Martin Larralde
Biochemical Programming
2018-2019

κmachine

A program machine implementation in Kappa

1. Counter Machines

Counter Machine

- Primitive model of *register machines*, close to actual computers
- Finite numbers of registers
- Small number of instructions:

  clr(r); inc(r); dec(r); cpy(r1, r2); jz(r, z); je(r1, r2, z)

- Program consists in a list of labelled instructions
- Turing complete *(with a few tricks)* !

Program Machine

- Defined by Marvin Minsky in 1967[1]
- Base model using the following instructions:

    inc(r); dec(r); jnz(r, z)

- All the remaining instructions can be emulated *(but require more registers to do so)*

1: Minsky, M. L. (1967). Computation: Finite and Infinite Machines. *ISBN: 978-0-13-165563-8*

2. Biological Model

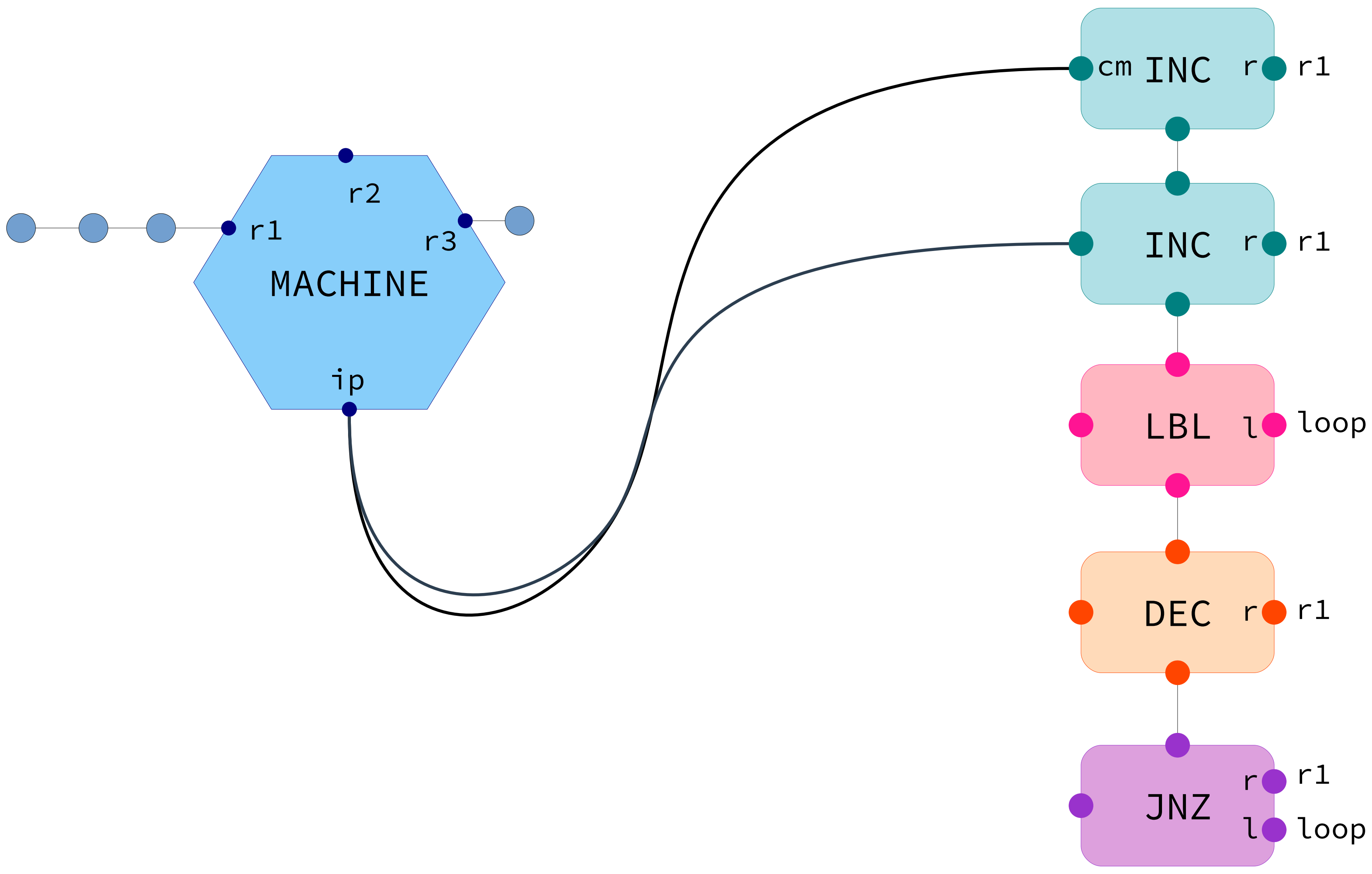
Program

- Polymer of instruction agents
- One agent /instruction, states as arguments
- Labels are treated as instructions

