A Mechanized Semantics for Dataflow Circuits

Sandrine Blazy

IFIP WG 1.9/2.15, Paris, 2025-07-01

Verified compilation

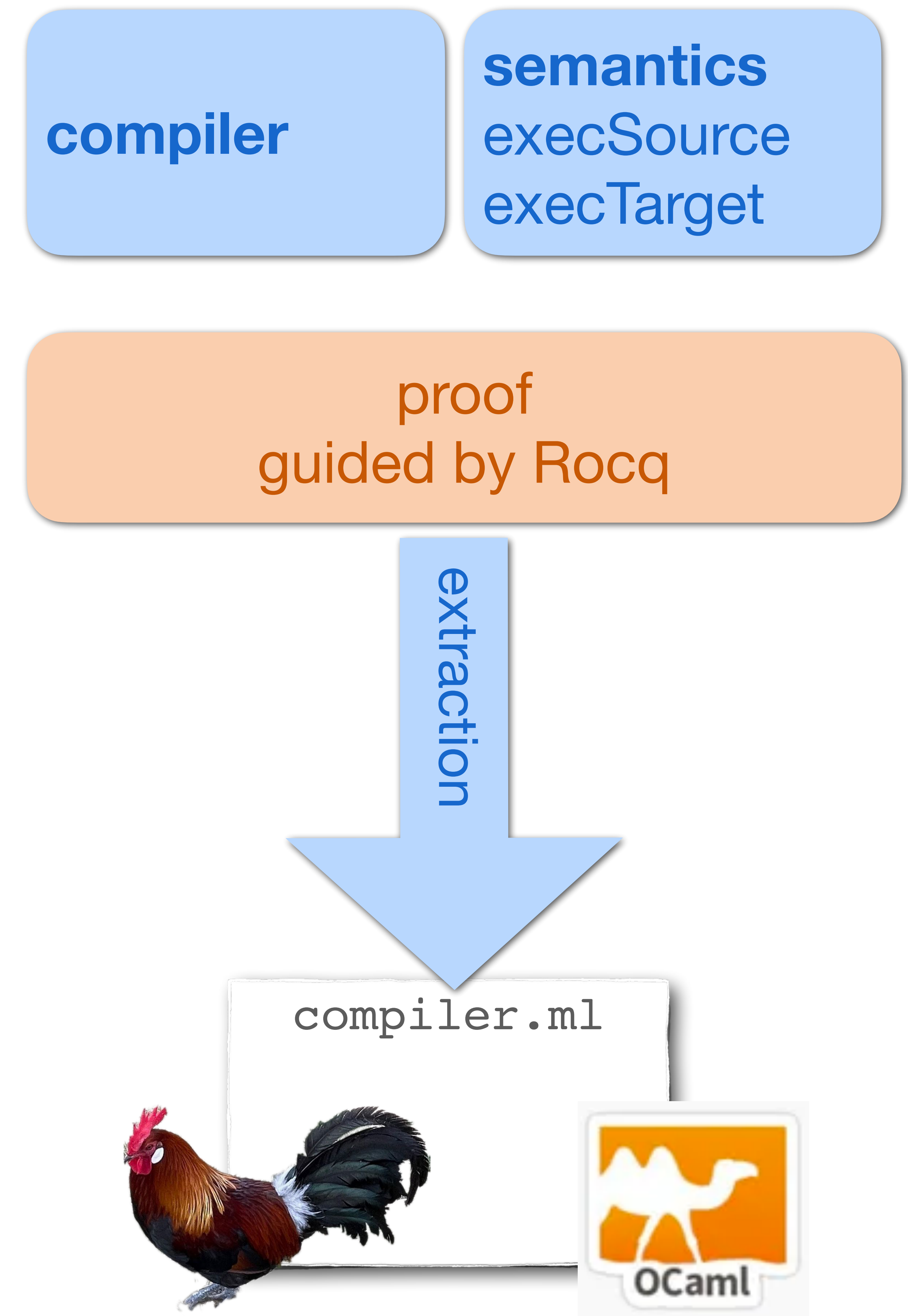
Program the compiler using a purely functional language

State its correctness w.r.t. the operational semantics of its source and target languages:
« The generated code must behave as prescribed by the semantics of the source program. »

