A Programming Language for Interaction Nets

Termgraph

Marc Thatcher

University of Sussex

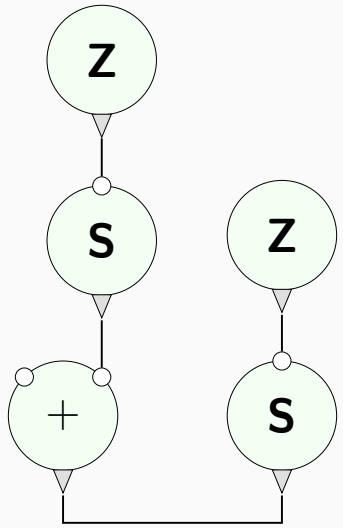
Lisbon, 19th July 2026

Interaction nets

"We propose a new kind of programming language. . . "

— Yves Lafont (POPL90)

Interaction net = finite port graph ; nodes (*agent*) + edges (*wires*).

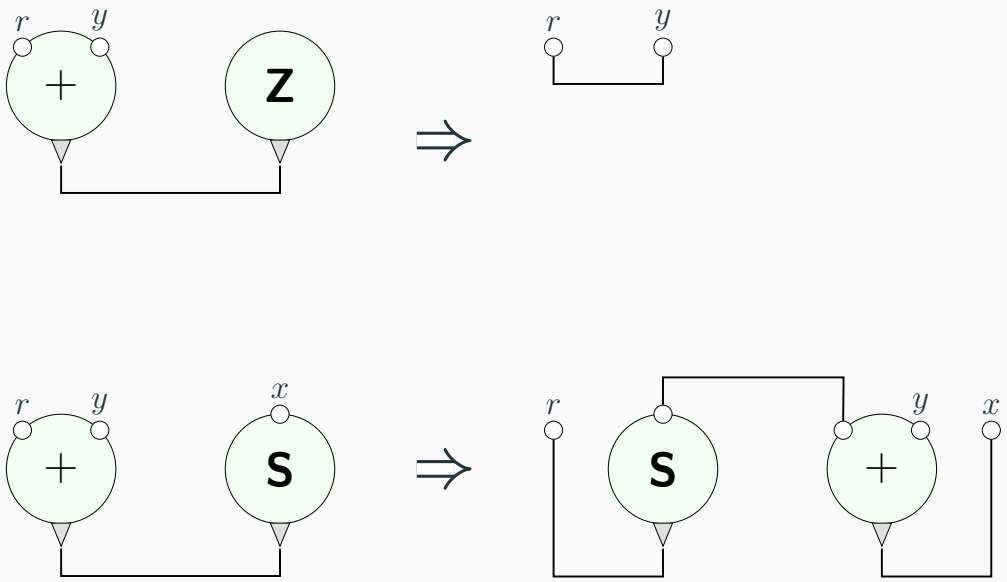


Agent label $\in \Sigma$; $ar : \Sigma \to \mathbb{N}$; $ar(\sigma)$ *auxiliary* ports .

May be cyclic, empty, disjoint.

Interaction (rewrite) rules defined between agents linked via *principal* ports.

Interaction rules

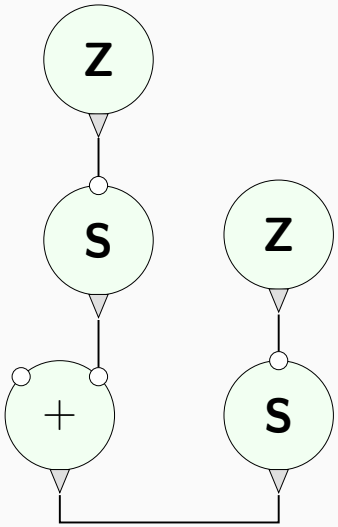


LHS (*active pair*) = two agents connected only by principal port.

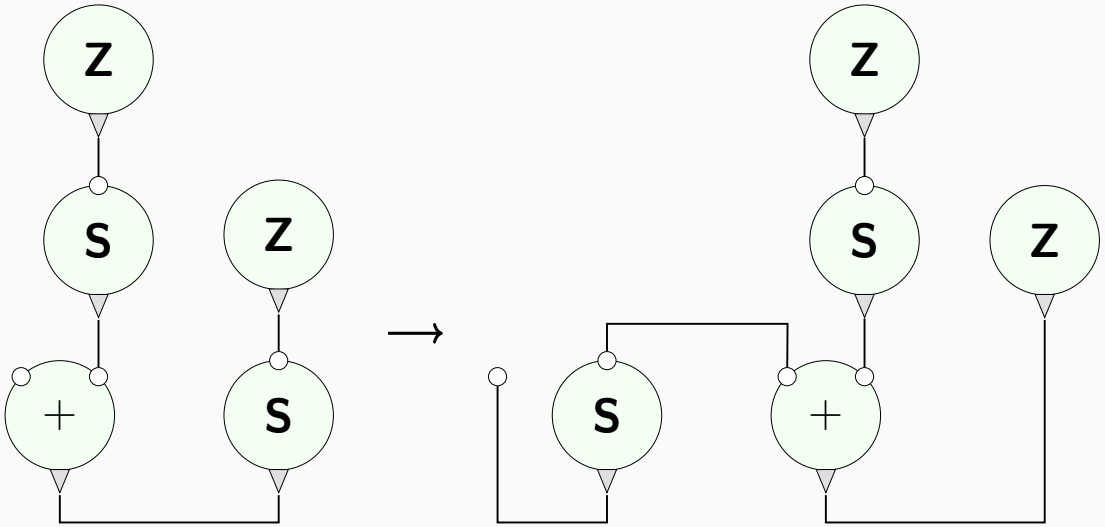
RHS = net that preserves LHS's *interface* (set of free ports).

LHSs are unique.