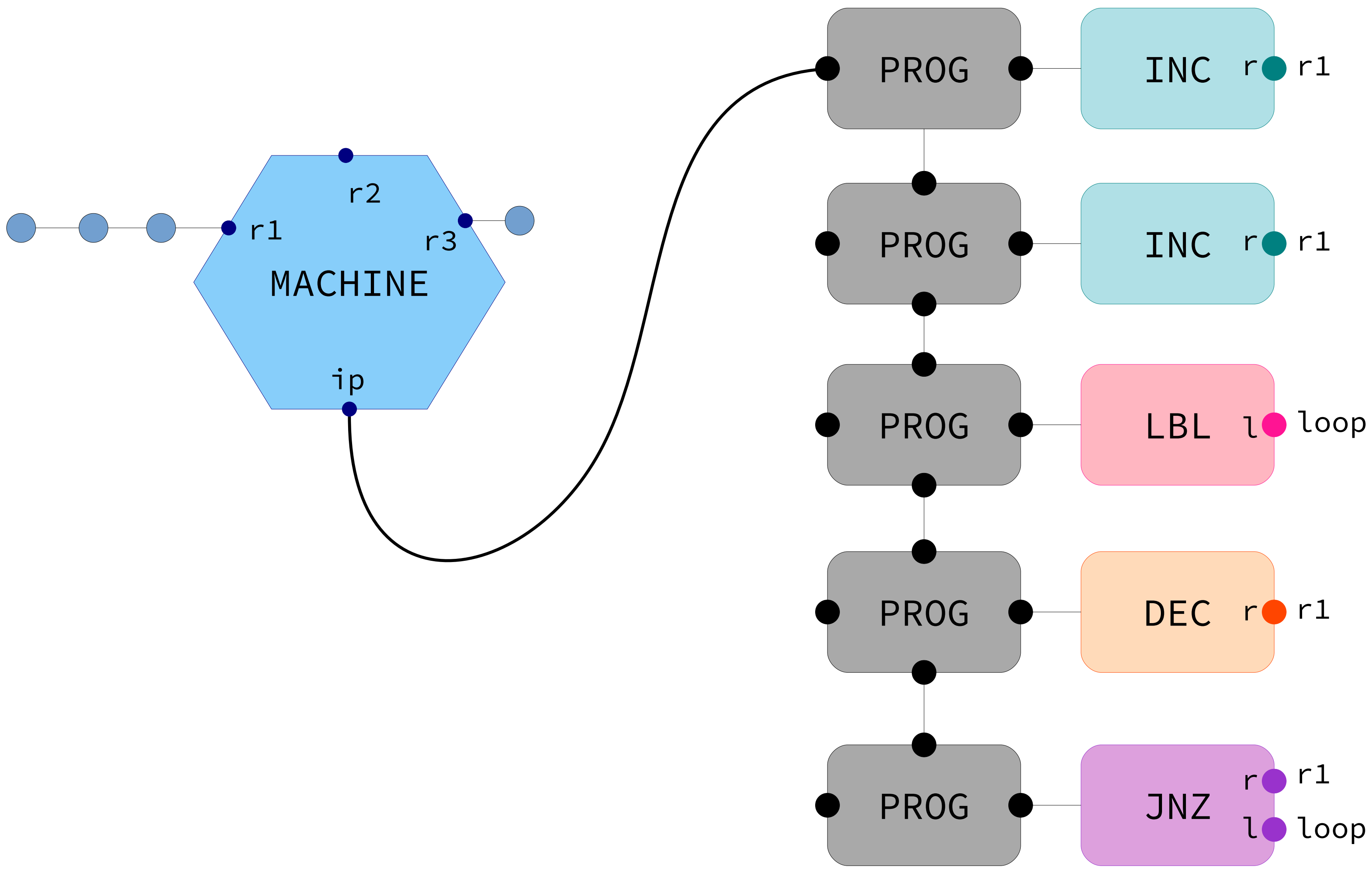
Machine

- The machine is modelled as a single agent
- Each register is a binding site
- Register value is stored as a polymer of *units*
- Units are agents

