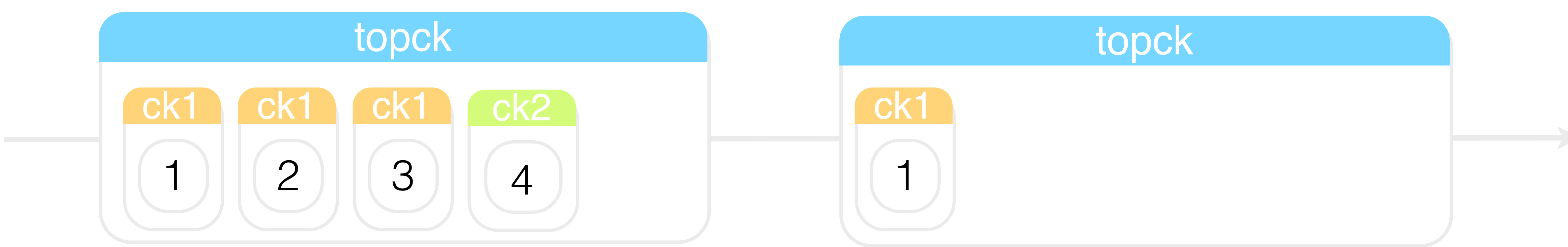
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



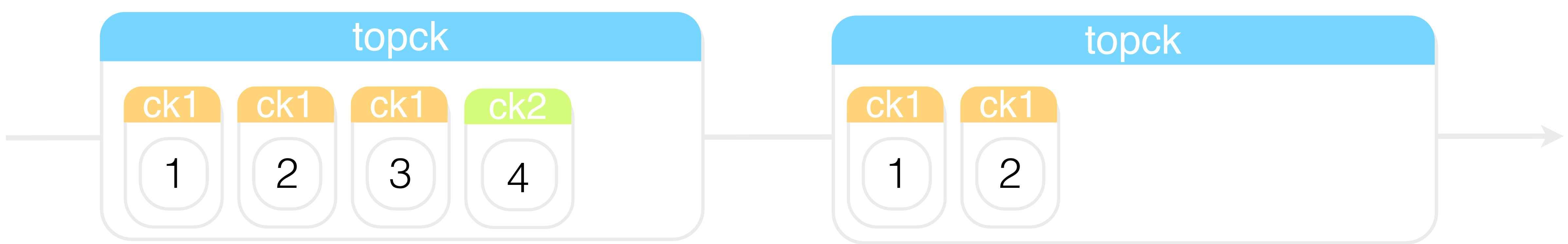
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



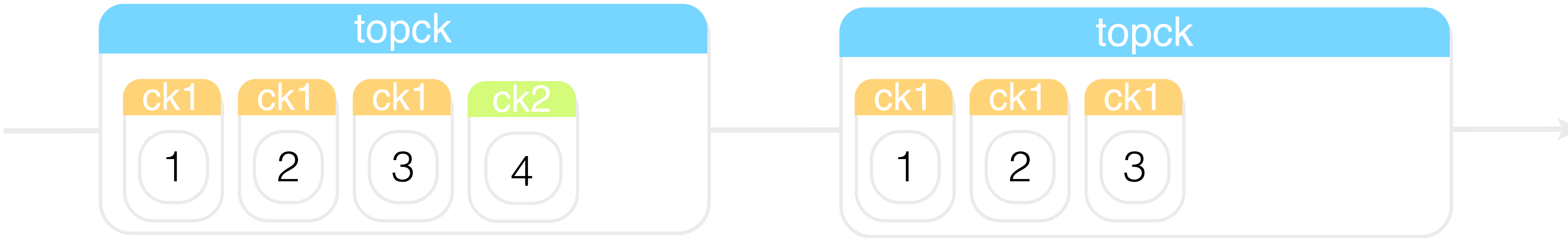
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



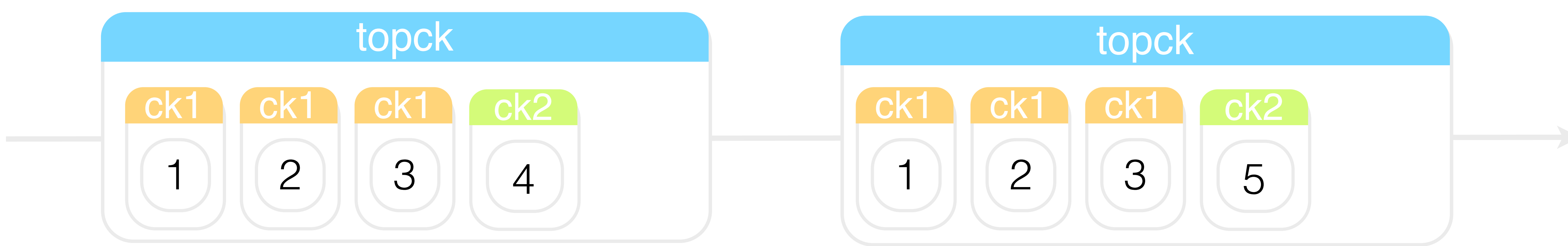
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



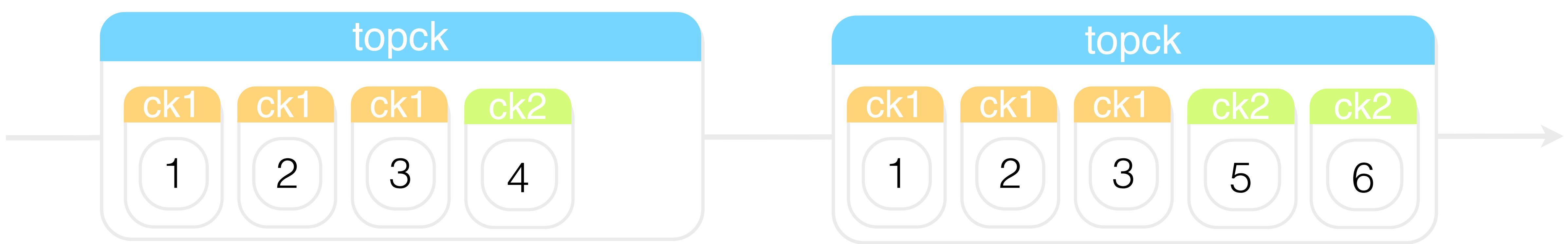
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