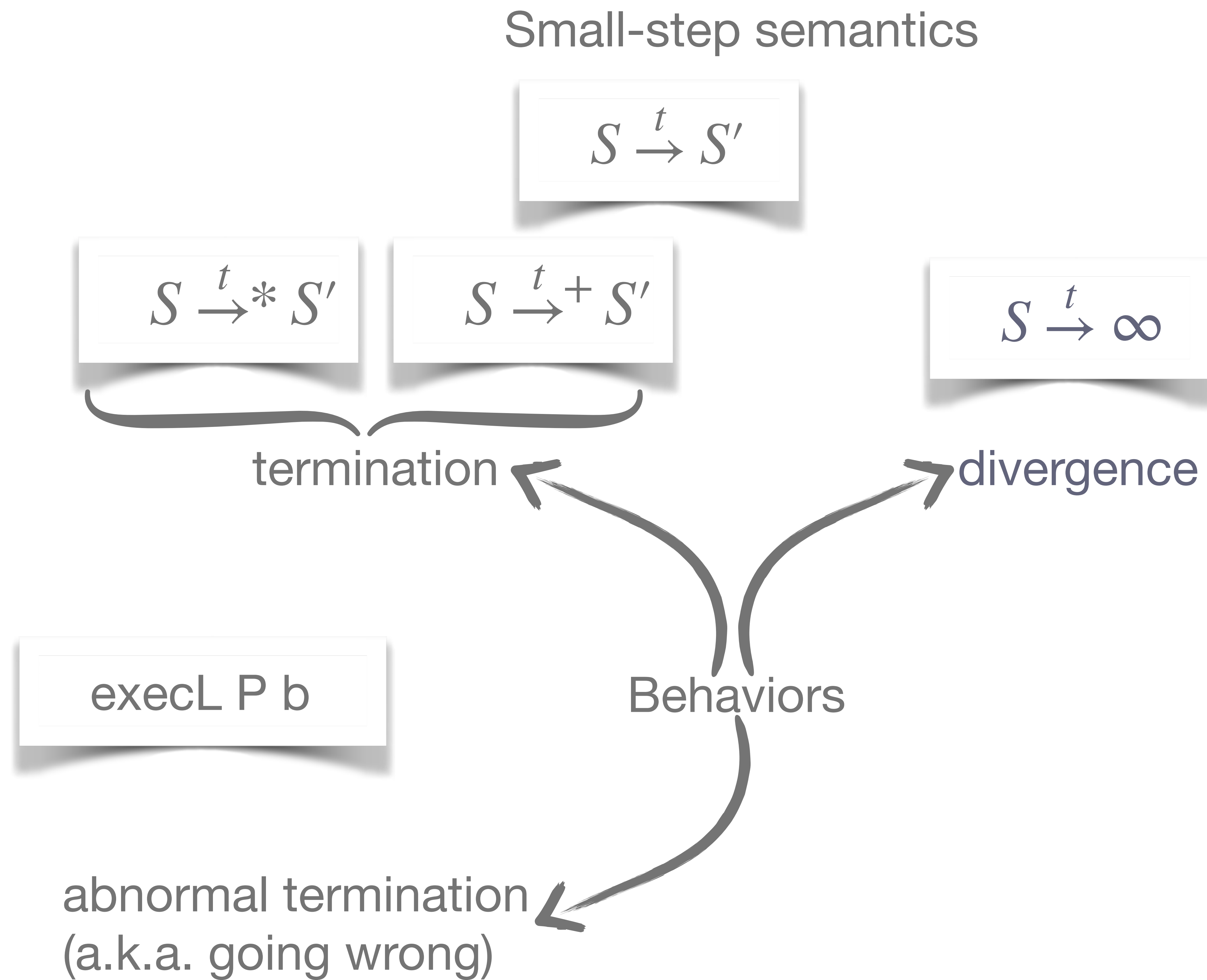
Prove this property interactively and mechanically