The Big Picture



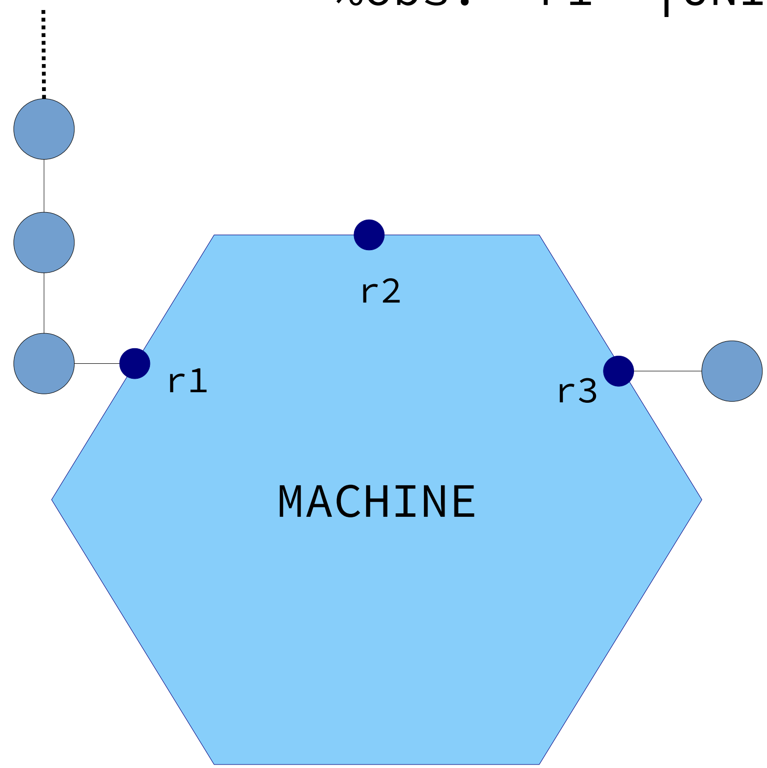
The ACTUAL Big Picture



Agents – Machine

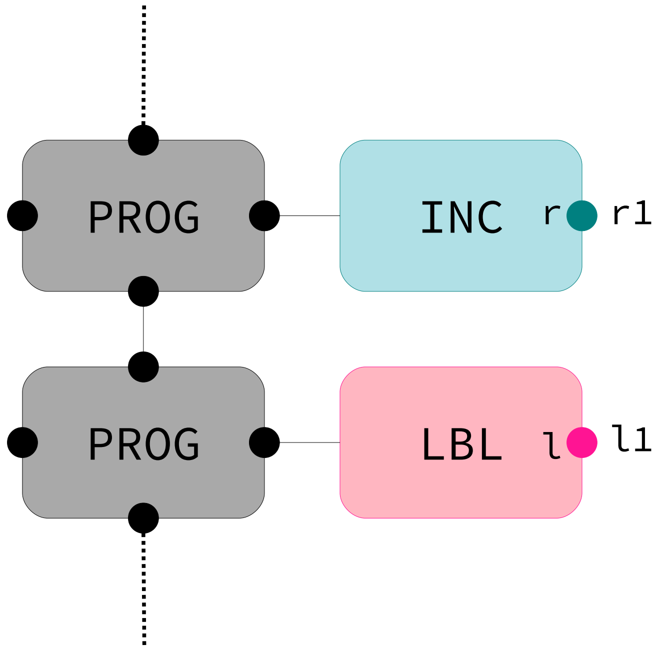
```
%agent: UNIT(
    prev[next.UNIT, r1.MACHINE, …,],
    next[prev.UNIT],
    r{none, r1, …}
)

%agent: MACHINE(
    ip[cm.PROG],
    state{run, move, bind},
    target{none, l1, …},
    r1[prev.UNIT],
    …,
)
```

```
%obs: ‘r1’ |UNIT(r{r1})|
```