Some examples

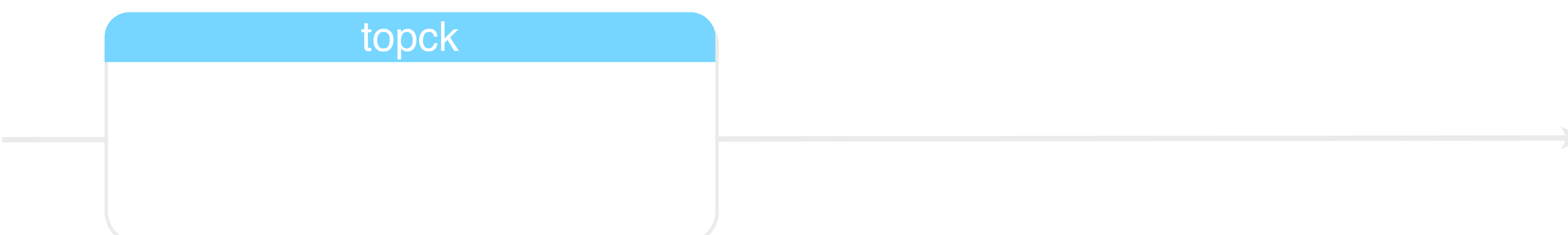
```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```

Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```

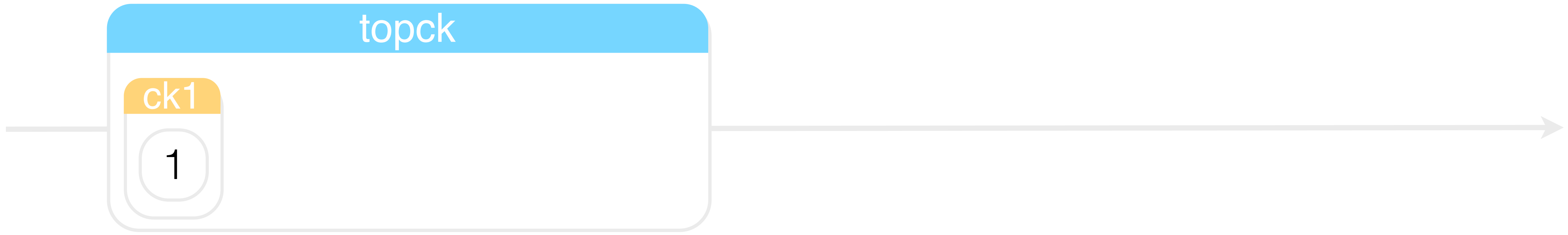
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



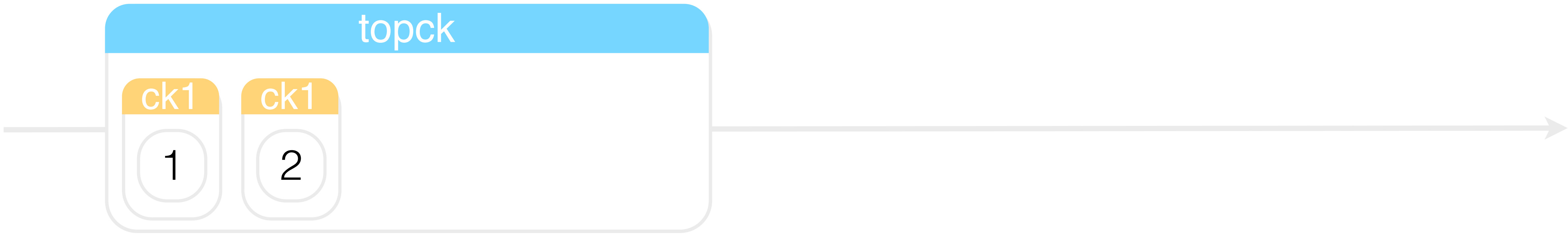
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



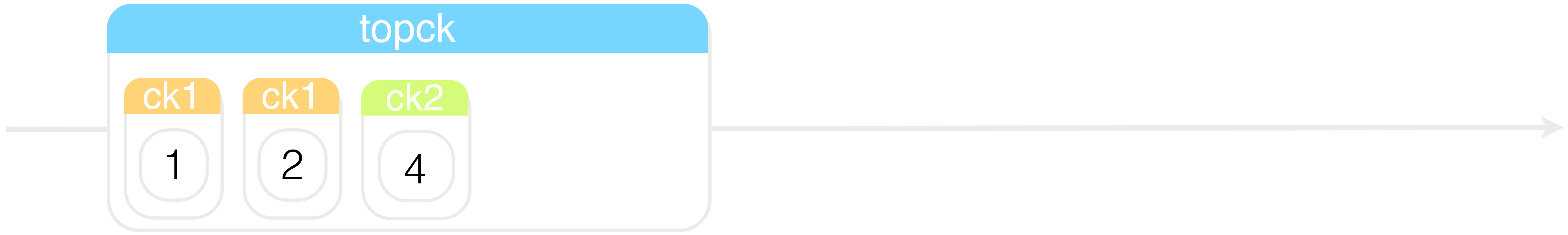
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



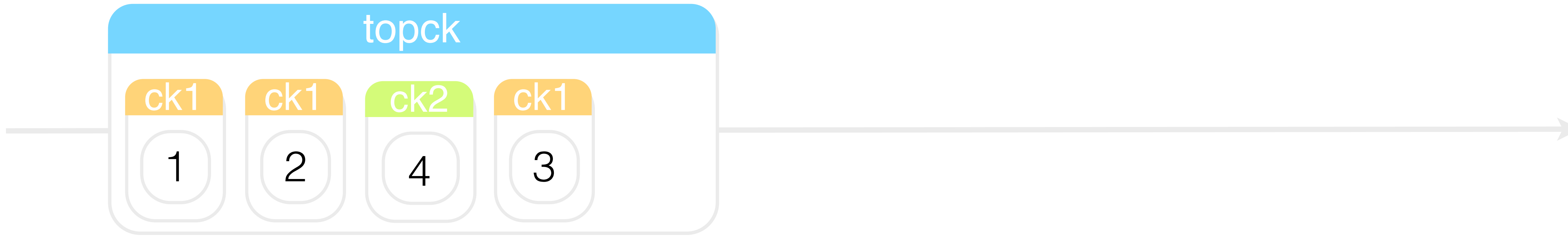
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



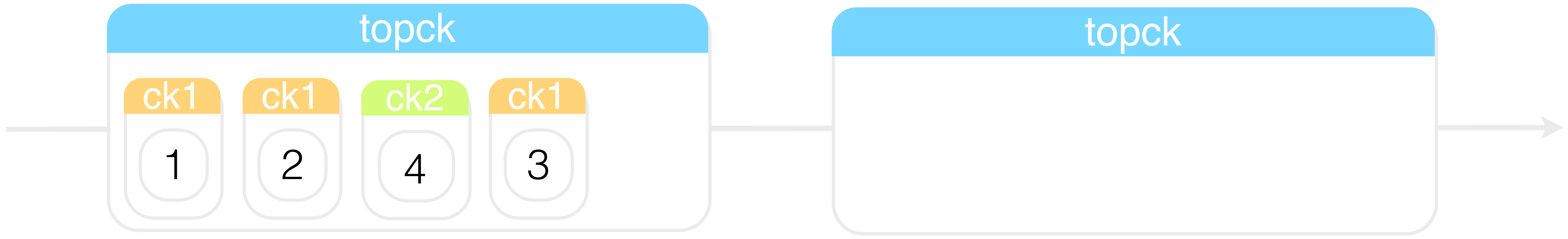
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



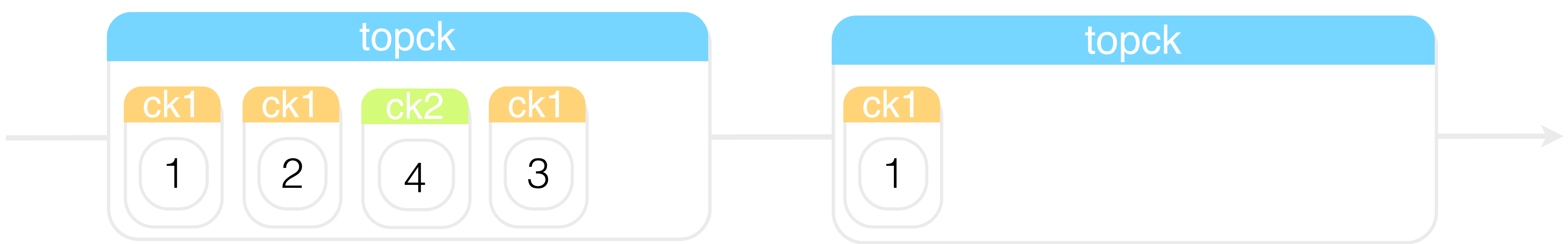
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



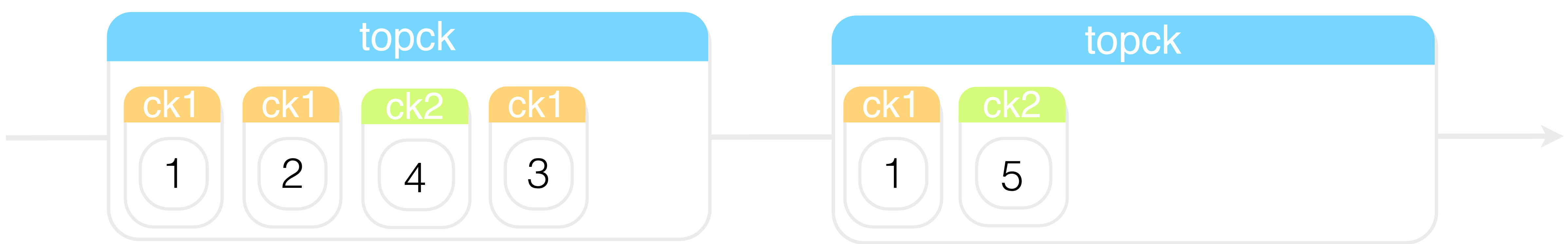
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



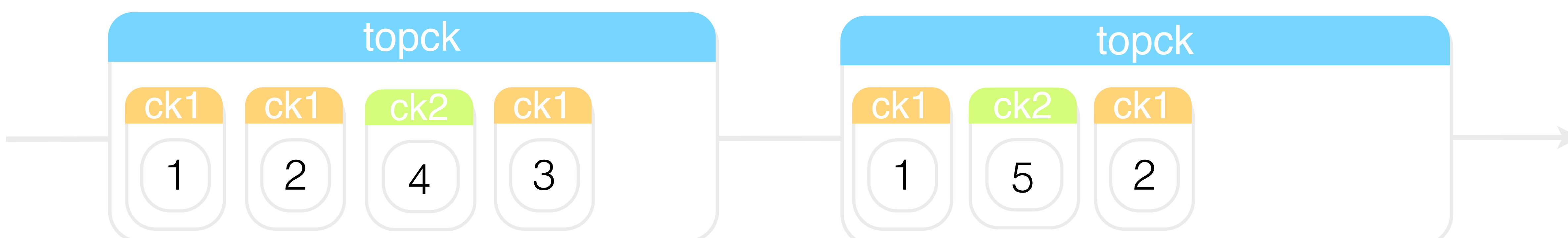
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



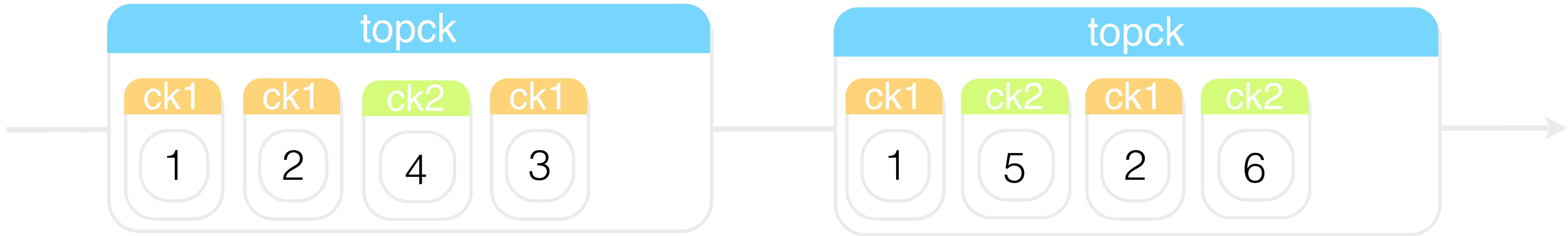
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



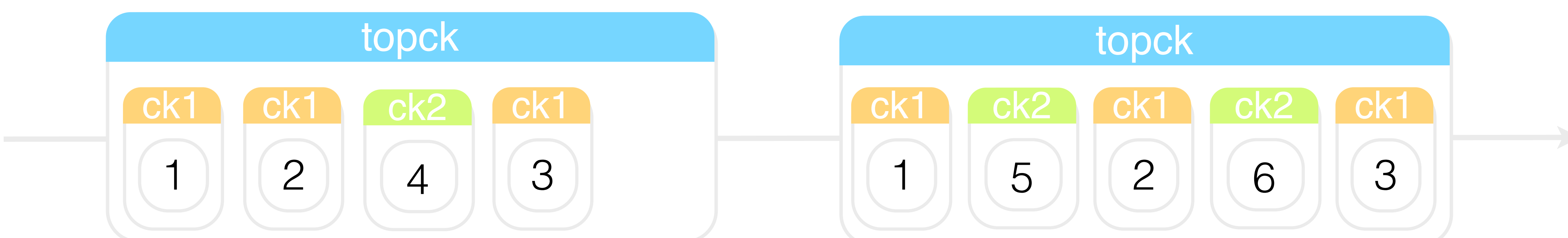
Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



Some examples

```
let process p s_out =
  newclock ck1 in
    loop
      emit s_out 1; pause ck1; emit s_out 2; pause ck1;
      emit s_out 3; pause topck
    end
  ||
  newclock ck2 in
    loop
      emit s_out 4; pause topck; emit s_out 5; pause ck2;
      emit s_out 6; pause topck
    end
```



Some examples

```
let process f compute s_in s_out =
 newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
  end
```

```
f : (clock -> 'a -> 'b process) ->
    ('c, 'a) event -> ('b, 'd) event -> unit process
```

- Same external behaviour no matter how long compute lasts

Some examples

```
let process f compute s_in s_out =
 newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
  end
```

```
f : (clock -> 'a -> 'b process) ->
    ('c, 'a) event -> ('b, 'd) event -> unit process
```

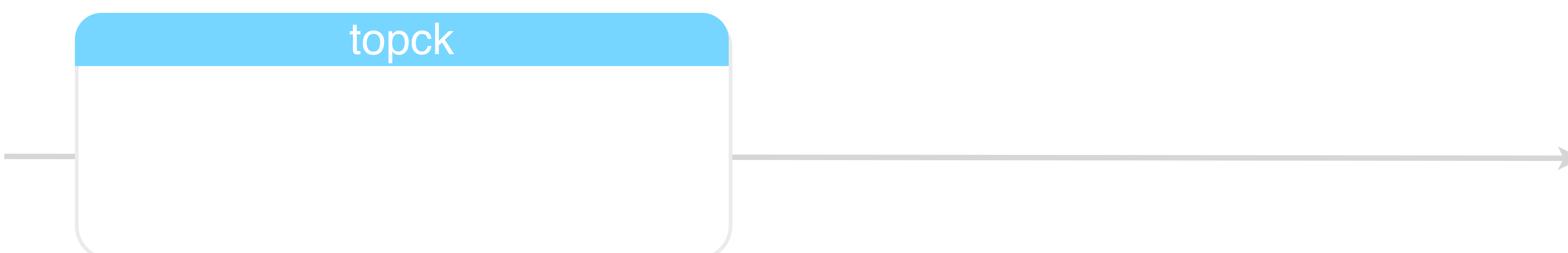
- Same external behaviour no matter how long compute lasts

Some examples

```
let process f compute s_in s_out =
 newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
  end
```

```
f : (clock -> 'a -> 'b process) ->
    ('c, 'a) event -> ('b, 'd) event -> unit process
```

- Same external behaviour no matter how long compute lasts



Some examples

```
let process f compute s_in s_out =
 newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
  end
```

```
f : (clock -> 'a -> 'b process) ->
    ('c, 'a) event -> ('b, 'd) event -> unit process
```

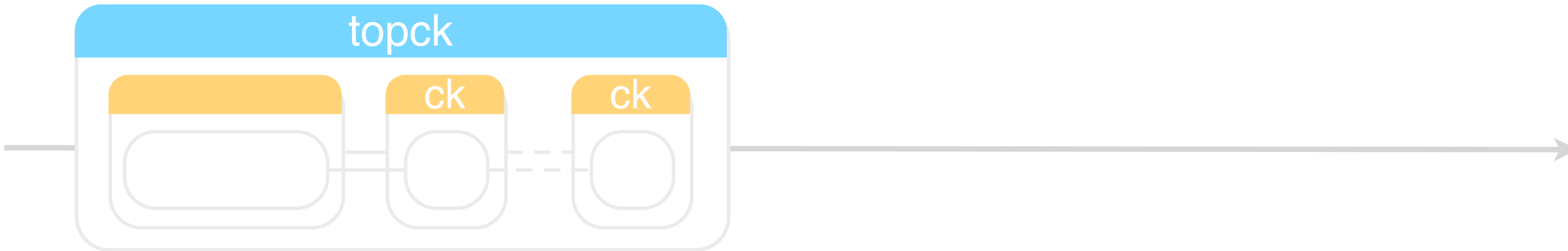
- Same external behaviour no matter how long compute lasts

Some examples