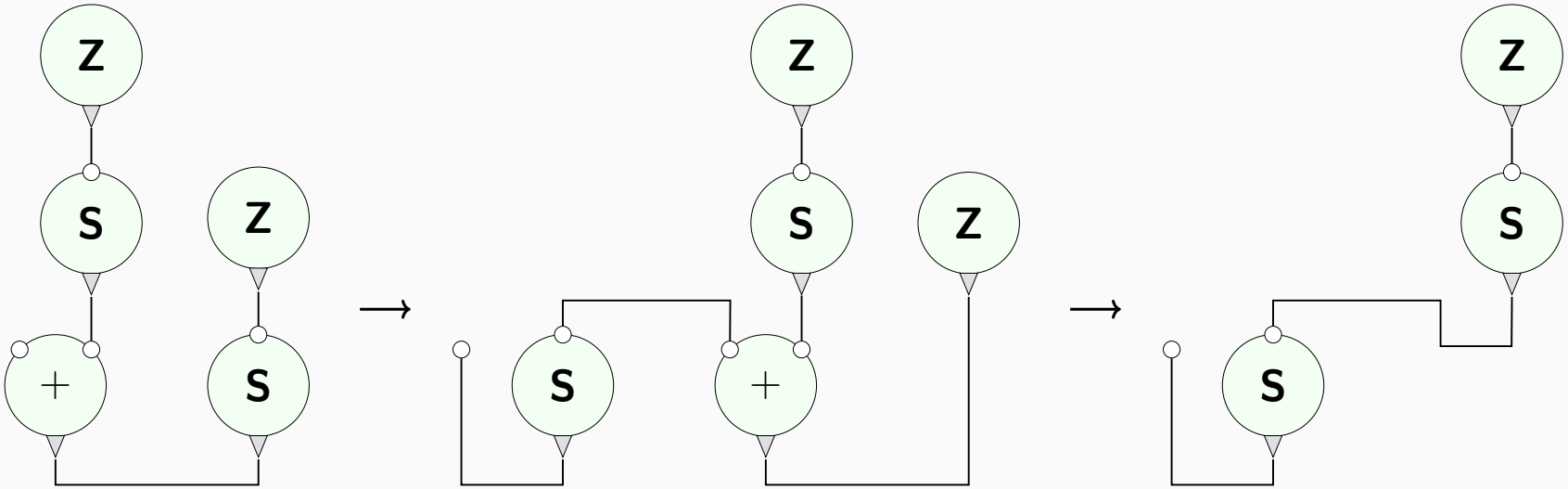
Interaction net evaluation



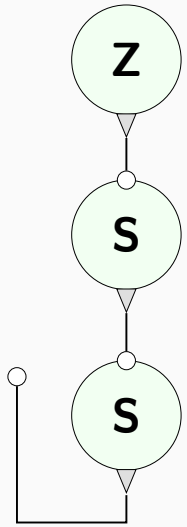
Interaction net evaluation



Interaction net evaluation



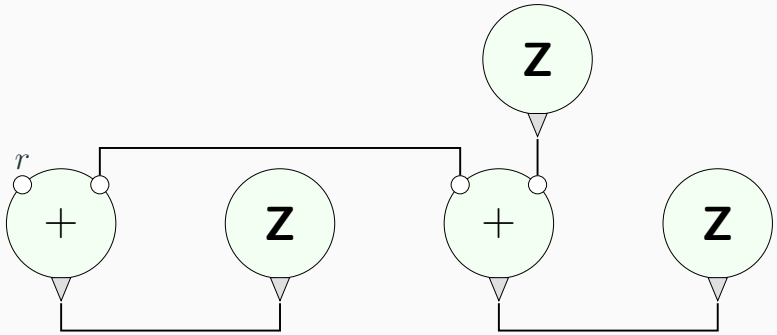
Result = normal form



$$N \Downarrow M \iff N \to^* M \text{ and } \nexists M' \text{ s.t. } M \to M'$$

Interaction nets as programs

- Turing complete.
- Explicit resource management.
- Parallel evaluation.



Hence

- `INPLA` : `github.com/inpla/inpla`
- `Higher Order Co`: `higherorderco.com`
- `tendrils`: `tendrils.co`

Interaction nets as programs

Questions:

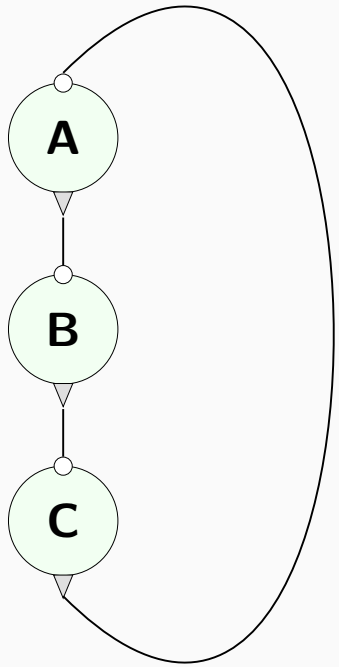
- What is input? Output?
- What are data?
- What are computations?
- What are results?
- How write nets?

Recall:

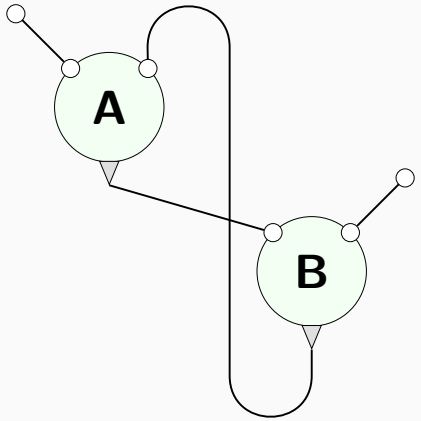
1. We can only reduce (compute) when there is an active pair.
2. Result of evaluation is normal form.