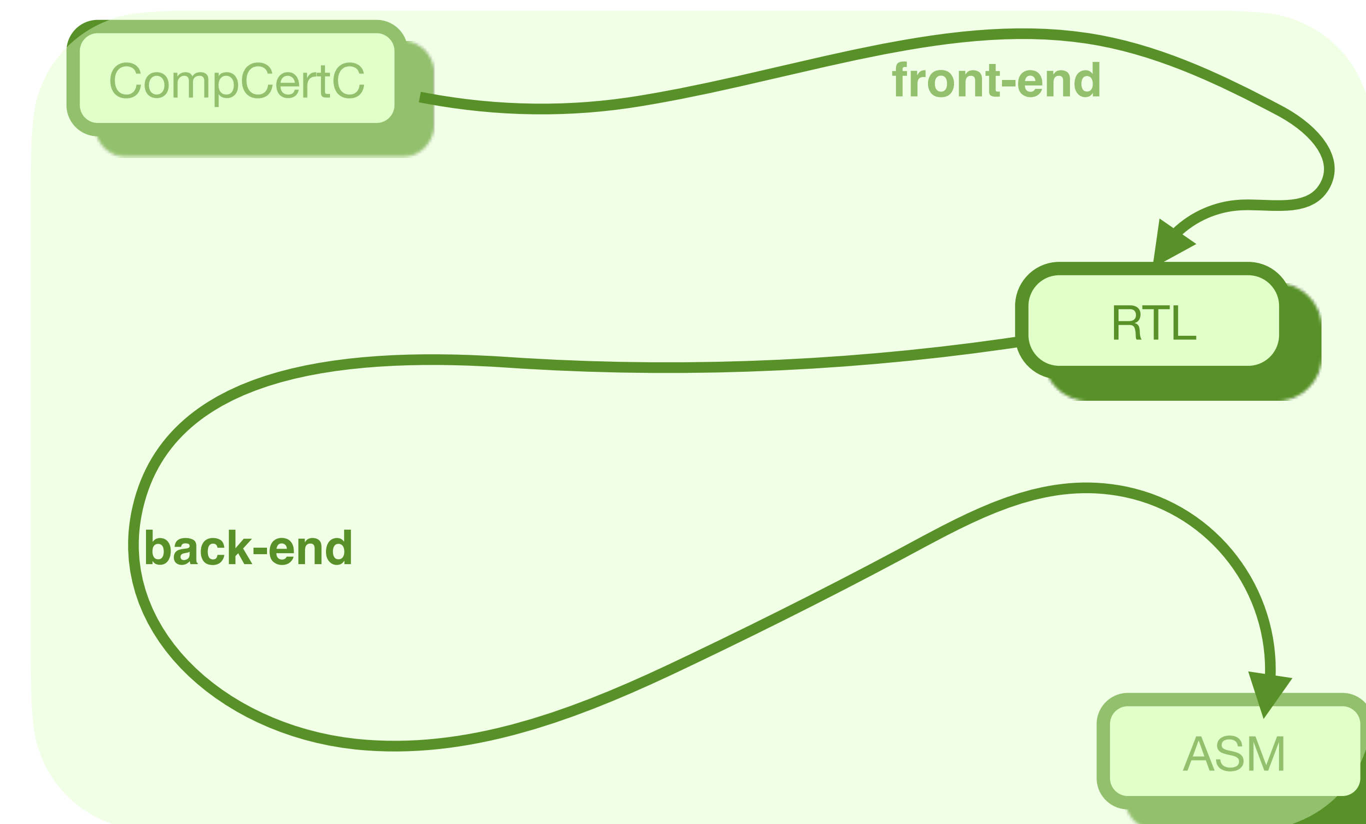
Design new IR and compose the proof for each compiler pass

Extract an OCaml implementation, and compare its efficiency w.r.t. non-verified compilers



CompCert compiler: 10 languages, 18 passes





Proving semantics preservation: the simulation approach

compiler

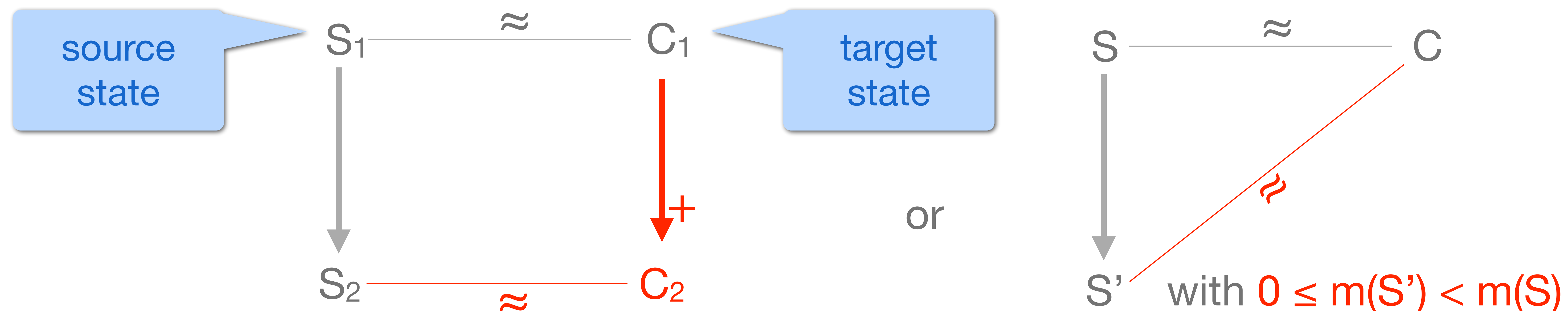
semantics
(execSource, execTarget)