```
let process f compute s_in s_out =
 newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
  end
```

```
f : (clock -> 'a -> 'b process) ->
    ('c, 'a) event -> ('b, 'd) event -> unit process
```

- Same external behaviour no matter how long compute lasts

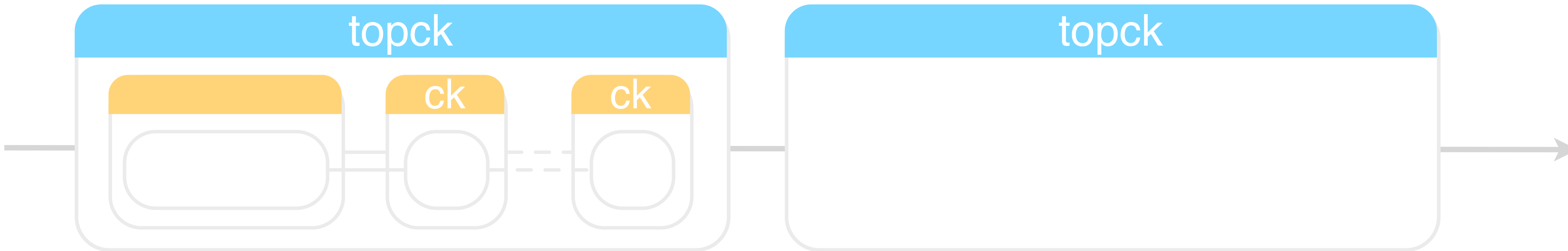


Some examples

```
let process f compute s_in s_out =
 newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
  end
```

```
f : (clock -> 'a -> 'b process) ->
    ('c, 'a) event -> ('b, 'd) event -> unit process
```

- Same external behaviour no matter how long compute lasts

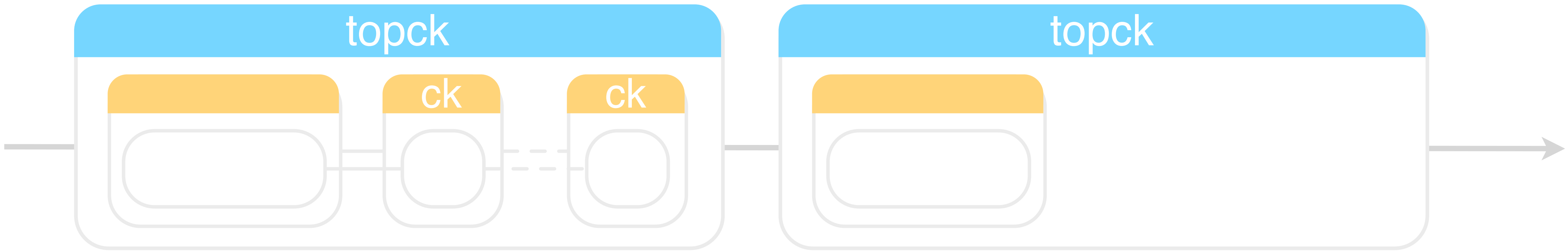


Some examples

```
let process f compute s_in s_out =
 newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
  end
```

```
f : (clock -> 'a -> 'b process) ->
    ('c, 'a) event -> ('b, 'd) event -> unit process
```

- Same external behaviour no matter how long compute lasts

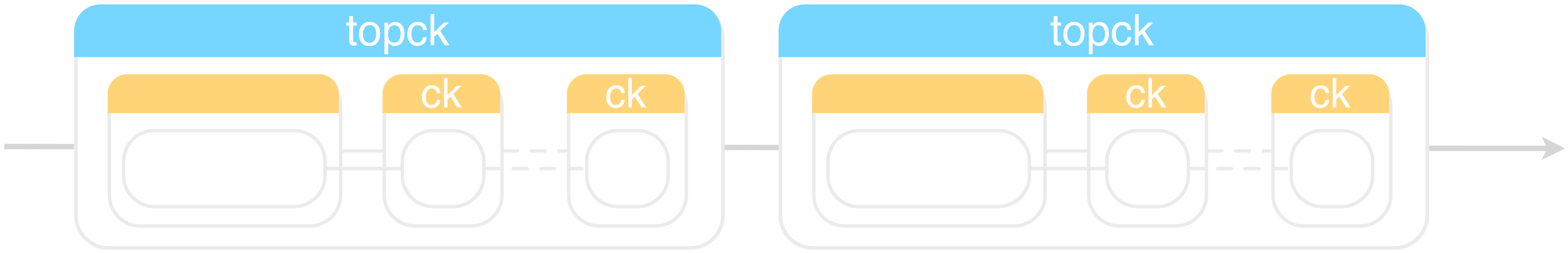


Some examples

```
let process f compute s_in s_out =
 newclock ck in
  loop
    await s_in(v) in
    let res = run (compute ck v) in
    emit s_out res
  end
```

```
f : (clock -> 'a -> 'b process) ->
    ('c, 'a) event -> ('b, 'd) event -> unit process
```

- Same external behaviour no matter how long compute lasts



Clocks and data dependencies

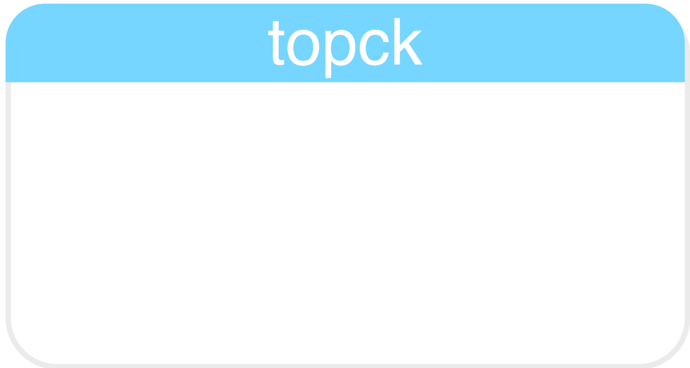
```
signal s in
newclock ck1 in (
   await immediate s;
   print_int 1
   ||
   pause ck1; pause ck1;
   print_int 0
)
||
newclock ck2 in
   emit s
```

Clocks and data dependencies

```
signal s in
newclock ck1 in (
  await immediate s;
  print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```

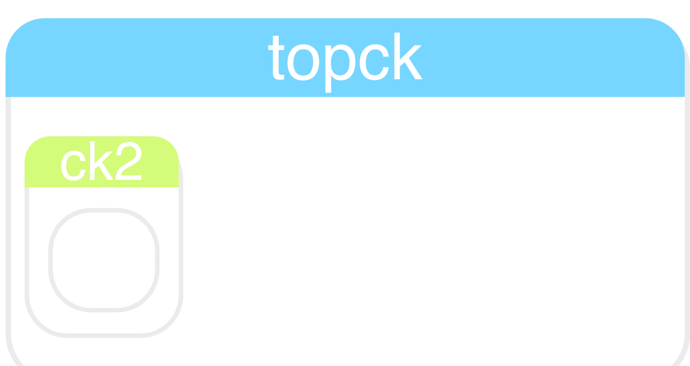
Clocks and data dependencies

```
signal s in
newclock ck1 in (
   await immediate s;
   print_int 1
   ||
   pause ck1; pause ck1;
   print_int 0
)
||
newclock ck2 in
   emit s
```



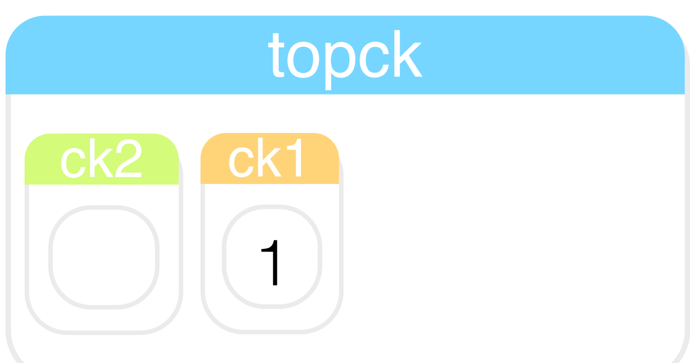
Clocks and data dependencies

```
signal s in
newclock ck1 in (
   await immediate s;
   print_int 1
   ||
   pause ck1; pause ck1;
   print_int 0
)
||
newclock ck2 in
   emit s
```



Clocks and data dependencies

```
signal s in
newclock ck1 in (
   await immediate s;
   print_int 1
   ||
   pause ck1; pause ck1;
   print_int 0
)
||
newclock ck2 in
   emit s
```



Clocks and data dependencies

```
signal s in
newclock ck1 in (
   await immediate s;
   print_int 1
   ||
   pause ck1; pause ck1;
   print_int 0
)
||
newclock ck2 in
   emit s
```

Clocks and data dependencies

```
signal s in
newclock ck1 in (
   await immediate s;
   print_int 1
   ||
   pause ck1; pause ck1;
   print_int 0
)
||
newclock ck2 in
   emit s
```

Clocks and data dependencies

```
signal s in
newclock ck1 in (
   await immediate s;
   print_int 1
   ||
   pause ck1; pause ck1;
   print_int 0
)
||
newclock ck2 in
   emit s
```

Clocks and data dependencies

```
signal s in
newclock ck1 in (
   await immediate s;
   print_int 1
   ||
   pause ck1; pause ck1;
   print_int 0
)
||
newclock ck2 in
   emit s
```

Clocks and data dependencies

```
signal s in
newclock ck1 in (
  await immediate s;
  print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```

Clocks and data dependencies

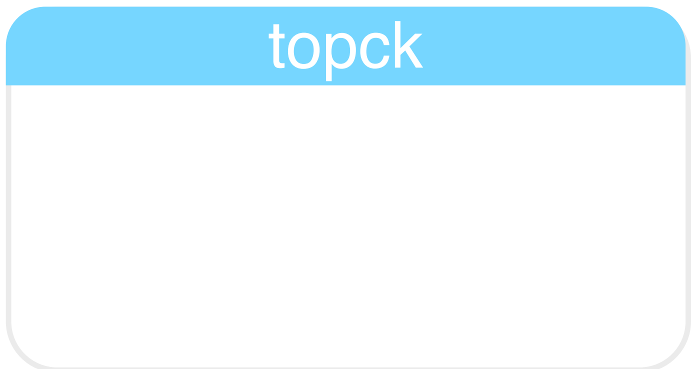
```
signal s in
newclock ck1 in (
  await immediate s;
  print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```

```
signal s in
newclock ck1 in (
  await s; print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```

Clocks and data dependencies

```
signal s in
newclock ck1 in (
  await immediate s;
  print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```

```
signal s in
newclock ck1 in (
  await s; print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```



signal

Clocks and data dependencies

```
signal s in
newclock ck1 in (
  await immediate s;
  print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```

```
signal s in
newclock ck1 in (
  await s; print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```

Clocks and data dependencies

```
signal s in
newclock ck1 in (
  await immediate s;
  print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```

```
signal s in
newclock ck1 in (
  await s; print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```

Clocks and data dependencies

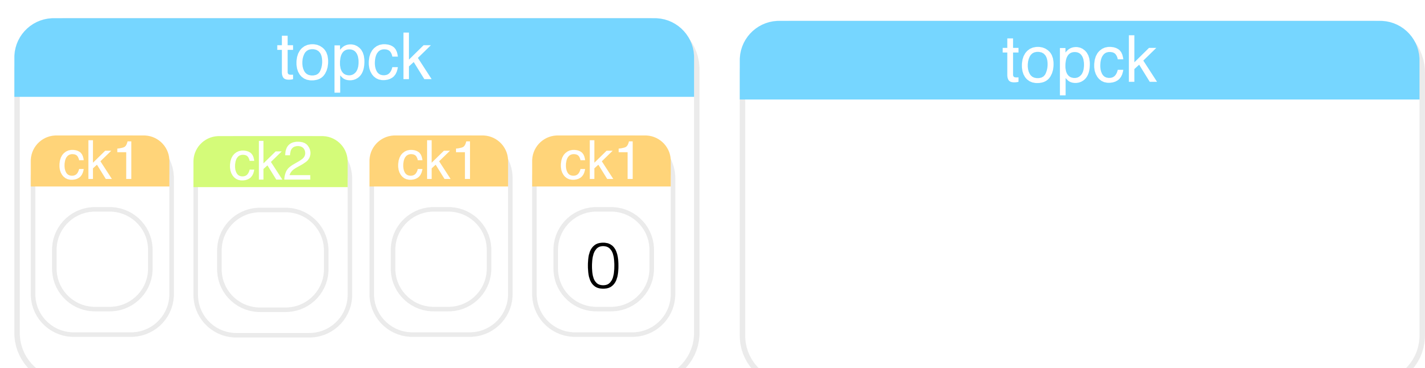
```
signal s in
newclock ck1 in (
  await immediate s;
  print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```

```
signal s in
newclock ck1 in (
  await s; print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```

Clocks and data dependencies