Problematic result #1 – Empty net

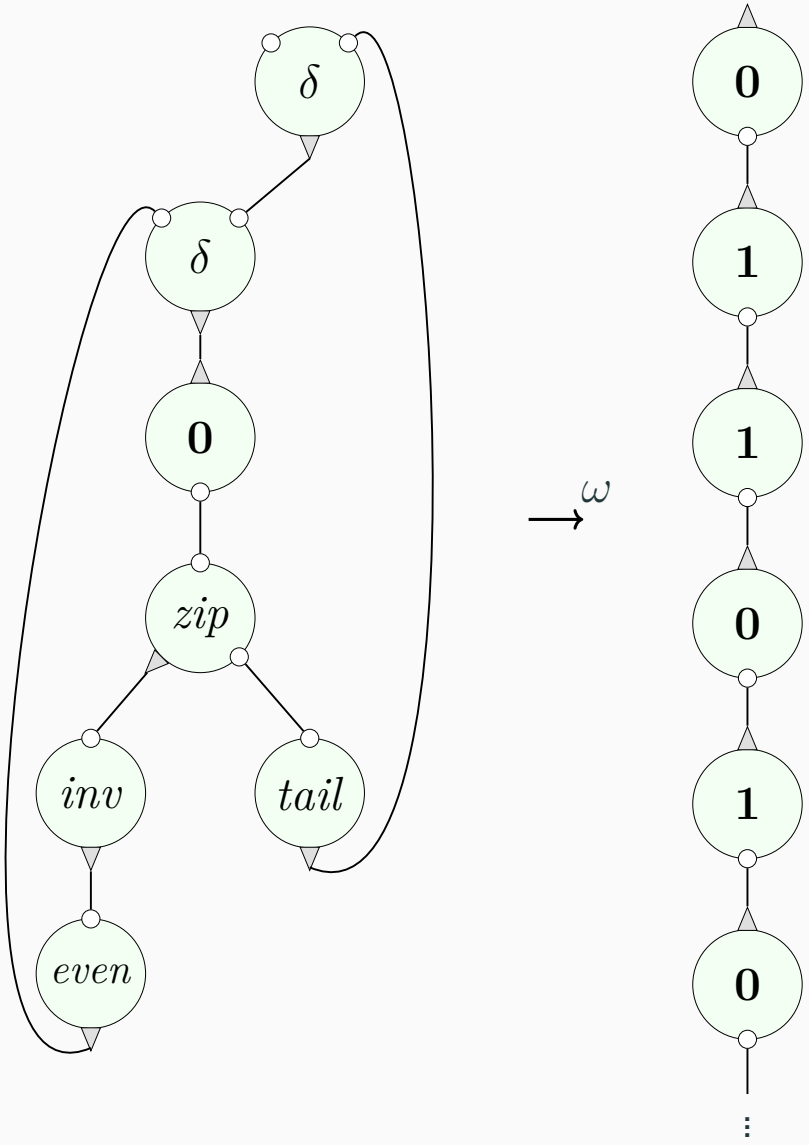
Problematic result #2 – Closed nets



Problematic result #3 – Vicious circles



Problematic non-result – Stream output



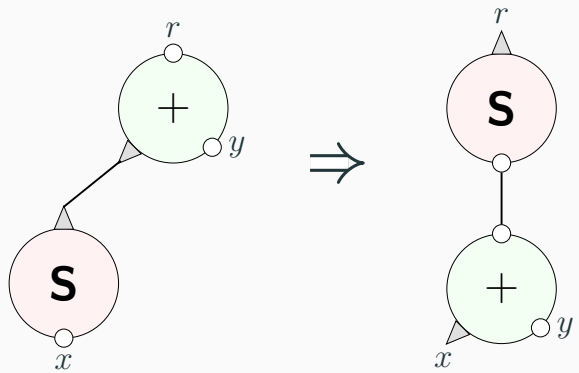
Function-constructor nets

Function-constructor nets

$\Sigma = \Sigma_{\mathcal{F}} \cup \Sigma_{\mathcal{C}}$; $\Sigma_{\mathcal{F}} \cap \Sigma_{\mathcal{C}} = \varnothing$

$\Sigma_{\mathcal{F}} = \{\epsilon/0,\ \mathsf{id}/1,\ \delta/2,\ \ldots,\ F/n_F\}$ — principal port is *input*.

$\Sigma_{\mathcal{C}} = \{\mathsf{Z}/0,\ \mathsf{S}/1,\ :/2,\ \ldots,\ \ C/n_C\}$ — principal port is (only) *output*.