Preserved behaviors = termination and divergence

« The generated code must behave as prescribed by the semantics of the source program. »

```
Theorem compiler-correct:
  ∀ S C b,
  compiler S = OK C →
  execSource S b →
  execTarget C b.
```

Proof technique: simulation diagram



Verified compilation of realistic languages
Lessons learned

Small-step semantics: preferred style for mechanized semantics of compiler verification

Semantic reasoning: necessarily tricky

Proof-engineering considerations

- reuse existing proofs as much as possible
  (e.g. instrument the semantics with leakage traces to ensure a security property)
- split proofs to facilitate their maintenance

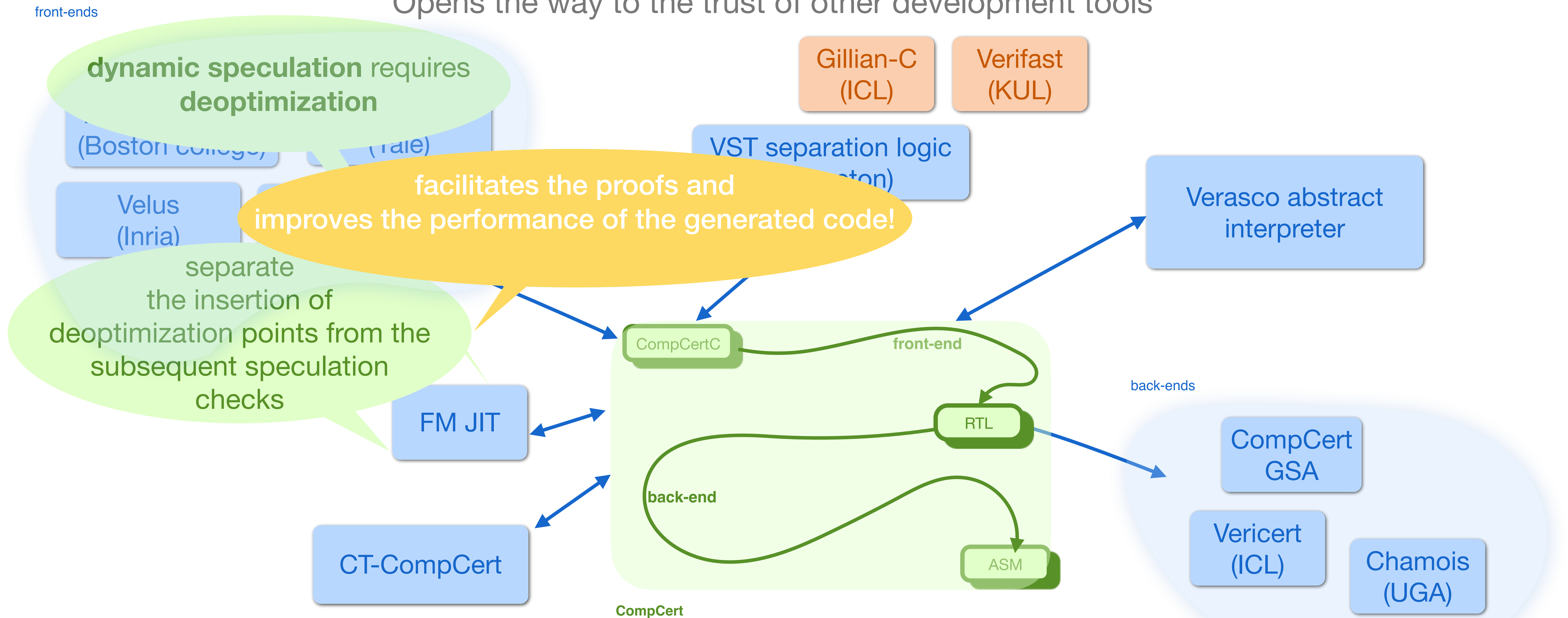
→ Design new IR (including semantic invariants) …