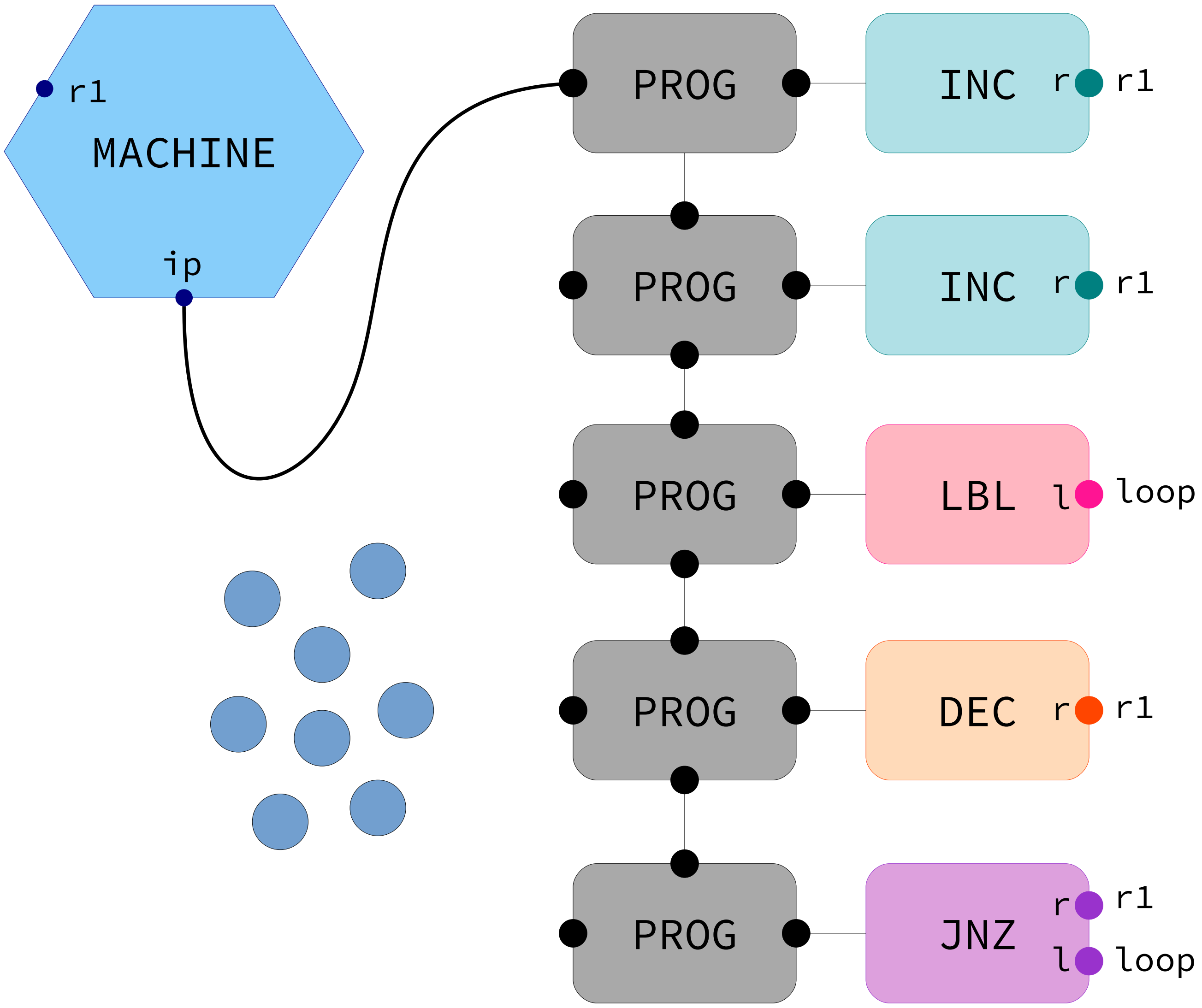
Agents – Program

```
%agent: PROG(
    prev[next.PROG],
    next[prev.PROG],
    cm[ip.MACHINE],
    ins[prog.ADD, …],
)
%agent: INC(
    prog[ins.PROG],
    r{r1, …},
)
%agent: JNZ(
    prog[ins.PROG],
    r{r1, …},
    l{l1, …},
)
%agent: LBL(
    prog[ins.PROG],
    l{l1, …},
)
```



Initial State

```
%init: 1
    MACHINE(
        ip[0],
        state{run},
        target{none},
        r1[.]
    ),
    PROG (
        cm[0],
        ins[1],
        next[2],
    ),
    …


%init: N UNIT()
```