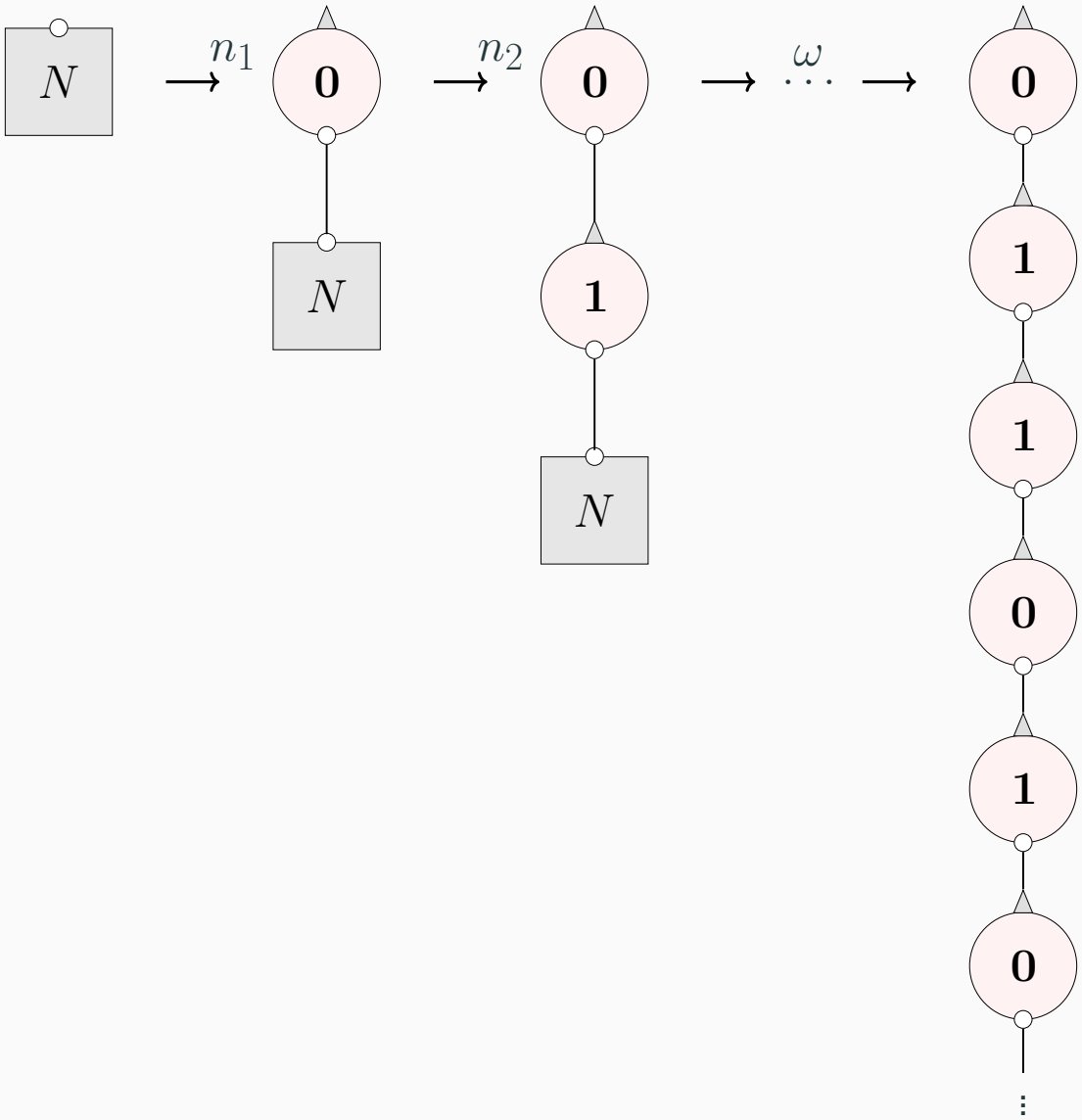
First-order system: constructors are function inputs and program outputs.

Still Turing complete.

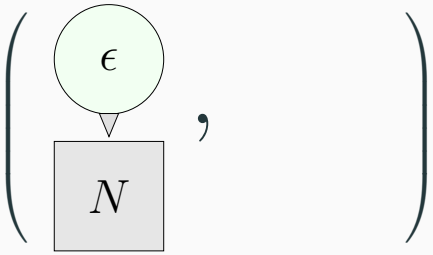
$\implies$ interface, $I = I_{in} \cup I_{out}$; $I_{in} \cap I_{out} = \varnothing$.

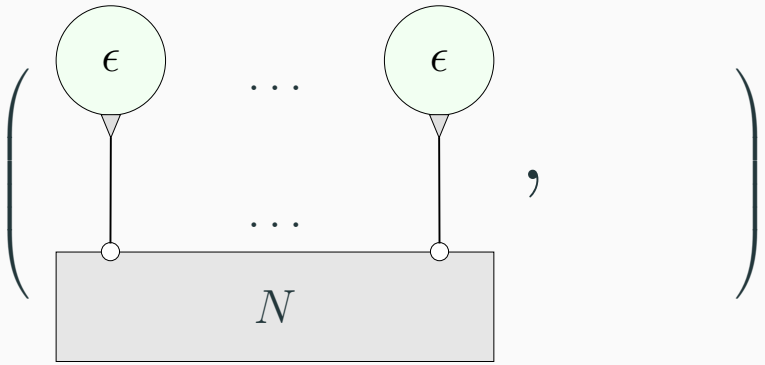
Consider nets with $I_{in} = \varnothing$.