Error prone and time consuming

Testing semantics is mandatory

and replay the proofs many times over

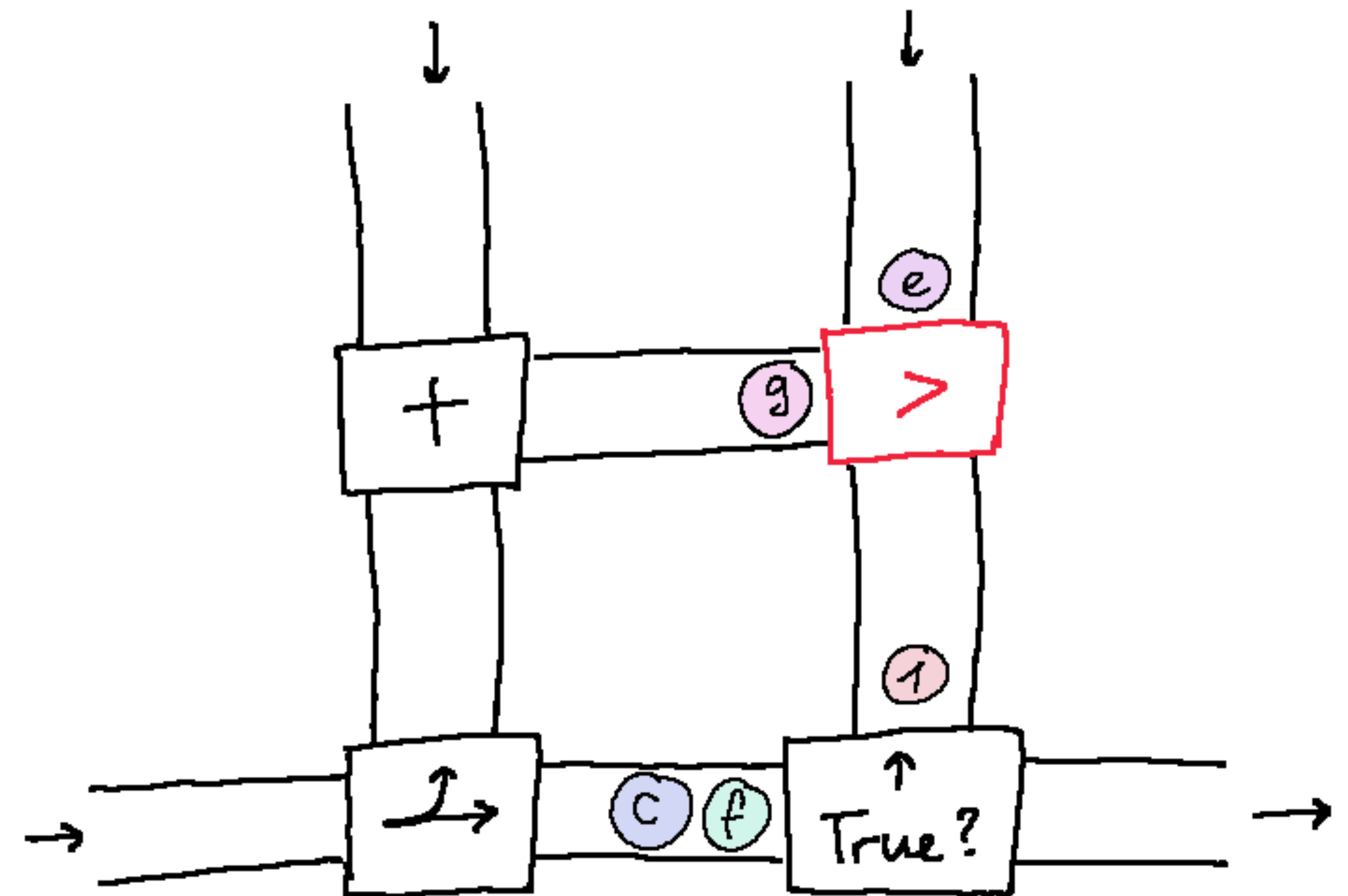
CompCert, an open infrastructure for research