```
signal s in
newclock ck1 in (
  await immediate s;
  print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```

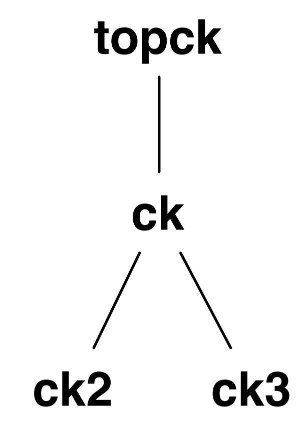
```
signal s in
newclock ck1 in (
  await s; print_int 1
  ||
  pause ck1; pause ck1;
  print_int 0
)
||
newclock ck2 in
  emit s
```



Clock domains

The clocks tree

```
newclock ck in
  newclock ck2 in p2
  ||
  newclock ck3 in p3
```



Execution of a clock domain

- Do one step of the internal clock
- If all processes are waiting on a slower clock, wait for the next instant of the parent clock
- Otherwise, do another step
- Dynamic, unbounded number of steps

An uncooperative clock domain

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
```

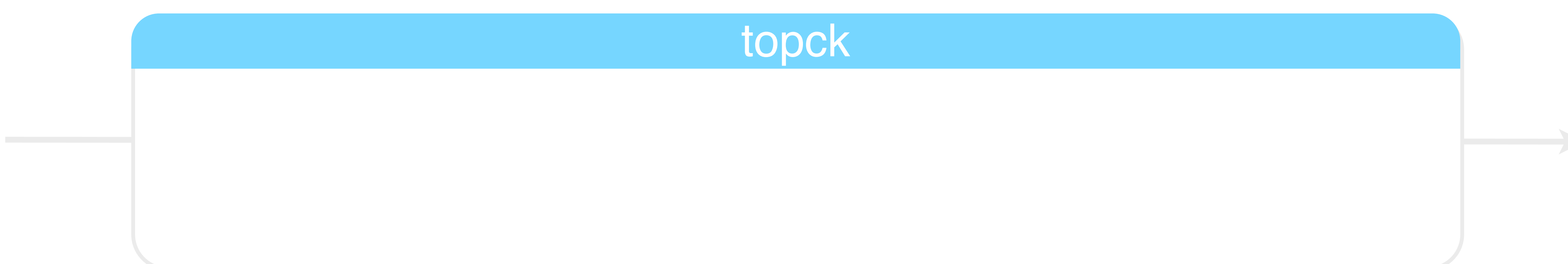
An uncooperative clock domain

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
```

Clocks and Reactivity

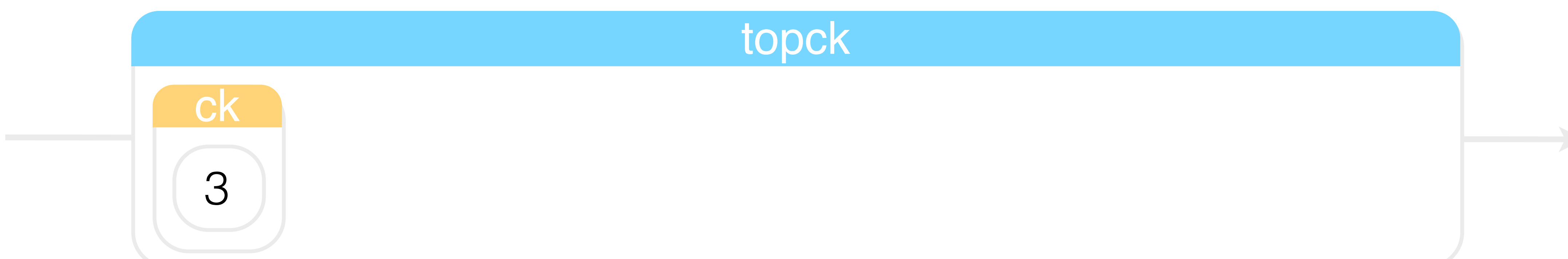
An uncooperative clock domain

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
```



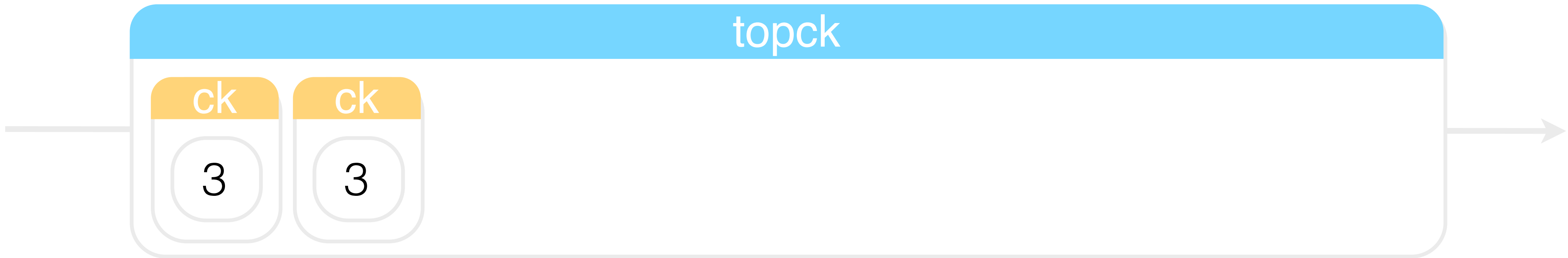
An uncooperative clock domain

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
```



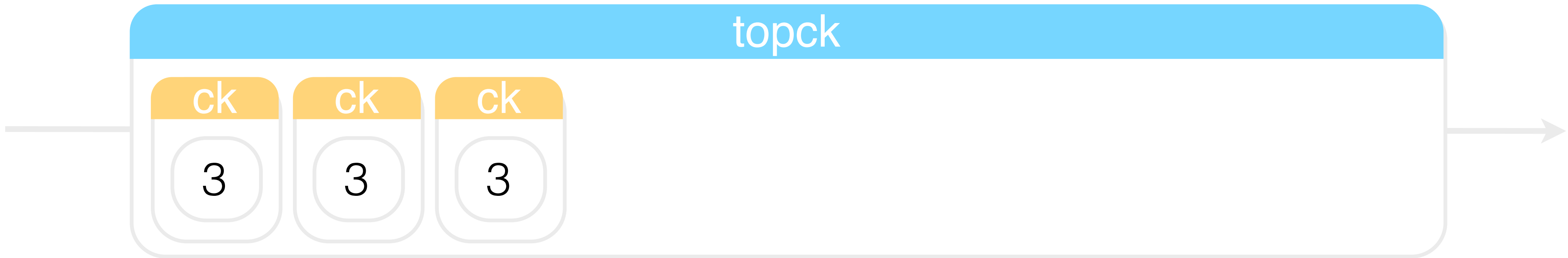
An uncooperative clock domain

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
```



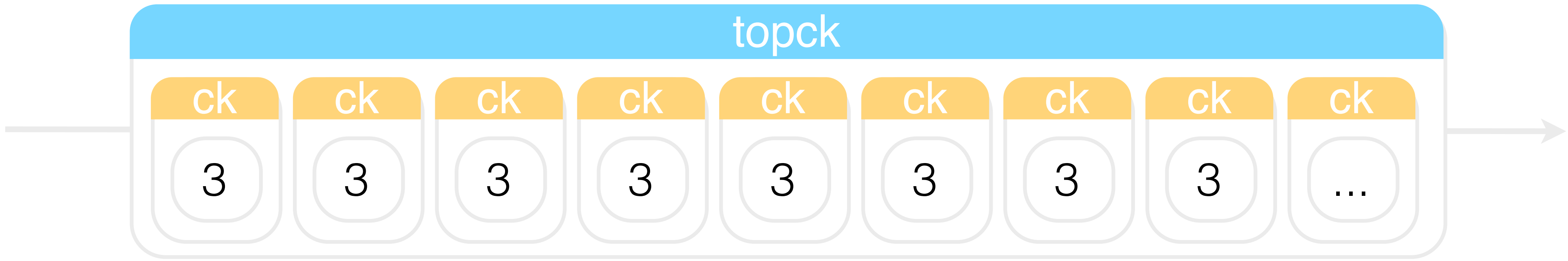
An uncooperative clock domain

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
```



An uncooperative clock domain

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
```



Pausing a clock domain

- **pauseclock ck** forces the clock domain ck to wait for the next instant of its parent clock

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
    ||
    loop
      pause ck; pause ck; pauseclock ck
    end
```

Pausing a clock domain

- **pauseclock ck** forces the clock domain ck to wait for the next instant of its parent clock

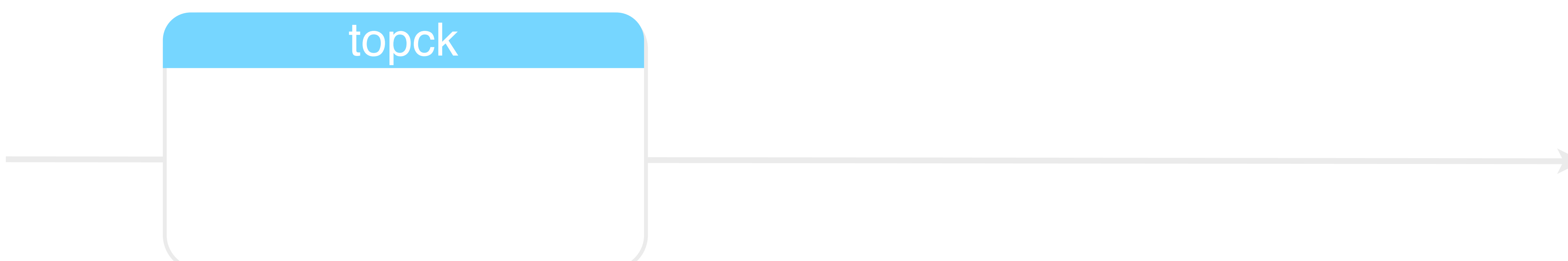
```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
    ||
    loop
      pause ck; pause ck; pauseclock ck
    end
```

Clocks and Reactivity

Pausing a clock domain

- **`pauseclock ck`** forces the clock domain ck to wait for the next instant of its parent clock

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
    ||
    loop
      pause ck; pause ck; pauseclock ck
    end
```



Pausing a clock domain

- **`pauseclock ck`** forces the clock domain ck to wait for the next instant of its parent clock

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
    ||
    loop
      pause ck; pause ck; pauseclock ck
    end
```

Pausing a clock domain

- **pauseclock ck** forces the clock domain ck to wait for the next instant of its parent clock

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
    ||
    loop
      pause ck; pause ck; pauseclock ck
    end
```

Pausing a clock domain

- **`pauseclock ck`** forces the clock domain ck to wait for the next instant of its parent clock

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
    ||
    loop
      pause ck; pause ck; pauseclock ck
    end
```