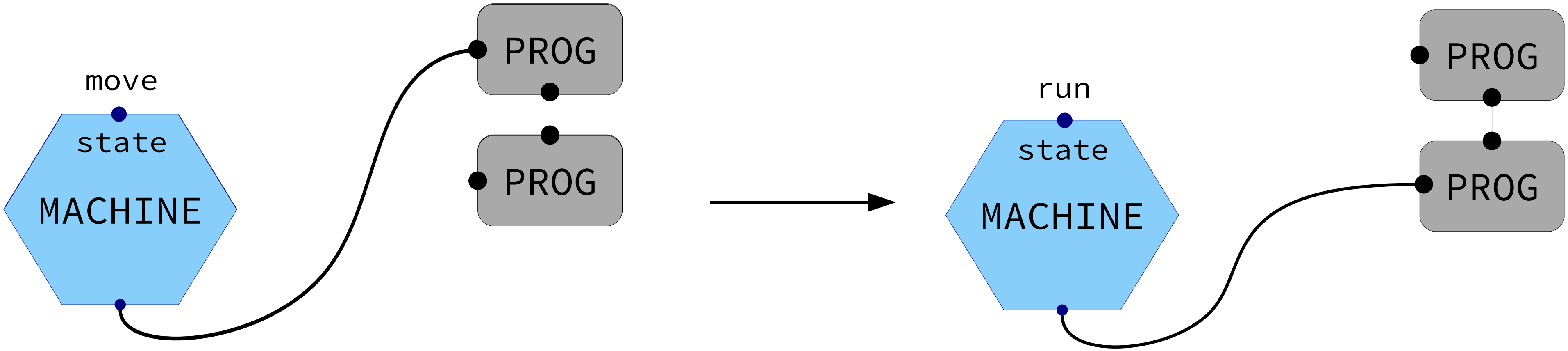


3. Execution

Execution

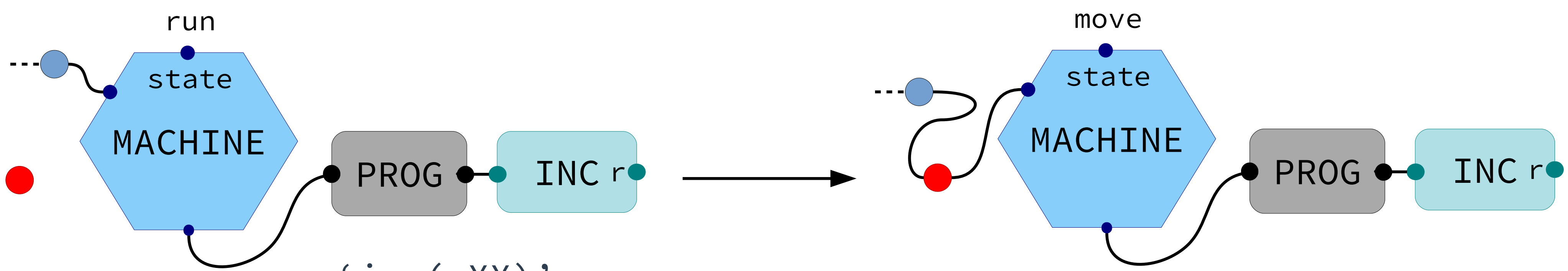
- **Sequential execution in 2 steps:**
  - Execute the instruction
  - Move to the next instruction
- **We store the current step (*run* or *move*) as a state.**

Execution – Movement



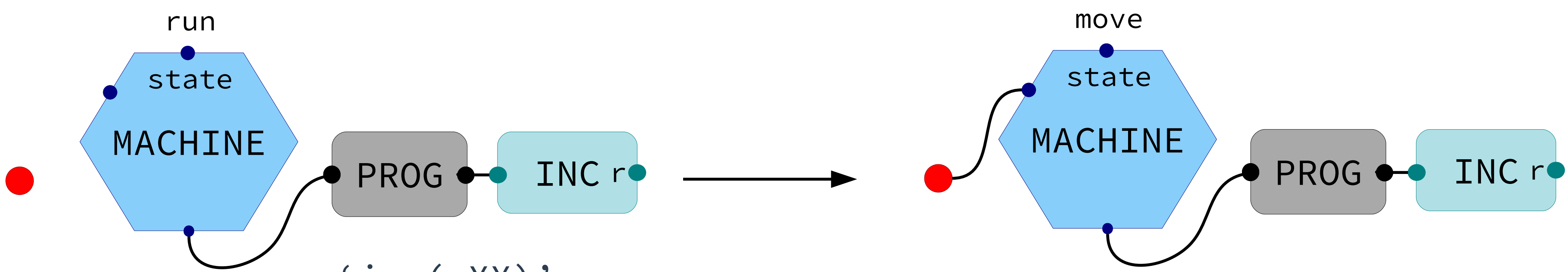

```
‘mov’
        MACHINE(state{move}, ip[1]),
        PROG(cm[1], next[2]),
        PROG(cm[.], prev[2])
→
        MACHINE(state{run}, ip[1]),
        PROG(cm[.], next[2]),
        PROG(cm[1], prev[2])
@ 1
```

Execution – *inc(r), ∀r*



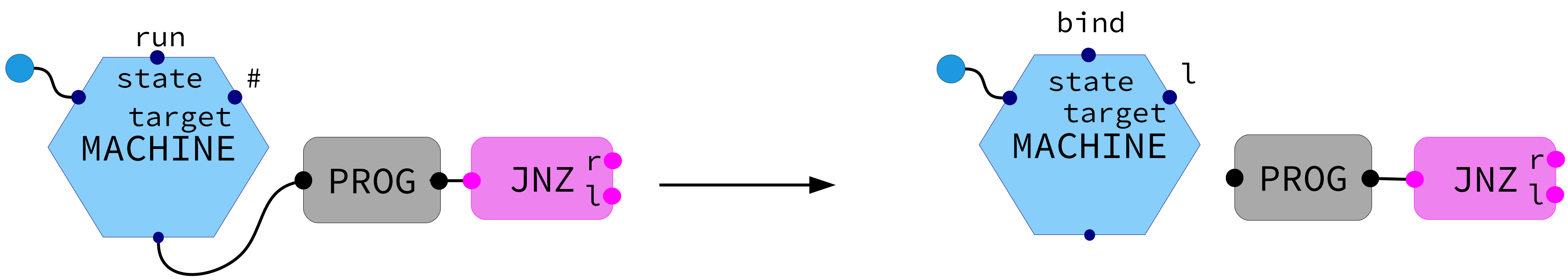

```
'inc(rXX)'
        MACHINE(state{run}, ip[1], rXX[3]),
        PROG(cm[1], ins[2]),
        INC(prog[2], r{rXX}),
        UNIT(prev[3]),
        UNIT(prev[.], next[.], r{none})
→
        MACHINE(state{move}, ip[1], rXX[3]),
        PROG(cm[1], ins[2]),
        INC(prog[2], r{rXX}),
        UNIT(prev[4]),
        UNIT(prev[3], next[4], r{rXX})
@ 1
```