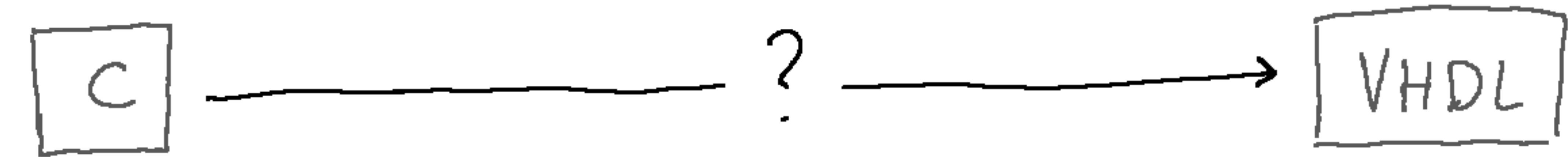
Mechanized semantics are the shared basis for verified compilers, sound program logics, and sound static analyzers

Part 2
Formalization of dataflow circuits

Joint work with Delphine Demange and Tony Law
[OOPSLA 2025]



High-level synthesis (HLS) and scheduling



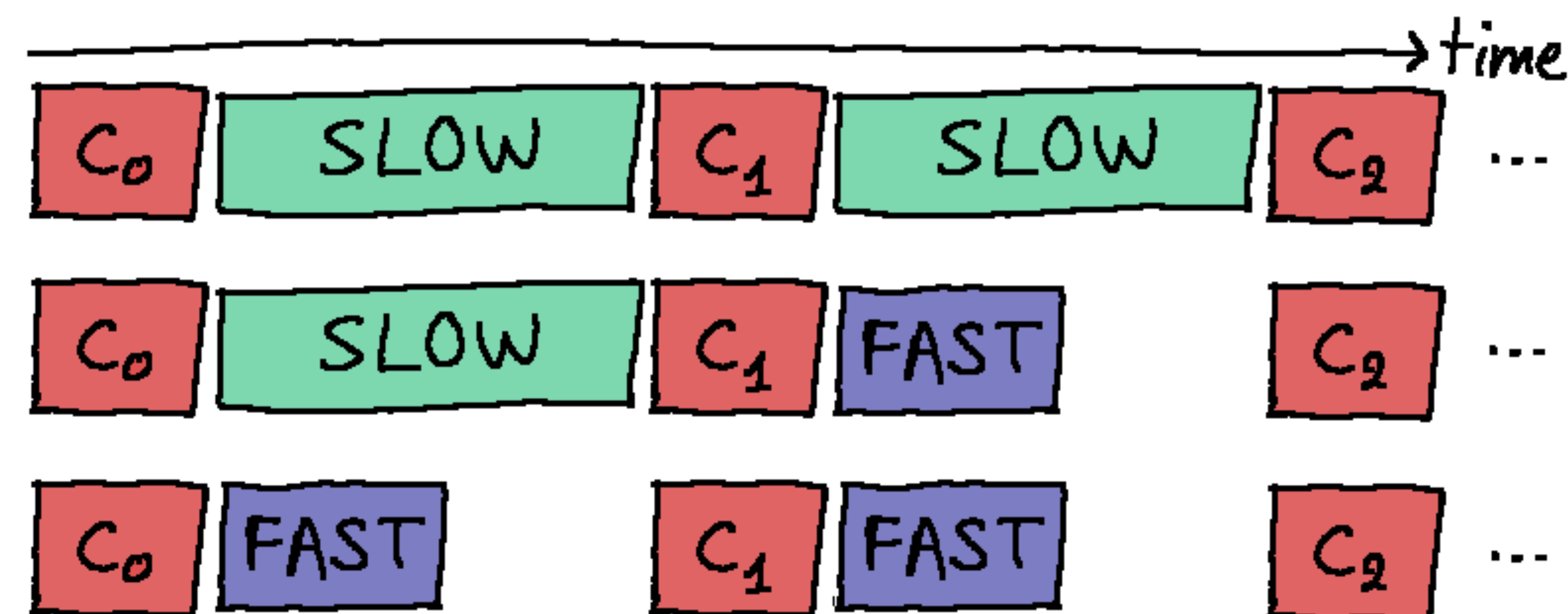
An HLS compiler must

- convert a sequence of instructions into execution units
- schedule their executions

```
for i in 0..N {
  if C[i] {
    SLOW
  } else {
    FAST
  }
}
```