Result = observational output

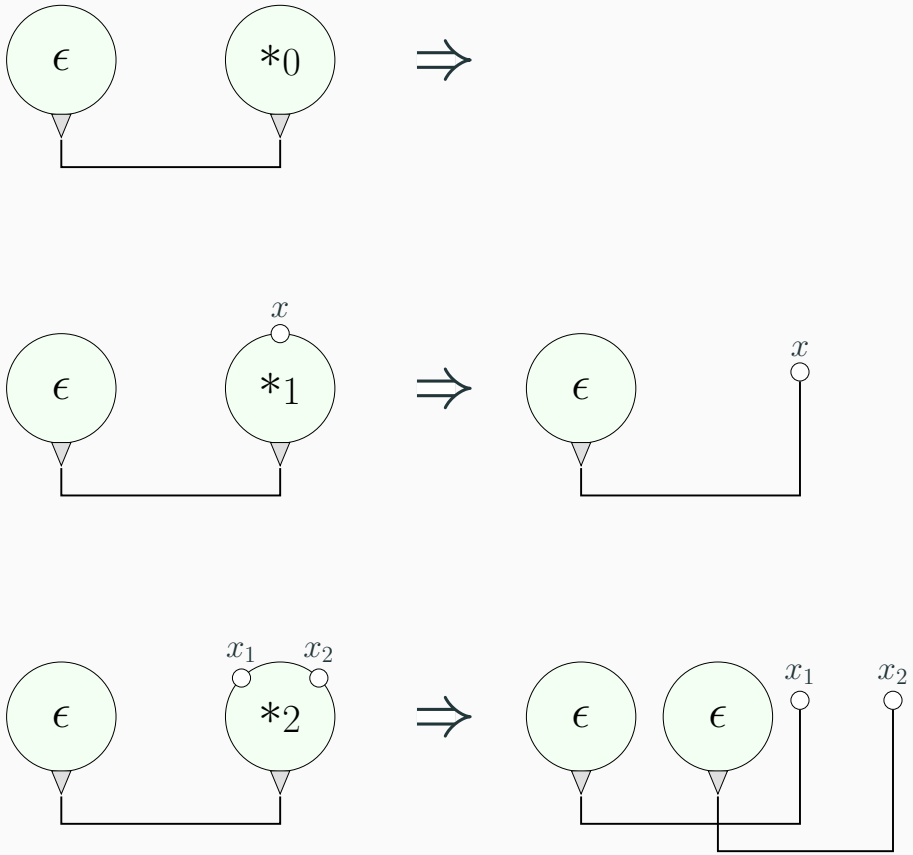


Function-constructor net semantics—set up

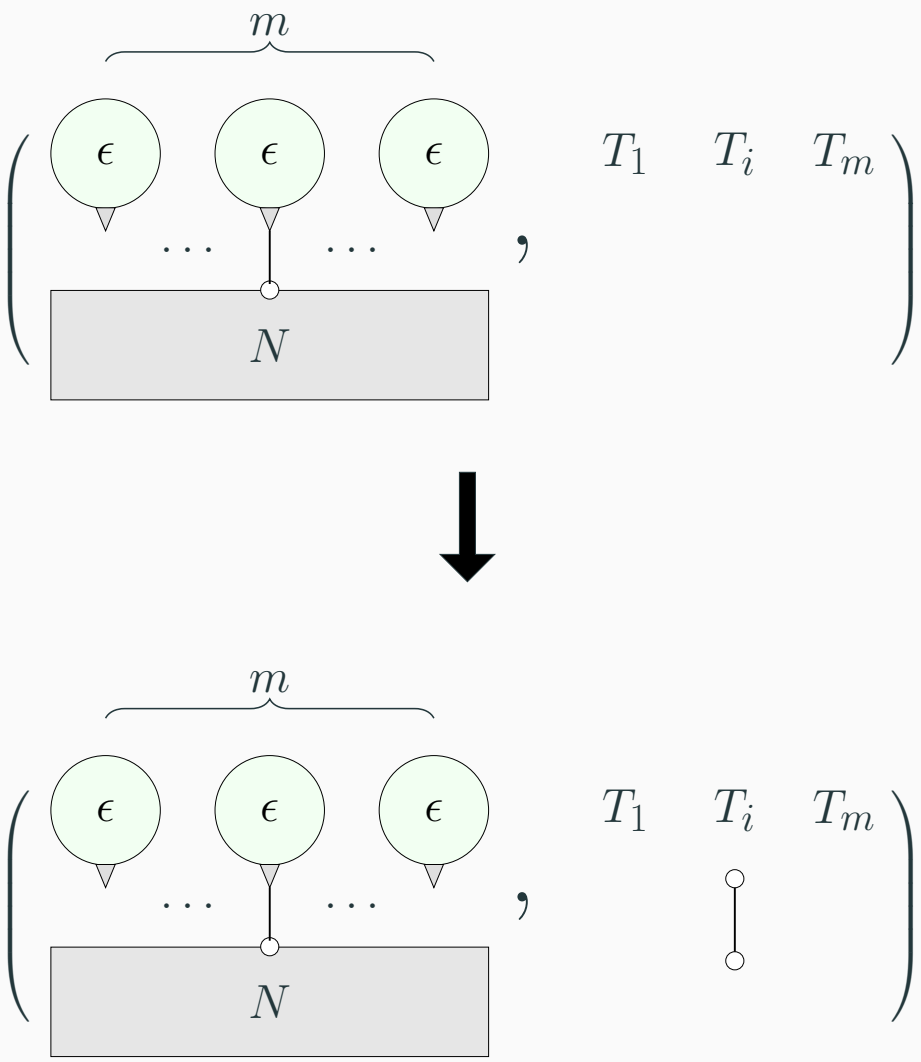




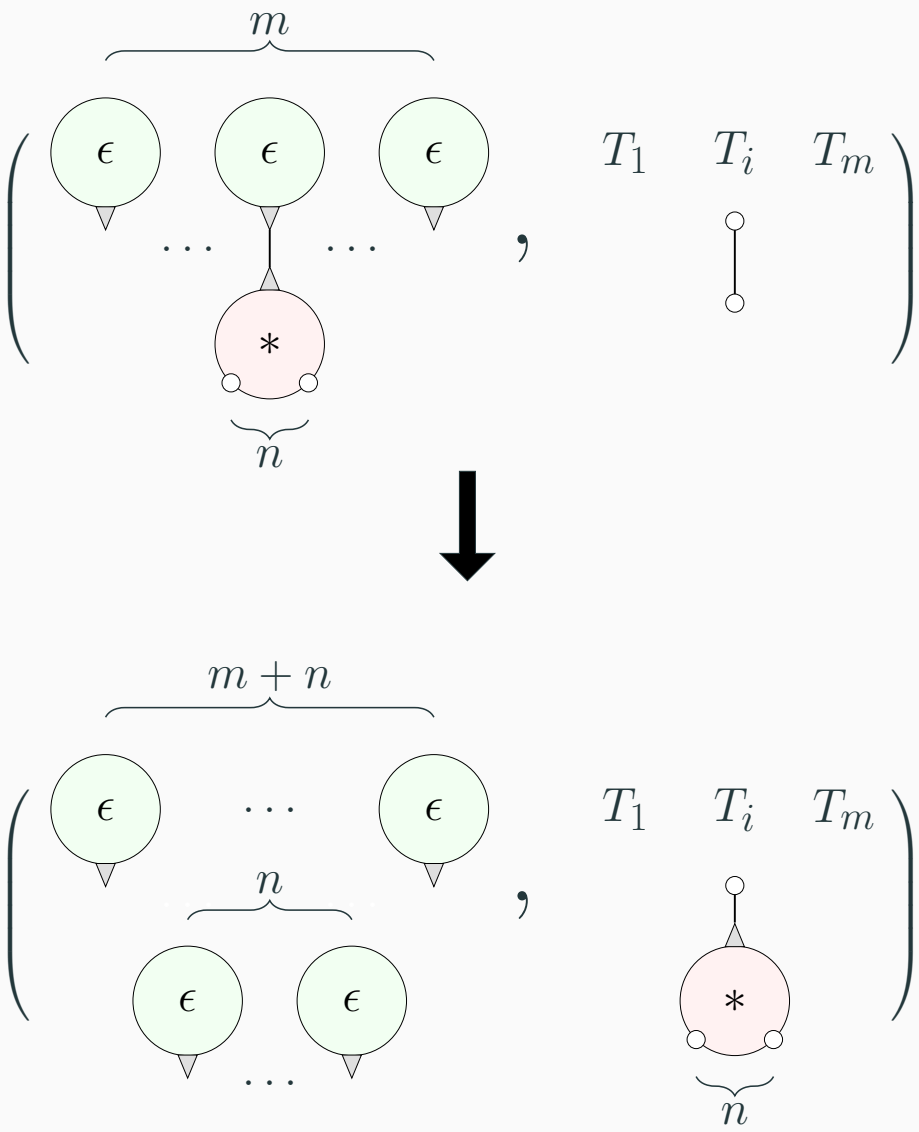
Interaction rules—erasure



Function-constructor net semantics—dynamics #1

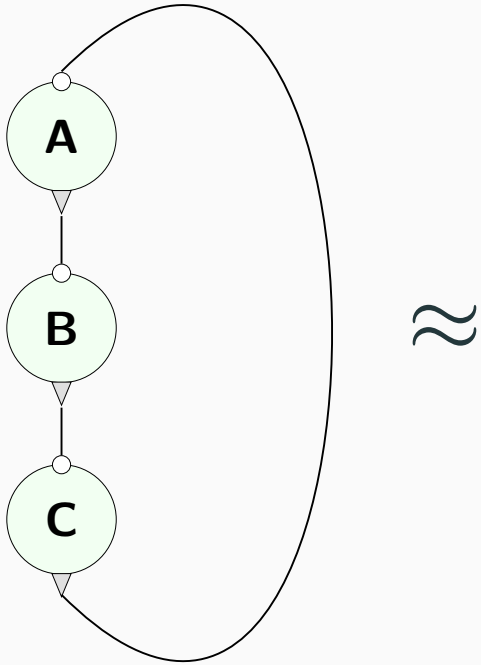


Function-constructor net semantics—dynamics #2



Observation value

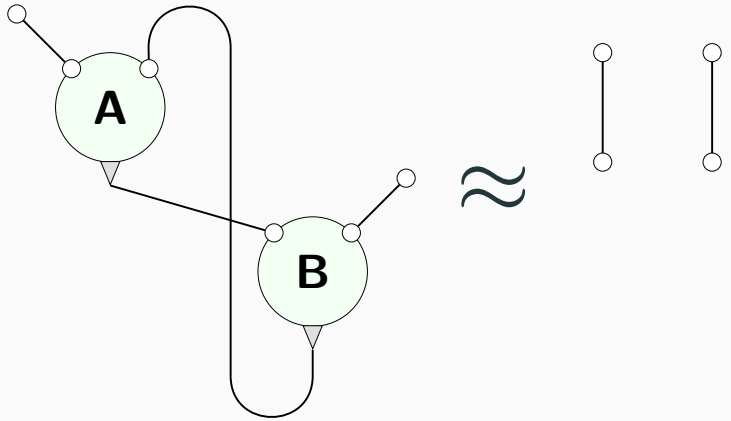
1. Inaccessible nets



≈

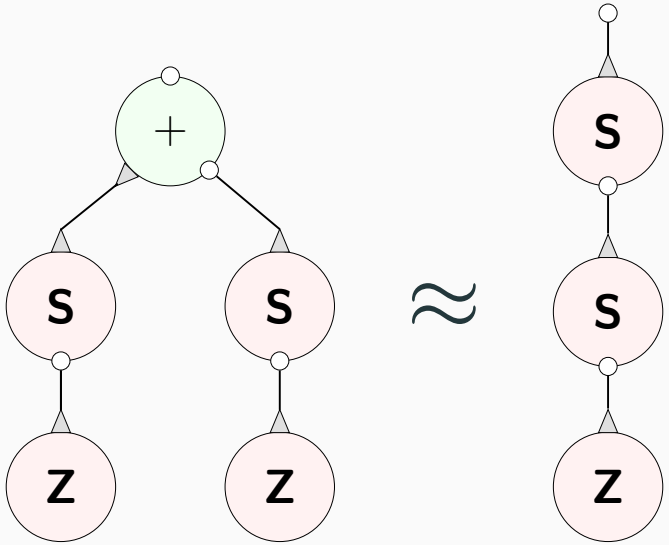
Observation value

2. Vicious circles



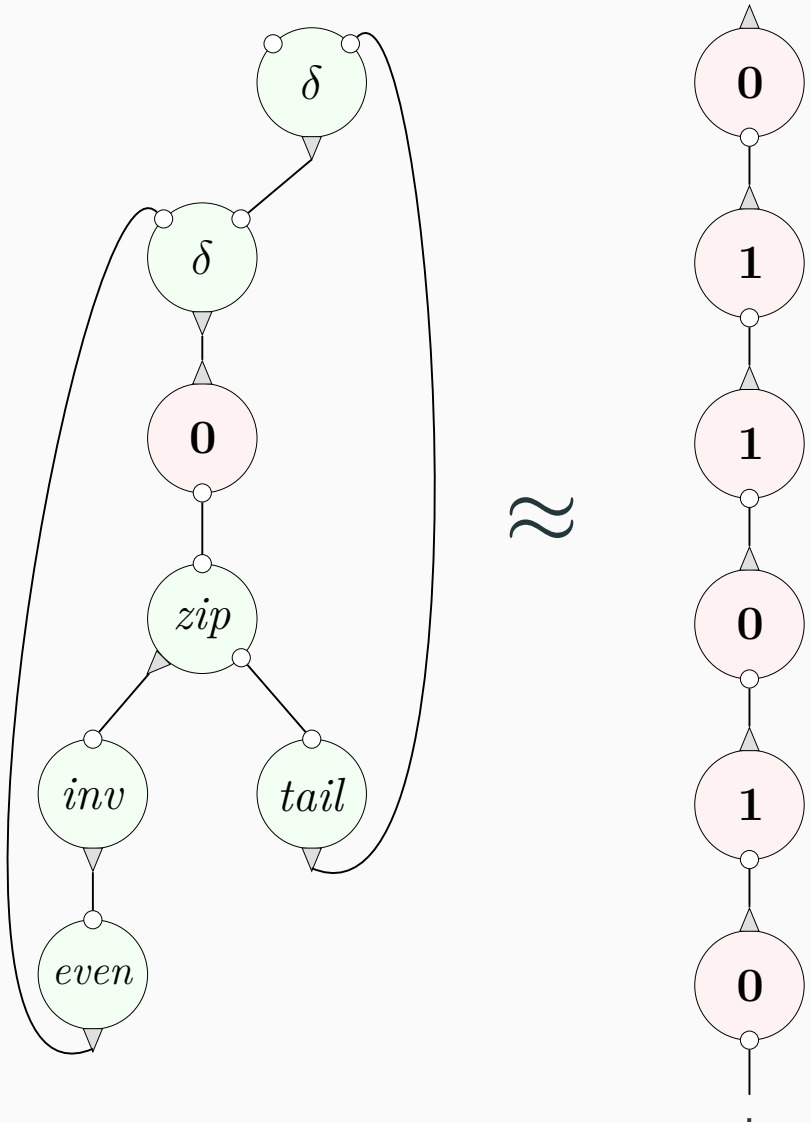
Observation value

3. Terminating (productive) nets