Pausing a clock domain

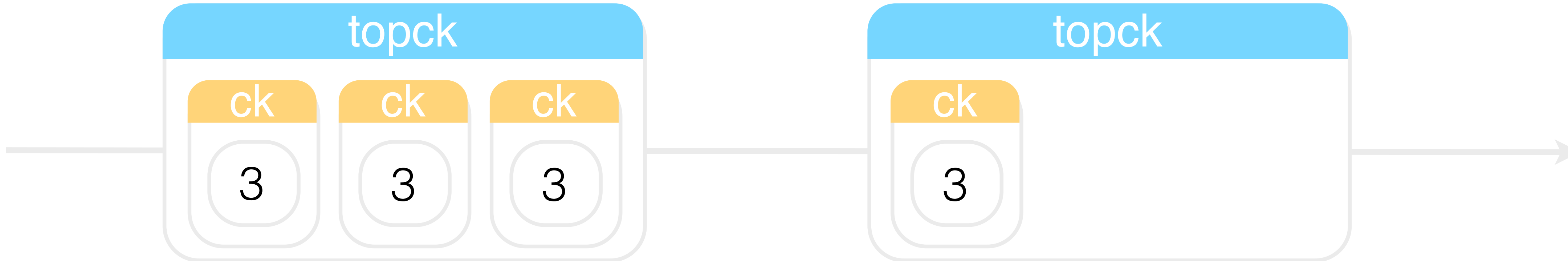
- **pauseclock ck** forces the clock domain ck to wait for the next instant of its parent clock

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
    ||
    loop
      pause ck; pause ck; pauseclock ck
    end
```

Pausing a clock domain

- **pauseclock ck** forces the clock domain ck to wait for the next instant of its parent clock

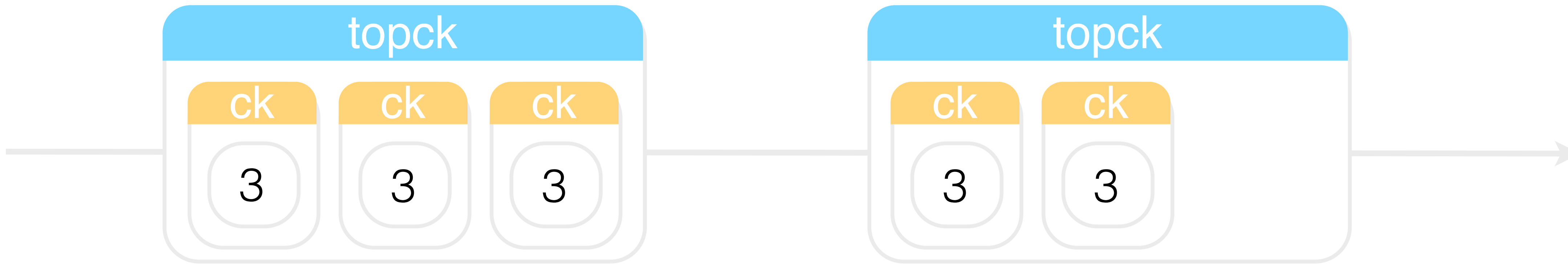
```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
    ||
    loop
      pause ck; pause ck; pauseclock ck
    end
```



Pausing a clock domain

- **pauseclock ck** forces the clock domain ck to wait for the next instant of its parent clock

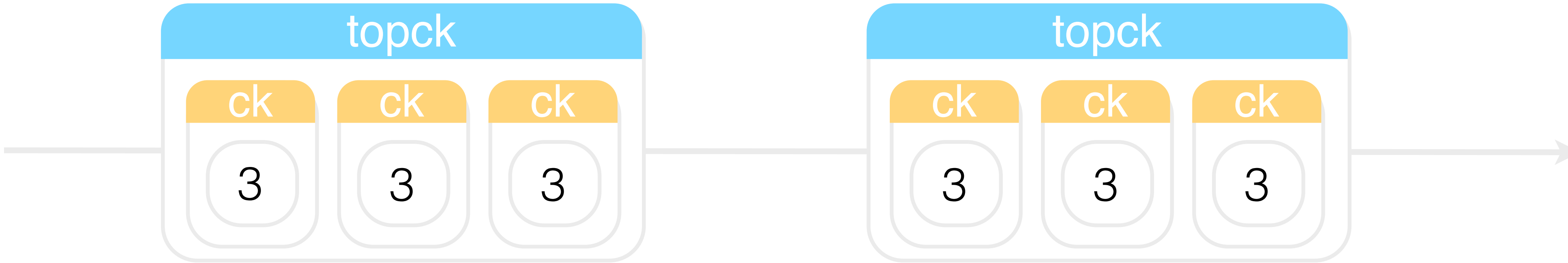
```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
    ||
    loop
      pause ck; pause ck; pauseclock ck
    end
```



Pausing a clock domain

- **`pauseclock ck`** forces the clock domain ck to wait for the next instant of its parent clock

```
let process p s_out =
  newclock ck in
    loop
      emit s_out 3; pause ck
    end
    ||
    loop
      pause ck; pause ck; pauseclock ck
    end
```



Clocks and Signals

A signal has a clock

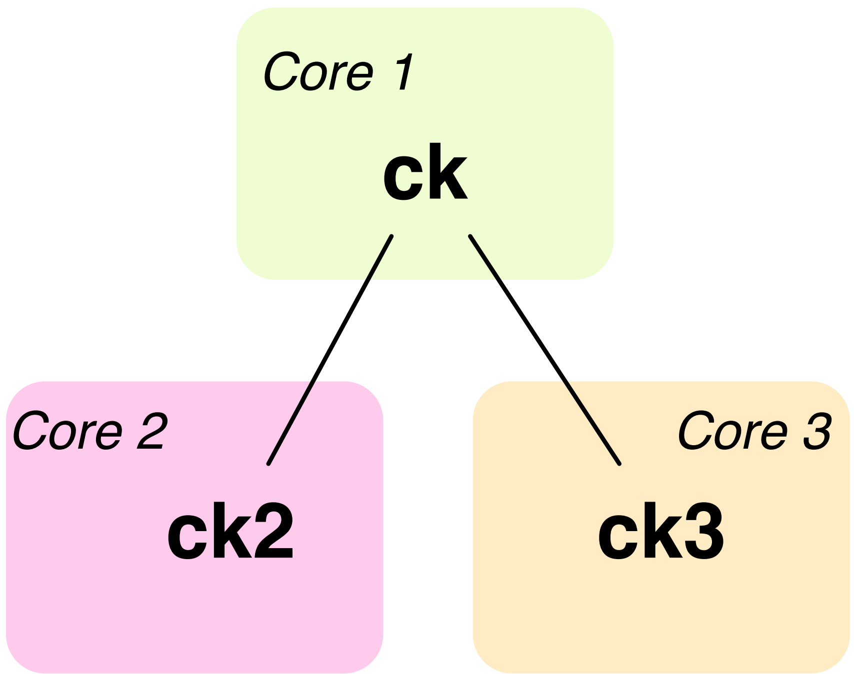
- Syntax: `signal s at ck in e`
- It cannot be used on a slower clock
- Type and effect system to prevent the use of a signal outside of its clock domain

Clocks and Parallelism

Parallelization is done at the level of clock domains

- Less synchronizations
- Locality of signals

```
newclock ck in
  newclock ck2 in p2
  ||
  newclock ck3 in p3
```



In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((
```