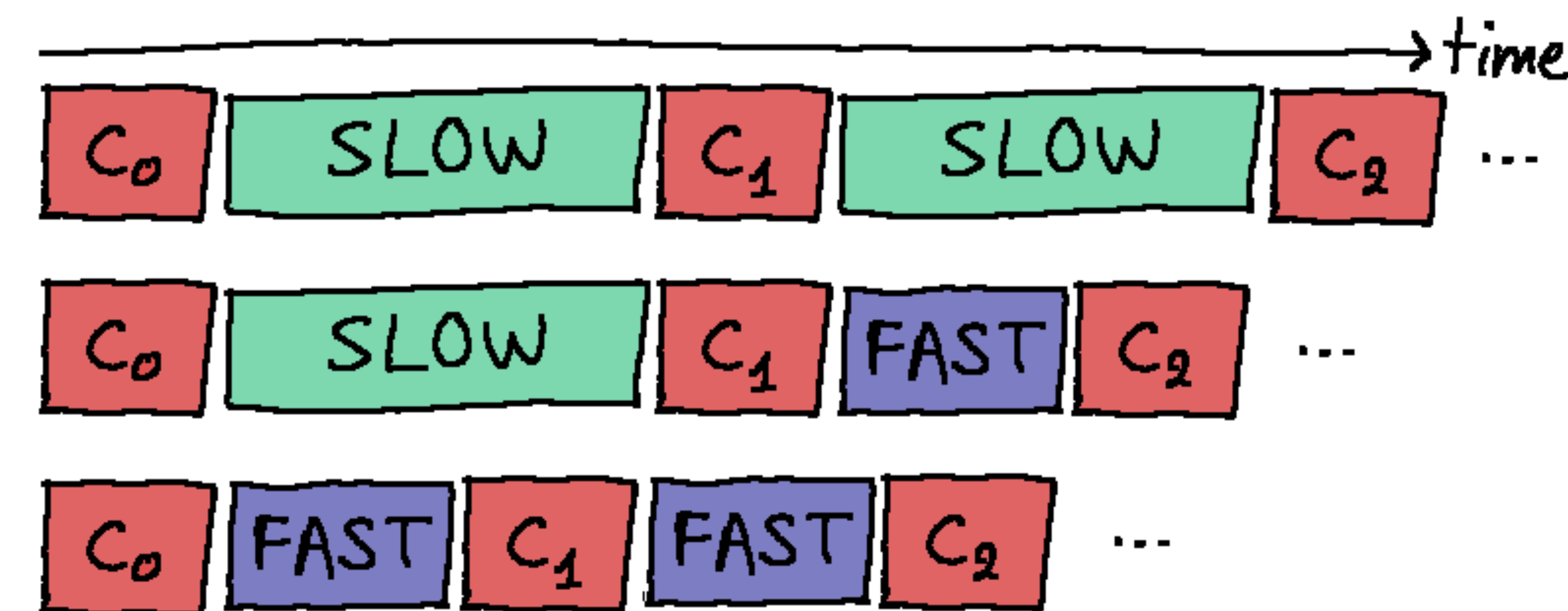
Two scheduling methods: compute a schedule at

compile time (static)



standard optimizations (pipelining, loop unrolling) are applicable, global control-flow is known

run time (dynamic)



reduce idling of computing units, preserving the original control-flow is trickier (unwanted reordering could happen)

Dynamatic: dynamically scheduled compiler

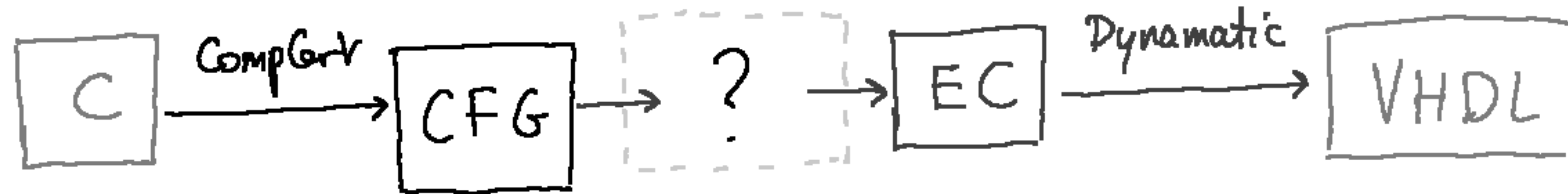
[Josipovic et al., 2018]

Main IR: **elastic circuits**

Network of **elastic components**

- Components consume and produce **tokens**
- Components act **independently**
- Components act when
  - they have **enough** input tokens
  - their successors are **ready** to receive tokens