Observation value

4. Non-terminating (productive) nets



A Functional Language for Interaction Nets

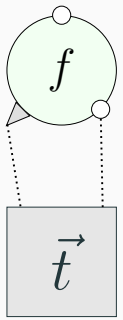
A Functional Language for Interaction Nets

Grammar (linear variable usage):

$$t, u := - \mid x \mid f(\vec{t}) \mid C(\vec{t}) \mid \text{let } \vec{x} = t \text{ in } u \mid t \| u$$

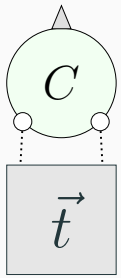
- $-$ is the empty net.
- x is a wire labelled x at its input port. 

- $f(\vec{t})$ is a function agent with each input port is connected to one element of $\vec{t}$.

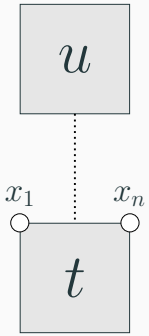


A Functional Language for Interaction Nets

- $C(\vec{t})$ is a constructor agent with input $\vec{t}$.



- let $\vec{x} = t$ in u : label u's output ports $\vec{x}$; connect to t, u (`letrec`).

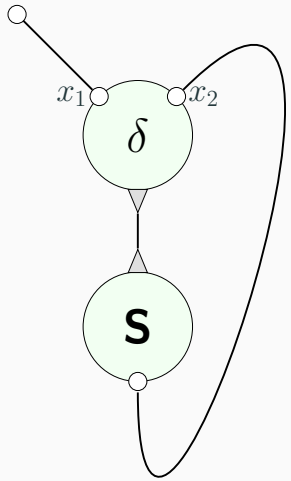


- $t \| u$ is the juxtaposition of the translated nets t and u.

Note: inputs explicit; output implicit (ex `letrec`)

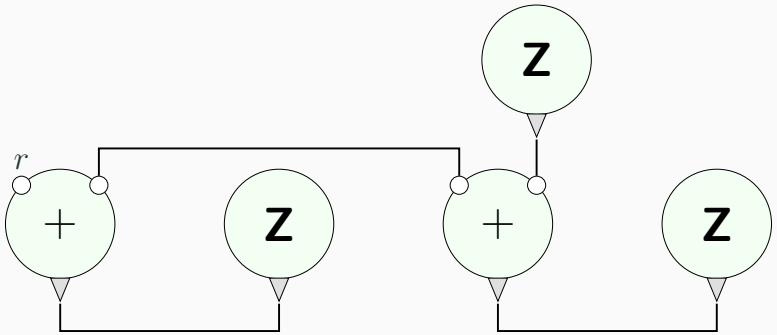
FLIN - example

```
let [x1,x2] = d(S(x2)) in x1
```



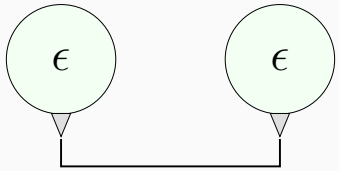
FLIN : bijection with FCNs

Every function-constructor net has a textual representation, and vice versa.




```
add(Z, add(Z, Z))
```

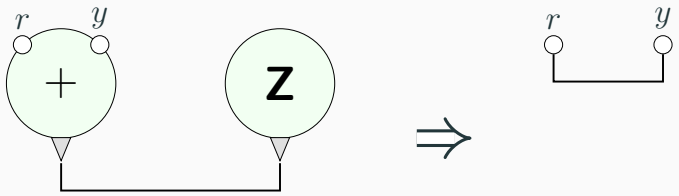
But not all interaction nets.