Execution – *inc(r)*, $\forall r$ $\quad (\llbracket r \rrbracket_\rho = 0)$



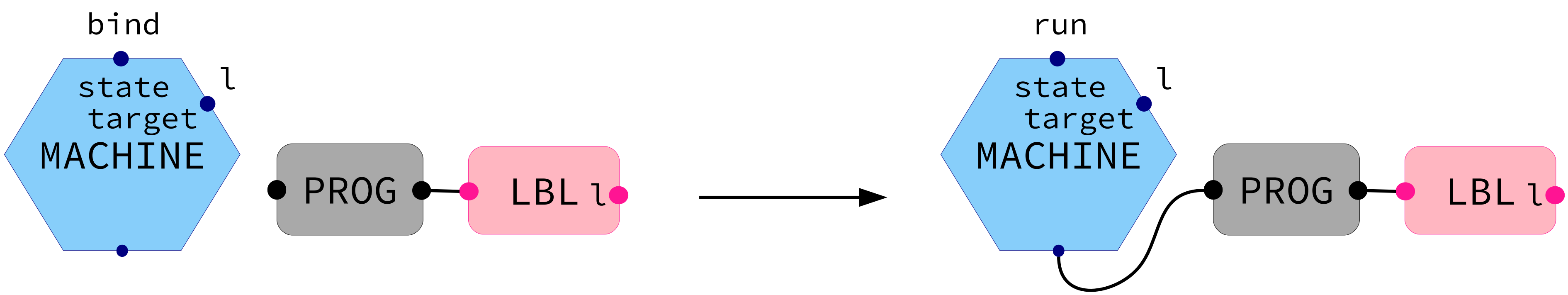

```
'inc(rXX)'
        MACHINE(state{run}, ip[1], rXX[.]),
        PROG(cm[1], ins[2]),
        INC(prog[2], r{rXX}),
        UNIT(prev[.], r{none})

→

        MACHINE(state{move}, ip[1], rXX[3]),
        PROG(cm[1], ins[2]),
        INC(prog[2], r{rXX}),
        UNIT(prev[3], r{rXX})

@ 1
```

Execution – *jnz(r, z)*, ∀*r*, ∀*z* ($[\![r]\!]_\rho \neq 0$)




```
‘jnz(rXX, lXX)’
        MACHINE(state{run}, ip[1], target{#}, rXX[_]),
        PROG(cm[1], ins[2]),
        JNZ(prog[2], r{rXX}, l{lXX}),
→
        MACHINE(state{bind}, ip[.], target{lXX}, rXX[3]),
        PROG(cm[.], ins[2]),
        JNZ(prog[2], r{rXX}, l{lXX}),
@ 1
```

Execution – Binding




```
'bind(lXX)'
        MACHINE(state{bind}, ip[.], target{lXX}),
        PROG(cm[.], ins[1]),
        LBL(prog[1], l{lXX}),
→
        MACHINE(state{run}, ip[2], target{none}),
        PROG(cm[2], ins[1]),
        LBL(prog[1], l{lXX}),
@ 1
```