In a data-flow setting

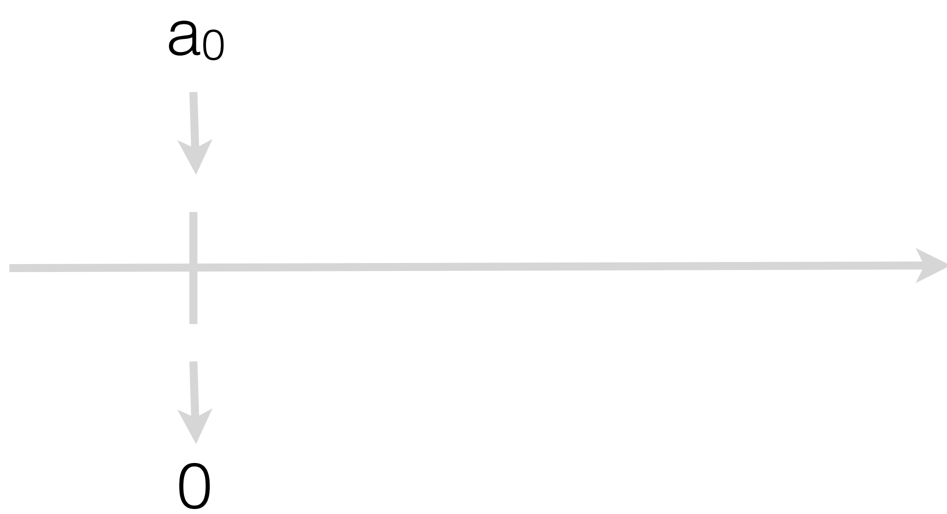
```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```

In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```

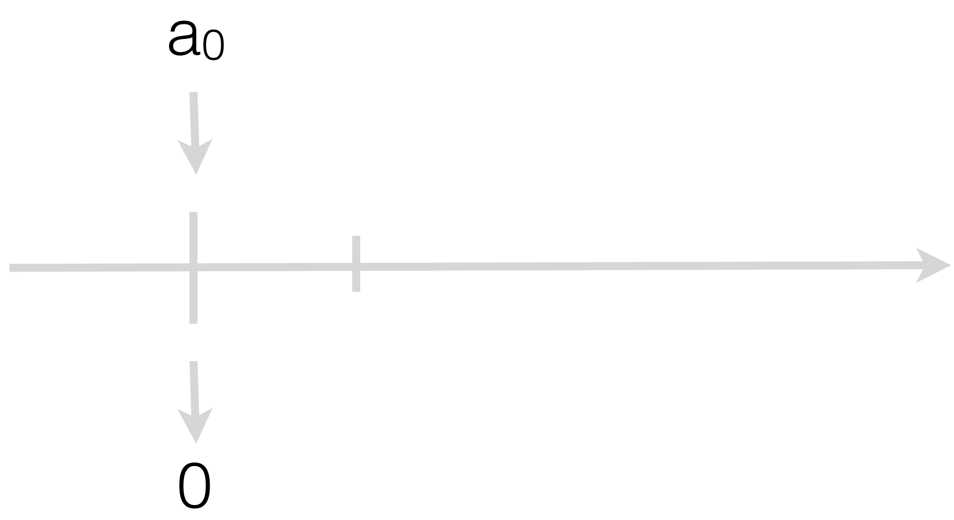
In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```



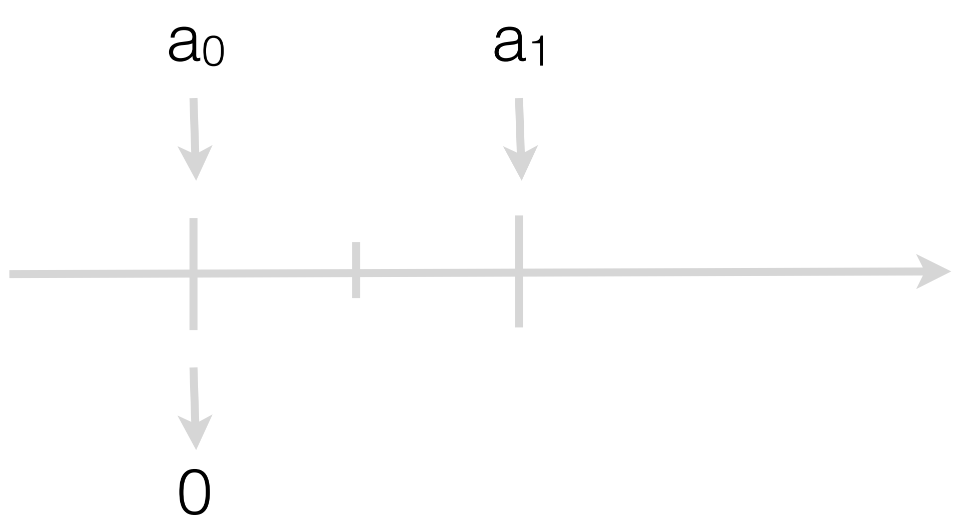
In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```



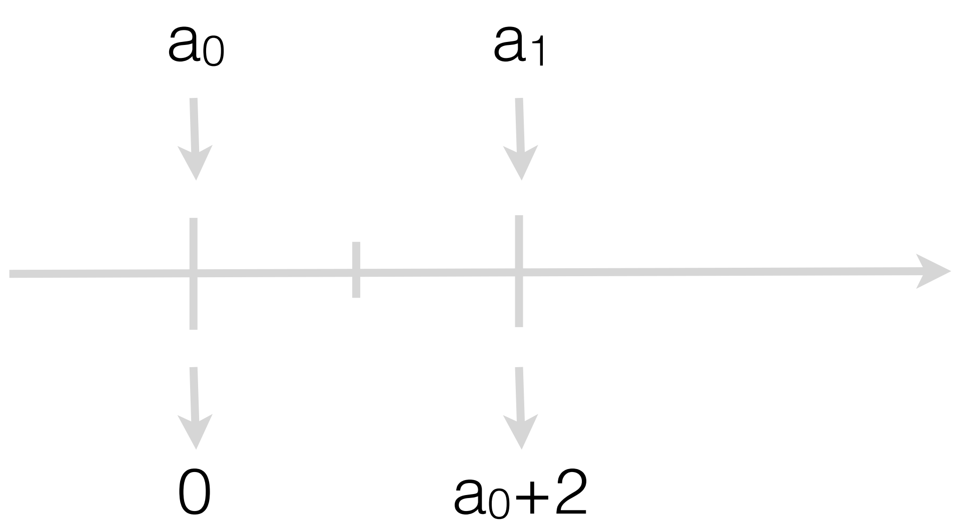
In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```



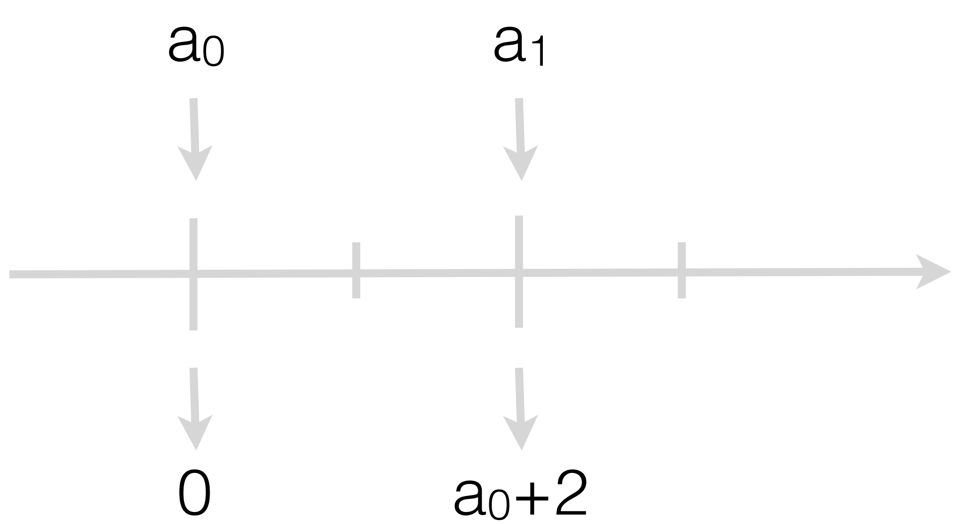
In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```



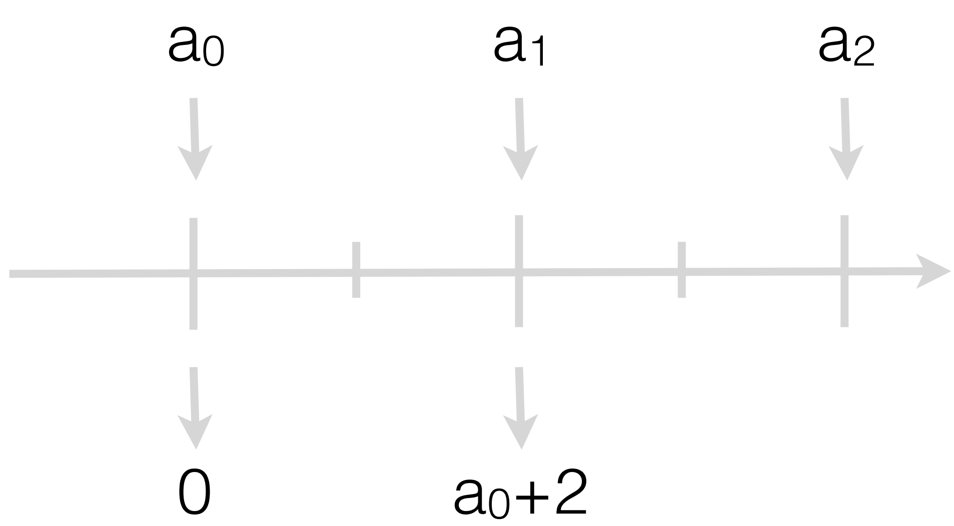
In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```



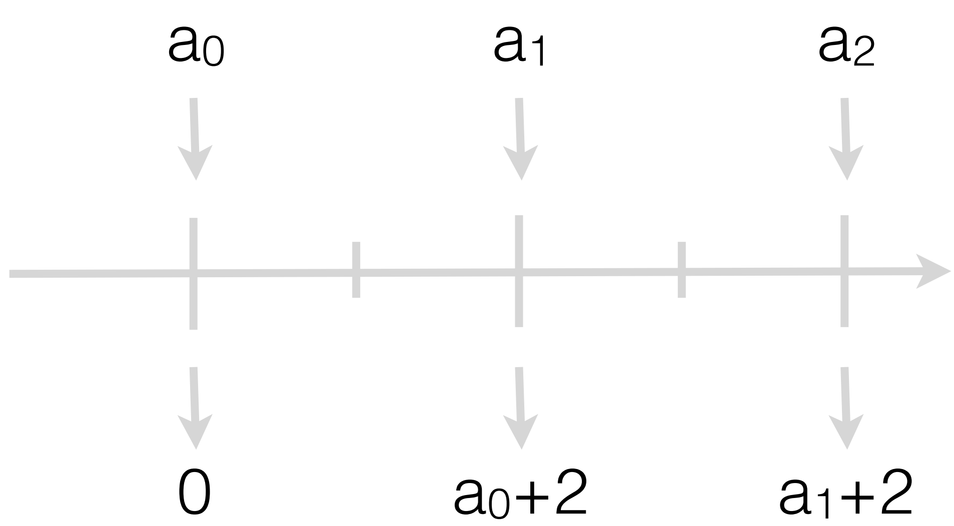
In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```



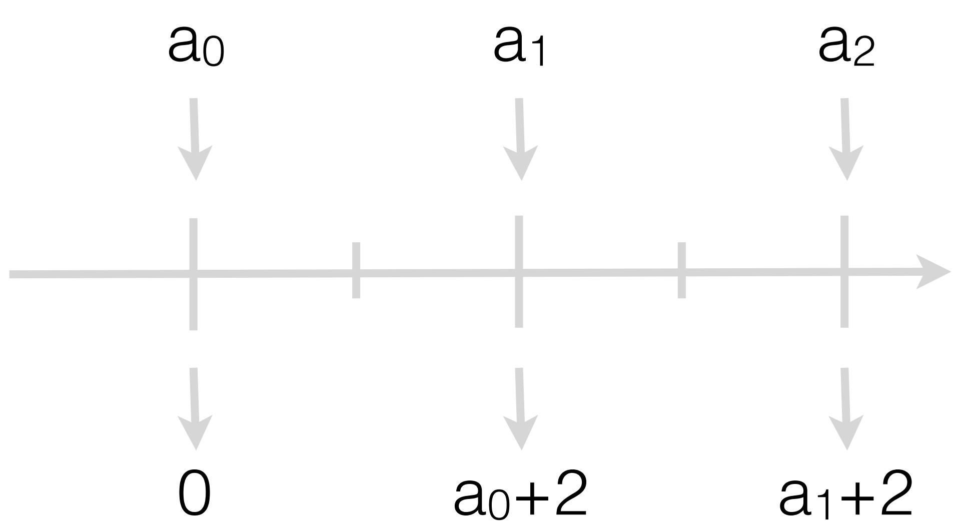
In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```



In a data-flow setting

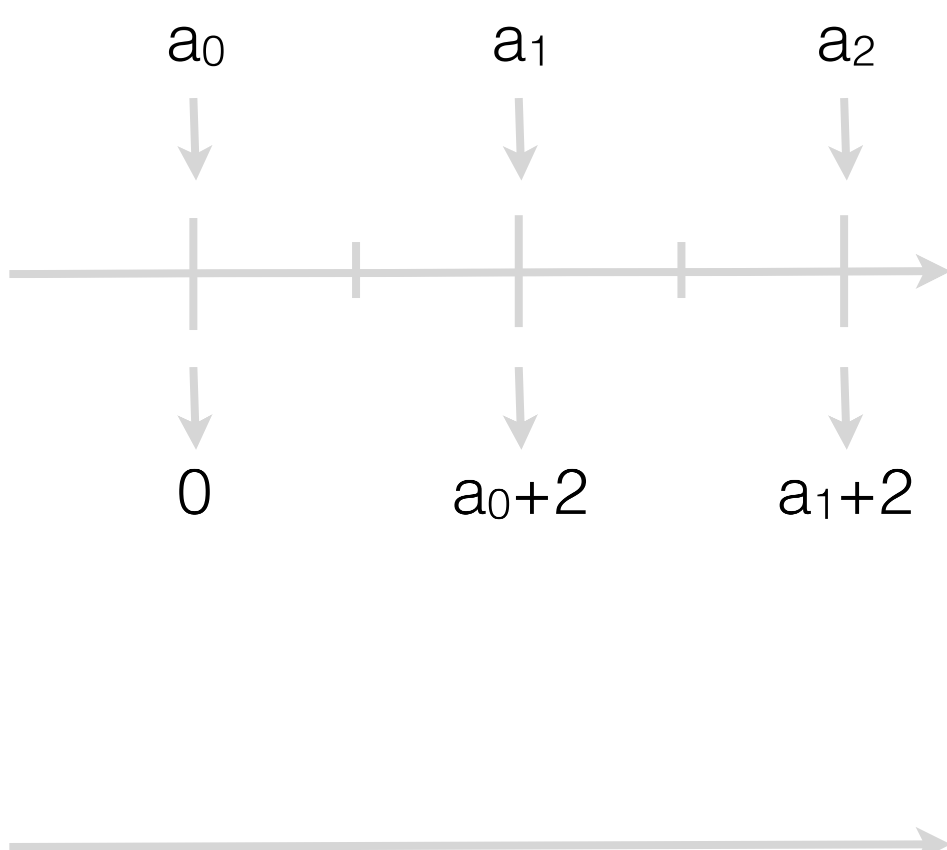
```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```