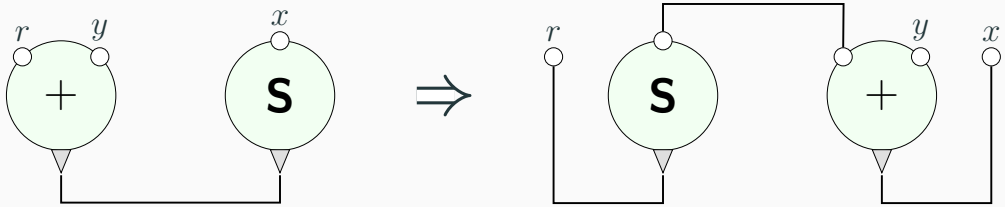
FLIN rules

$$\ell = r$$

where ℓ is a function with n_f outputs applied to a constructor with n_C inputs and n_x free inputs; r is a function-constructor term with n_f outputs and $n_C + n_x$ inputs.




```
add(Z, y) = y
```

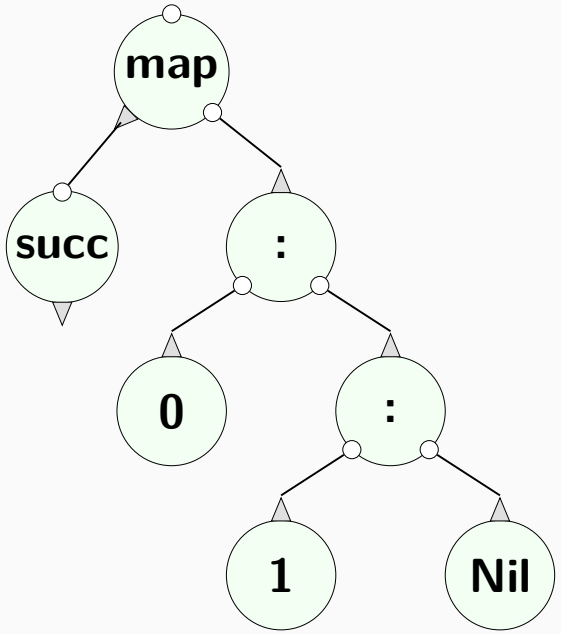



```
add(S(x), y) = S(add(x, y))
```

Higher-order functions

FLIN is first order:

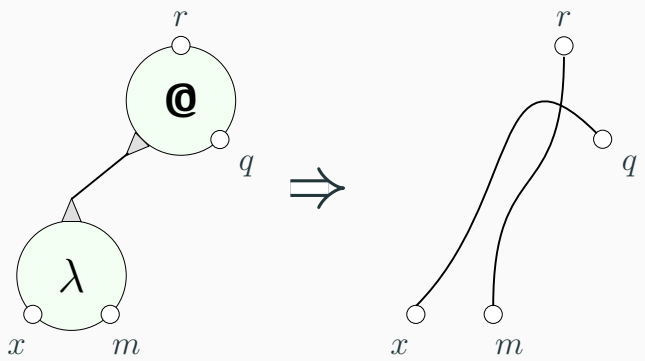
`map(succ(), [0,1])`:



Higher-order functions

Introduce *higher-order constructor*, λ, with two outputs: `\x.succ(x)` .

Interacts with application function, @:



Higher-order functions

Syntactic sugar:

```
map(^f, (x:xs)) = ((*f(x)) : (map(^f, xs))
```

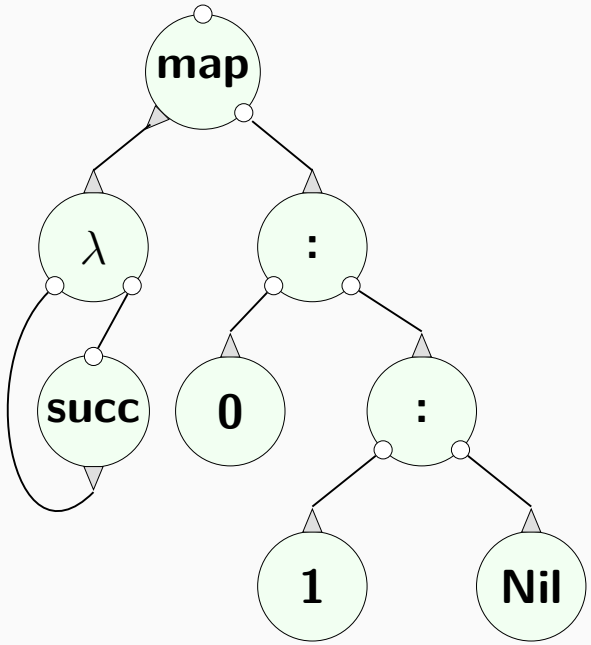
`^f` $\Longrightarrow$ λ-abstracted function .

`*f` $\Longrightarrow$ function agent.

```
map(\x.succ(x), [0,1])
```

or

```
map(^succ(), [0,1])
```



Prototype implementation

`https:github.com/MarcThatcher/Termgraph-implementation`

Translate FLIN $\rightarrow$ `INPLA`.

Extensions:

- Generic constructors (`*0`, `*1`, ...).
- Implicit memory management.
- Nested pattern matching, e.g. `last(x1:x2:xs) = last(x2:xs)`.
- Natural numbers (`succ`, `pred`).
- Multiple inputs, e.g. `max(x,y) = incr(max(y-1,x-1))`.
- Higher-order functions.

Batch and interactive modes; script to run batch mode end-to-end.

Example

```
fib(0)  = 0
fib(1)  = 1
fib(_n) = add(fib(pred(n)), fib(pred(pred(n))))
```

becomes:

```
fib(r) >< (int n) | n==0 => 0~r | n==1 => 1~r
                  | _ => Dup(n1,n2)~n, pred(r_0r_1_1_1)~n2,
                    pred(r_0r_1_1)~r_0r_1_1_1,
                    fib(r_0r_1)~r_0r_1_1,
                    pred(r_0_1)~n1, fib(r_0)~r_0_1,
                    add(r,r_0r_1)~r_0;
```