4. Optimisations

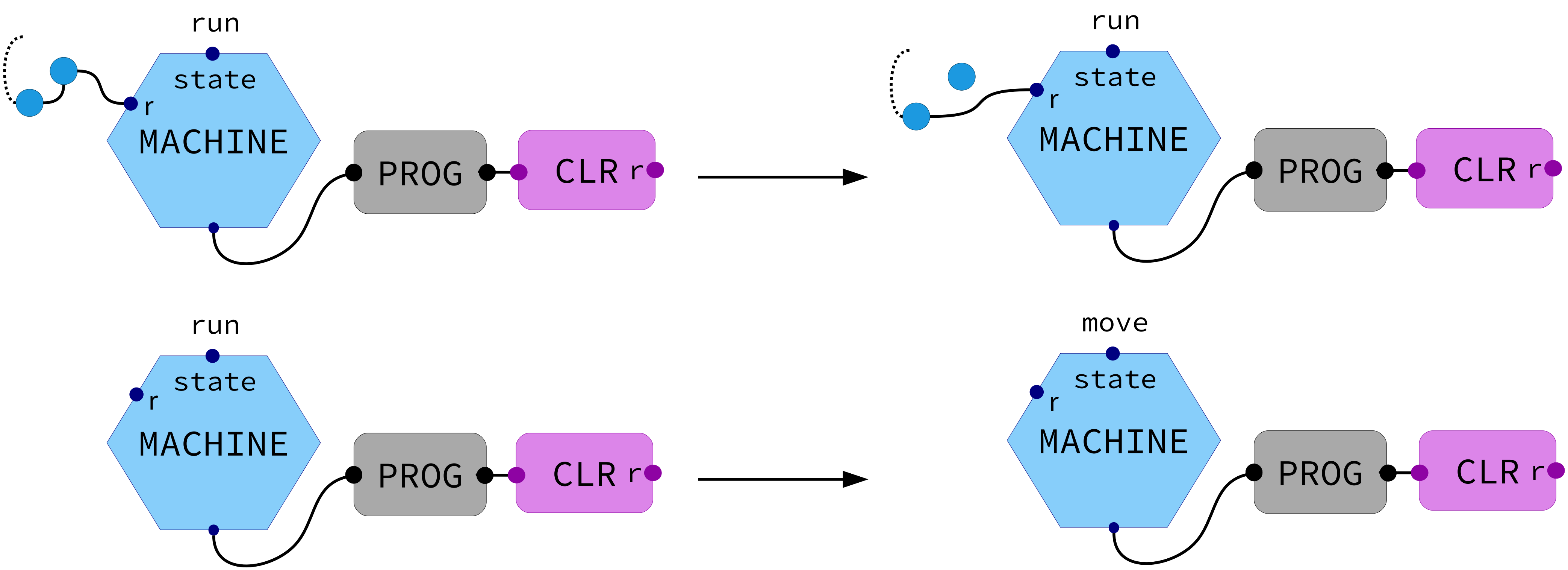
Usual emulation of *clr(r)*

- *clr* can easily be implemented with *dec* and *jnz* :

```
clr %rax   ──────►   clear:
                       dec %rax
                       jnz %rax, clear
```

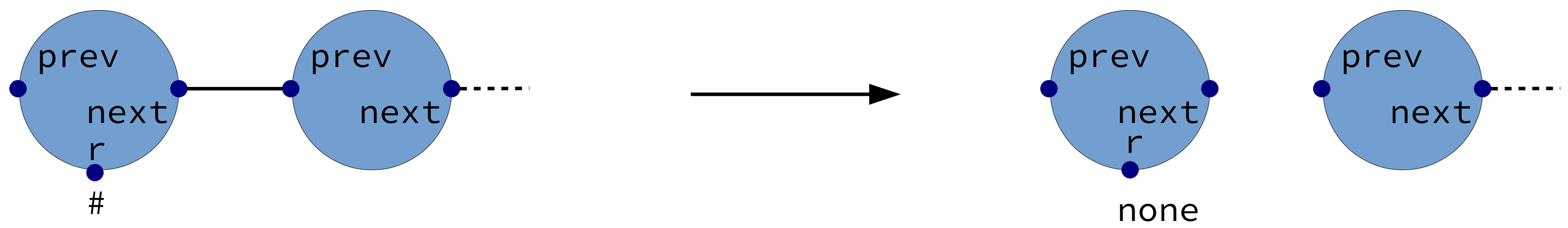
Direct Kappa implementation of *clr(r)*

- Possible implementation in Kappa without emulation:



Actual Kappa implementation of *clr(r)*

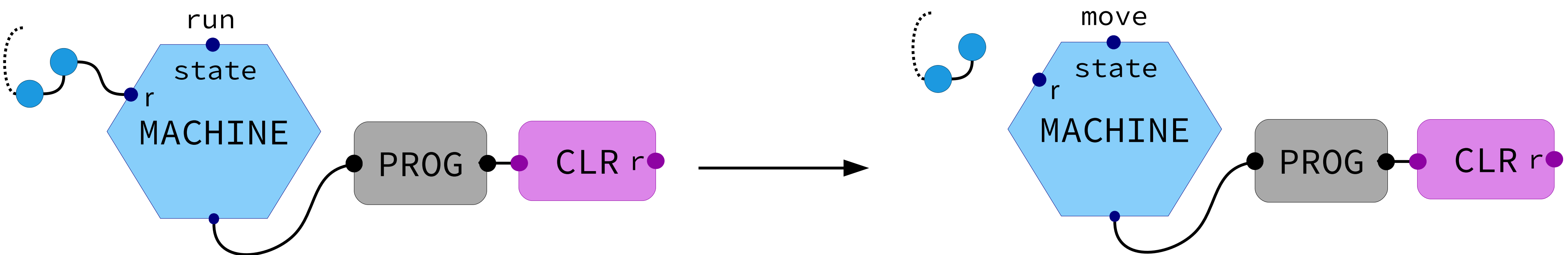
- Trick: force depolymerisation of unbound UNIT polymers:




```
'reset units'
        UNIT(prev[.], next[_], r{#})
->
        UNIT(prev[.], next[.], r{none})
@ inf
```