$$f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c$$

In a data-flow setting

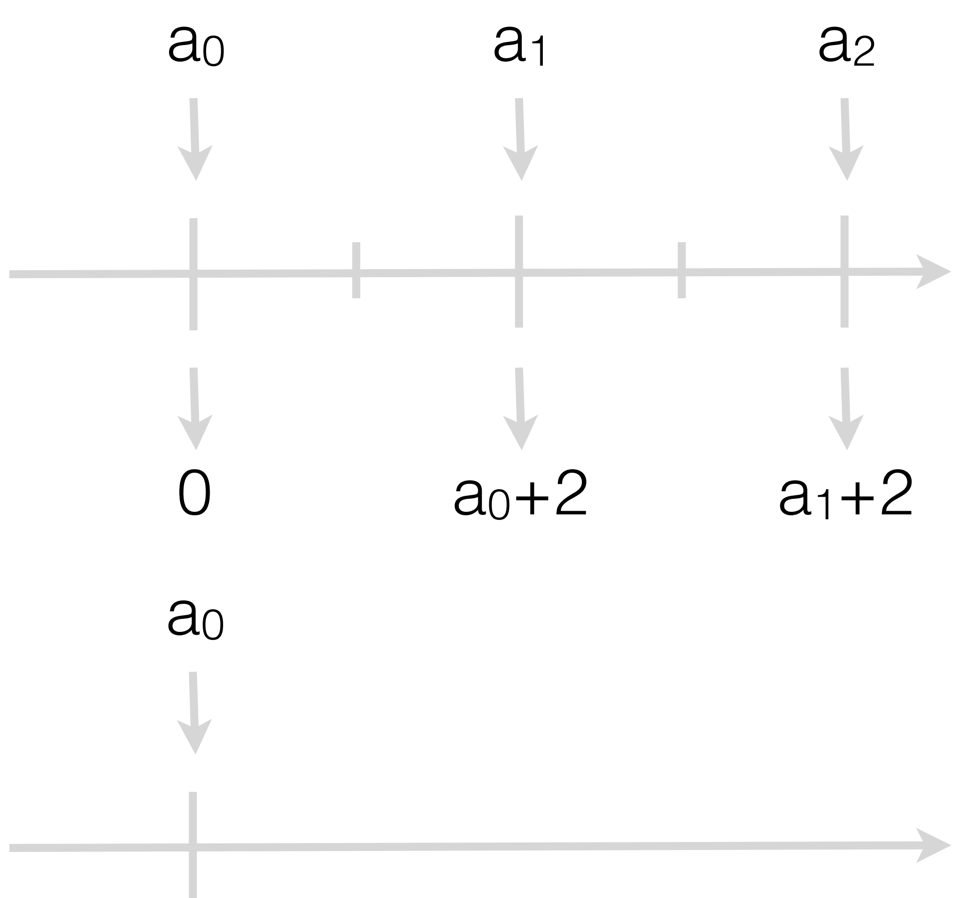
```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```



$$f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c$$

In a data-flow setting

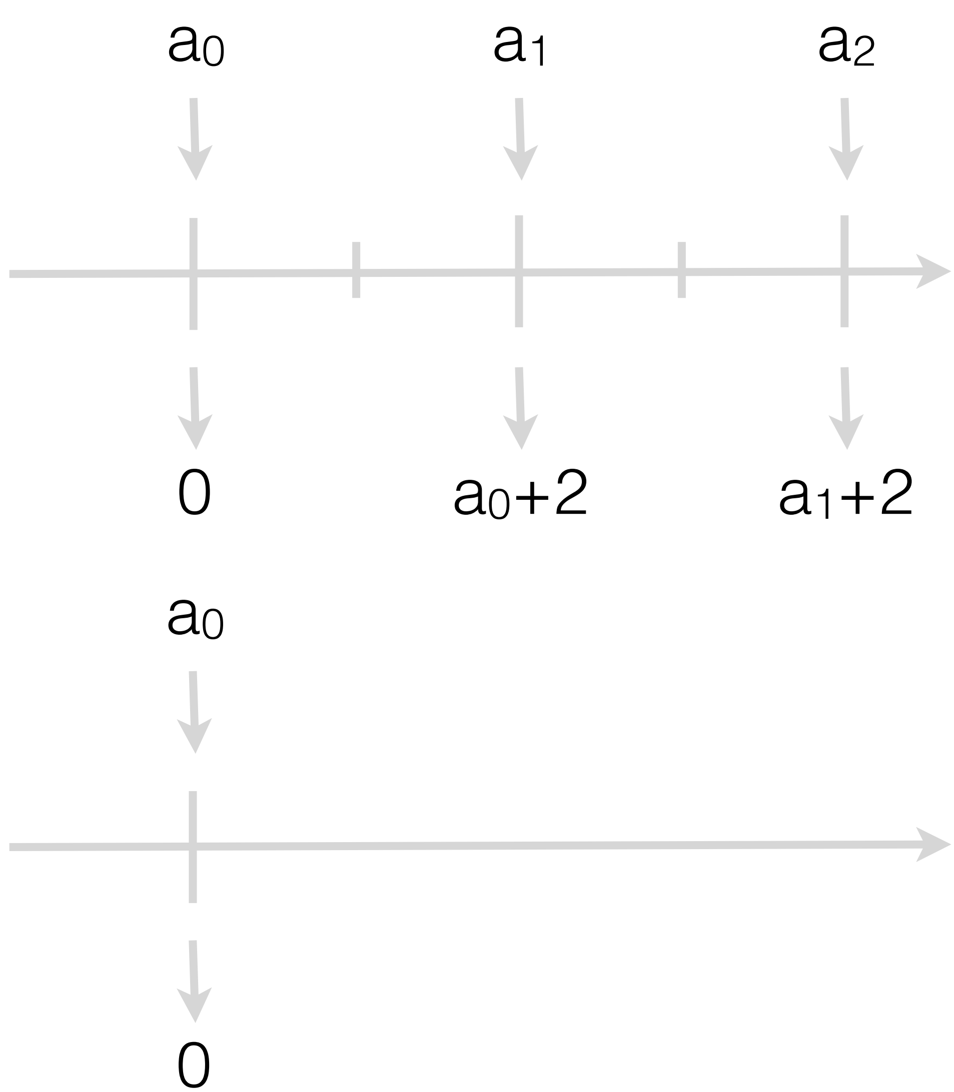
```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```



$$f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c$$

In a data-flow setting

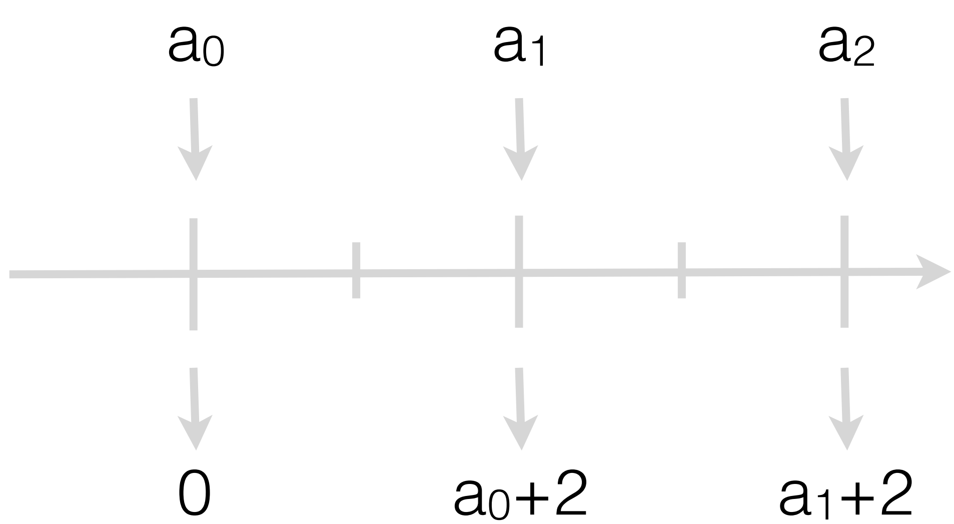
```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```



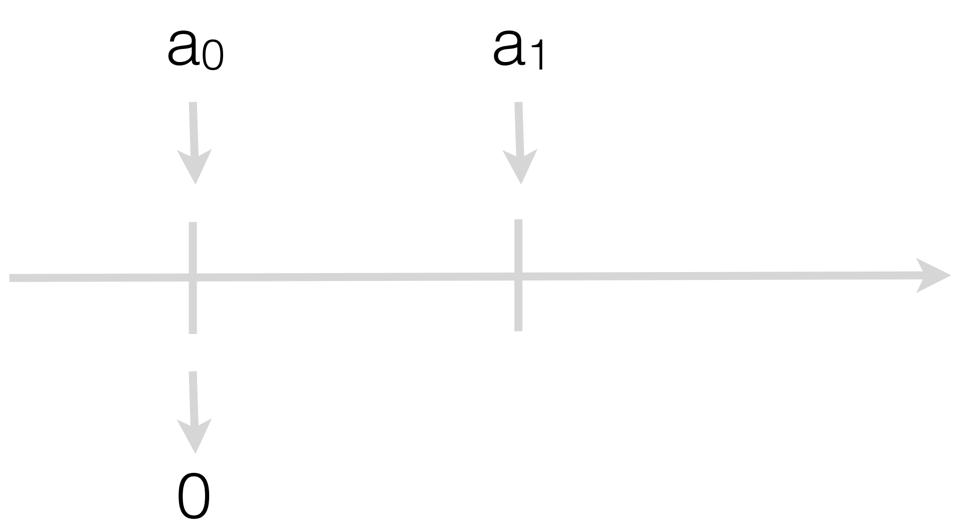
$$f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c$$

In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```

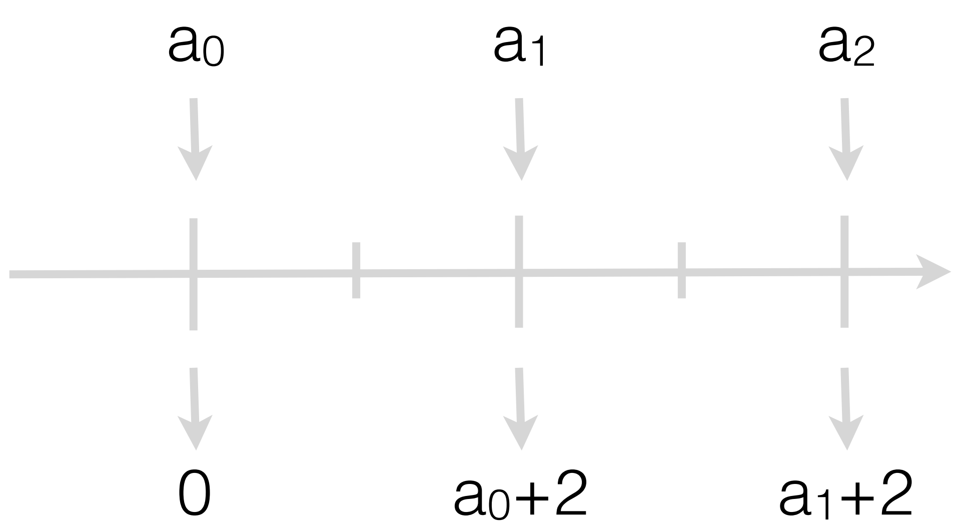


$$f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c$$

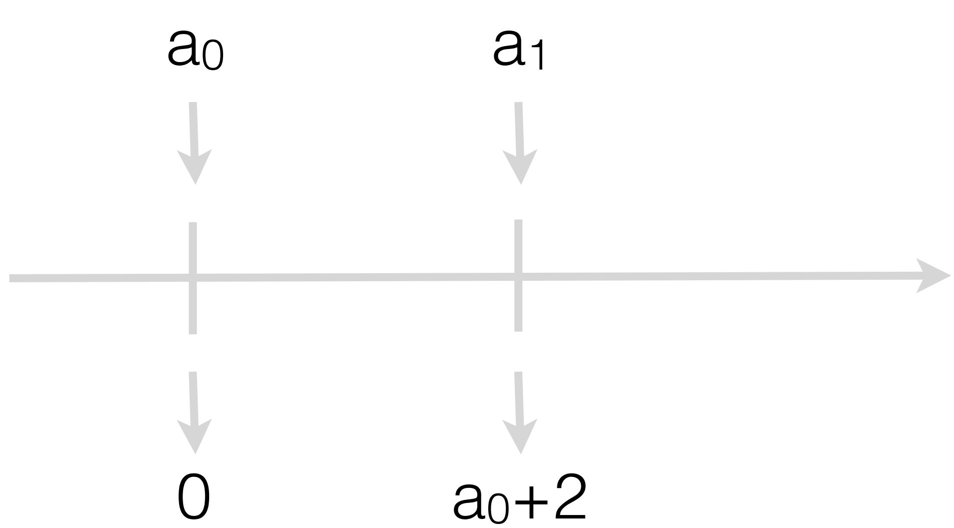


In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```

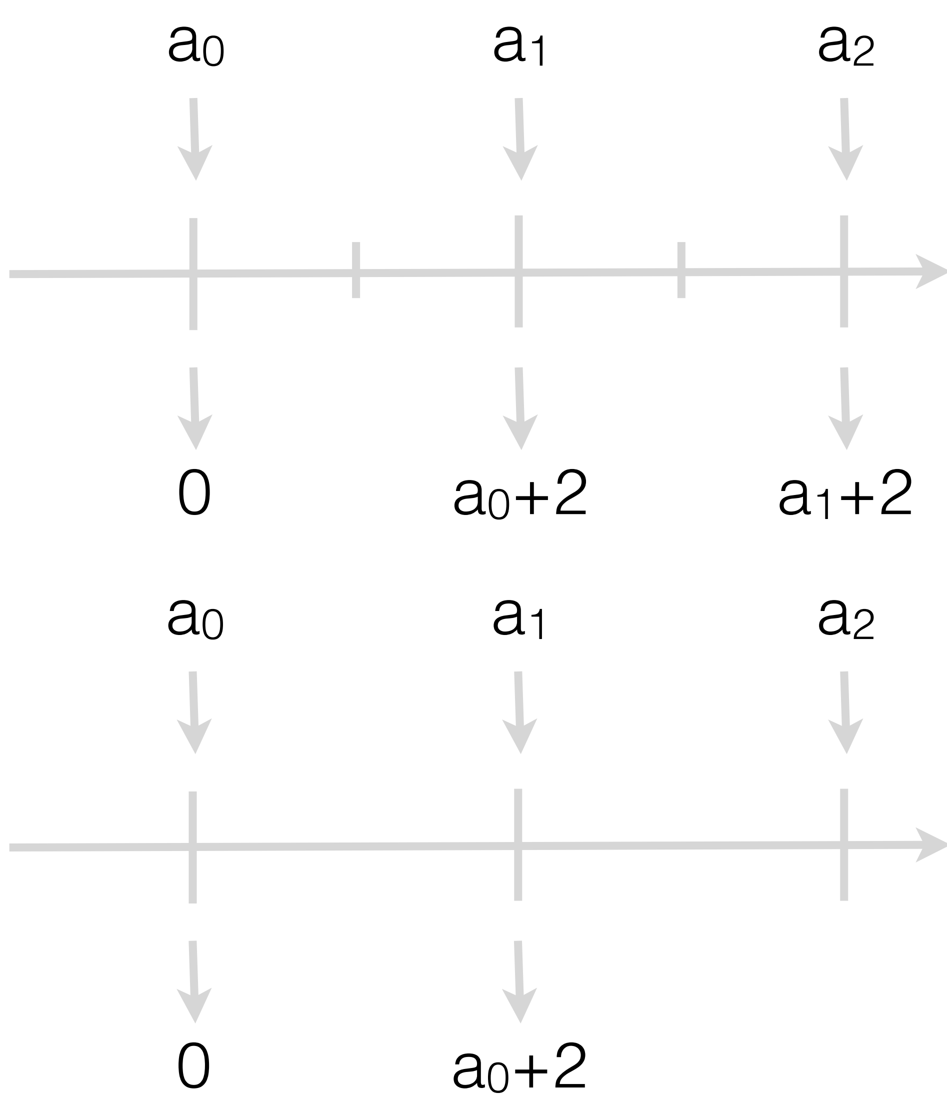


$$f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c$$



In a data-flow setting

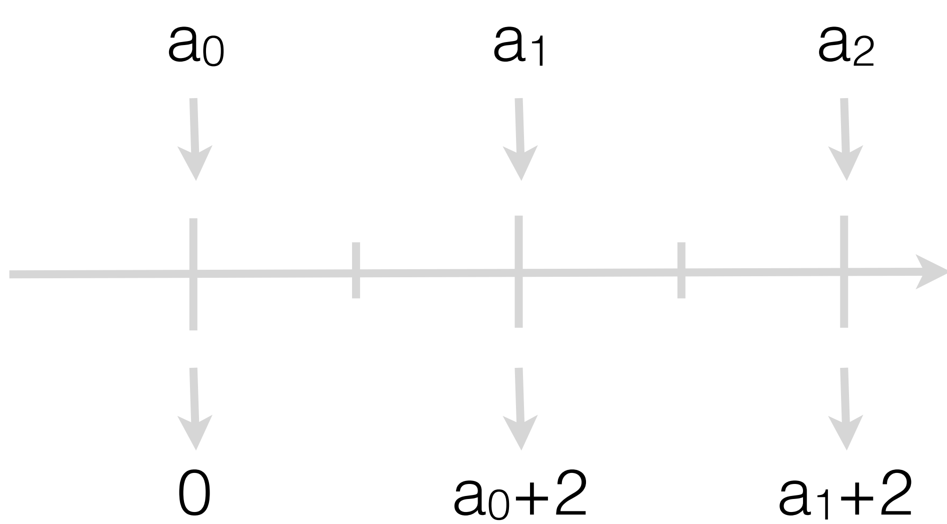
```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```