Actual Kappa implementation of *clr(r)*

- *clr(r)* is O(1), UNITs are reset in the background




```
'clr(rXX)'
        MACHINE(state{run}, ip[1], rXX[3]),
        PROG(cm[1], ins[2]),
        CLR(prog[2], r{rXX}),
        UNIT(prev[3]),
->
        MACHINE(state{move}, ip[1], rXX[.]),
        PROG(cm[1], ins[2]),
        CLR(prog[2], r{rXX}),
        UNIT(prev[.]),
@ 1
```

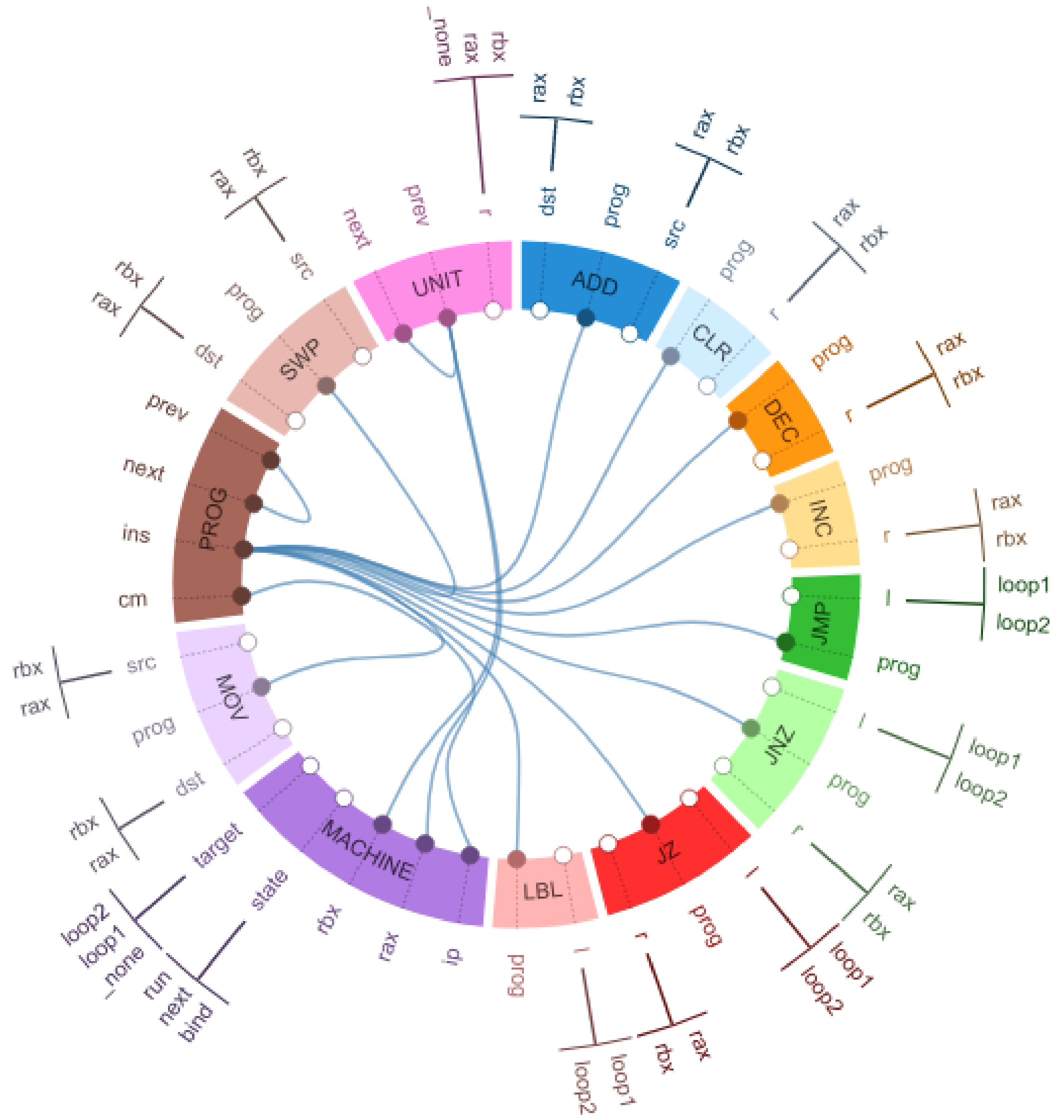
5. Demonstration

Instructions

- **Compiler from pseudo-ASM to Kappa code**
- ***https://github.com/althonos/kmachine***
- **Following instructions available:**

add(r1, r2)	*O(1)*
clr(r)	*O(1)*
cpy(r1, r2)	*O([r1])*
inc(r)	*O(1)*
jmp(z)	*O(1)*
jnz(r, z)	*O(1)*
jz(z)	*O(1)*
mov(r1, r2)	*O(1)*
mul(r1, r2)	*O([r1])*
swp(r1, r2)	*O(1)*

Contact Map



6. Questions ?