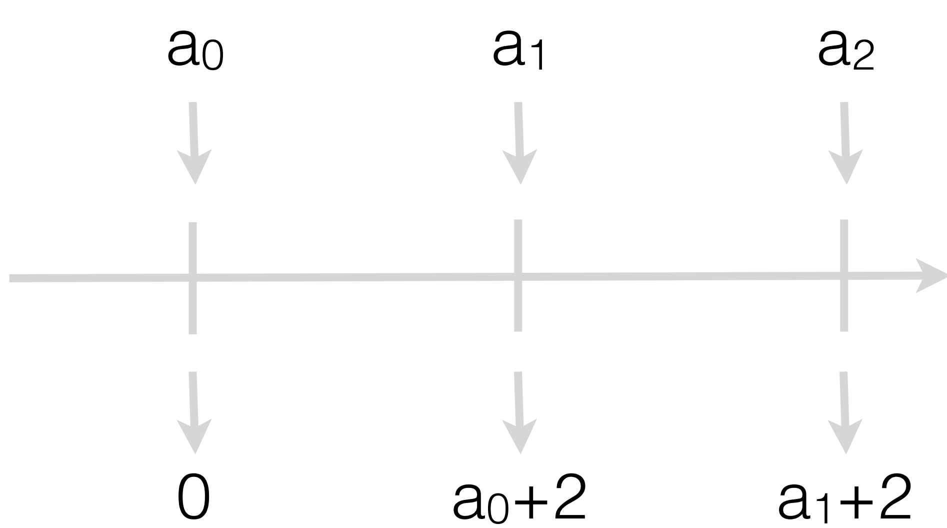
$$f :: \alpha \ \mathtt{on}\ c \xrightarrow{\alpha} \alpha \ \mathtt{on}\ c$$

In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```

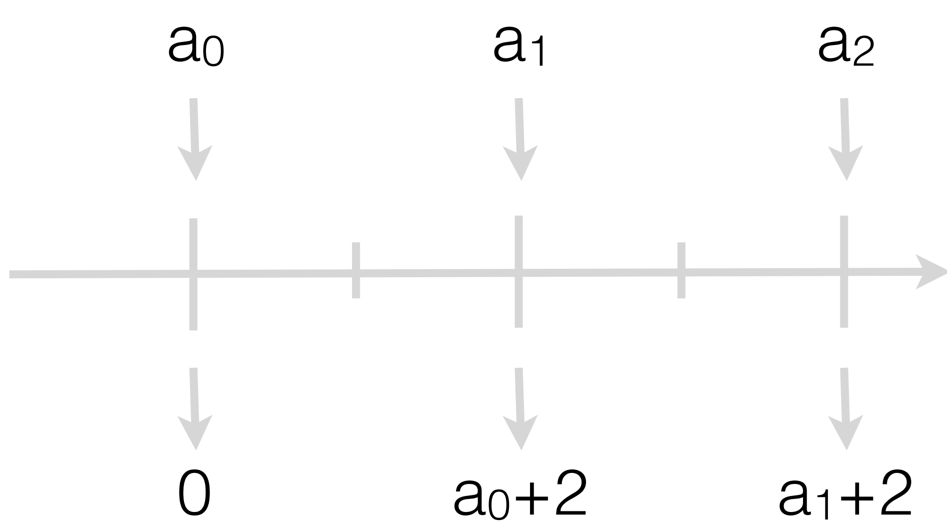


$$f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c$$

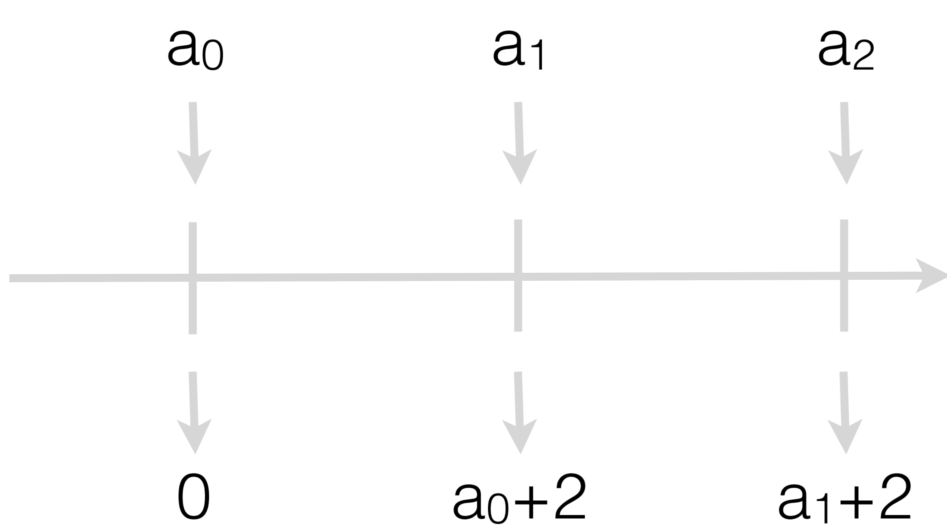


In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```



$$f :: \alpha \text{ on } c \xrightarrow{\alpha} \alpha \text{ on } c$$



$$f :: \beta \xrightarrow{\beta} \beta$$

In a data-flow setting

```
node f(a : int) = (o : int)
var x : int; c : bool;
let
  c = true fby false fby c;
  x = 0 fby ((merge c a (x whenot c)) + 1);
  o = x when c
tel
```

```
int f_step (int a) {
  do {
    ...
    if(c)
      o = ...
    ...
  } while(!c)
  return o;
}
```

Demo: n-body with multiple steps

Multiple step integration methods

- The computation of the next position is done in multiple steps, shared by all planets
- Internal steps hidden by a clock domain
- Can easily and transparently switch between methods
- eg Runge-Kutta:

$$y' = f(t, y)$$
$$y(t_0) = y_0$$

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$
$$k_1 = hf(t_n, y_n)$$
$$k_2 = hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_1)$$
$$k_3 = hf(t_n + \frac{1}{2}h, y_n + \frac{1}{2}k_2)$$
$$k_4 = hf(t_n + h, y_n + k_3)$$

Demo: n-body with multiple steps

```
let process compute_a env x w =
  emit env (force (x, w));
  await env(f) in
  f x

let process compute_rk4 env st =
  (* step 1 *)
  let k1_v = run (compute_a env st.b_pos st.b_weight) in
  let k1_p = st.b_vel in
  (* step 2 *)
  let k2_p = add_v st.b_vel (sc_mult (dt /. 2.0) k1_v) in
  let x_2 = add_v st.b_pos (sc_mult (dt /. 2.0) k1_p) in
  let k2_v = run (compute_a env x_2 st.b_weight) in
....
```

Demo: n-body adaptive

Adaptive integration

- Compute in multiple steps
  - The new positions
  - An estimation of the error
- If the error is too big, try again with a smaller step

Implementation

- Two nested clock domains
- Still one step on the top clock

Current and Future Work

Done so far

- Operational semantics
- Proof of concept of type system
- Sequential implementation

Future work

- Distributed implementation (message passing)

Conclusion

New notion of clock domain

- Improves modularity
- Helps parallelization
- Little changes to the language and runtime