Our work: a simpler model for elastic circuits



Abstract away hardware specifics

Physical time (clock cycles, latency, communication protocol)

Resource constraints (unbounded number of tokens)

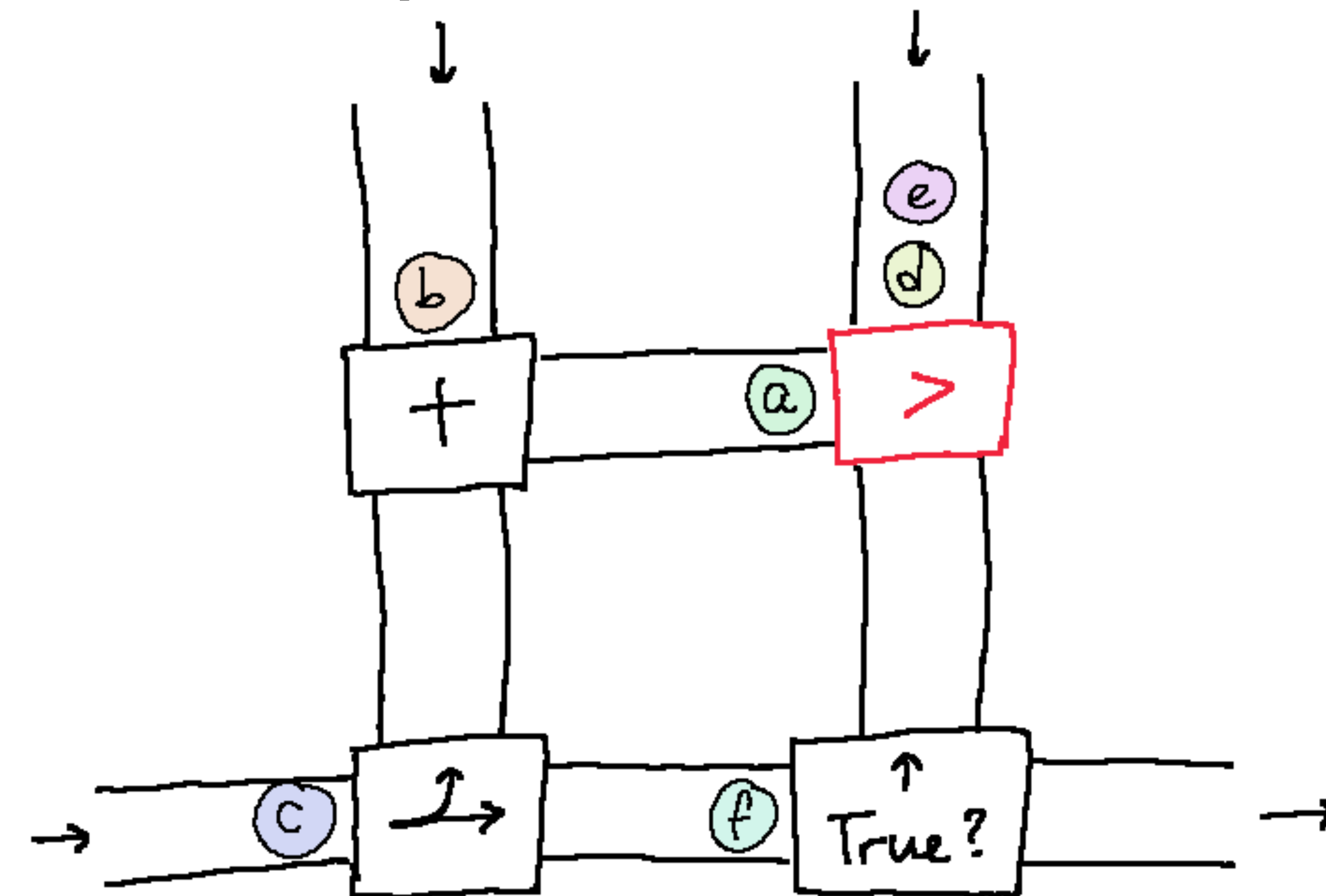
Abstract specification of elastic circuits: asynchronous dataflow circuits

This corresponds to the dataflow execution model

Dataflow execution model [Campbell, Krishna, Ballance 1993]

Network of **components asynchronously** connected by **channels**

Used in architecture, parallel programming



Component

- **independent** computing unit
- consumes / produces tokens

Channel

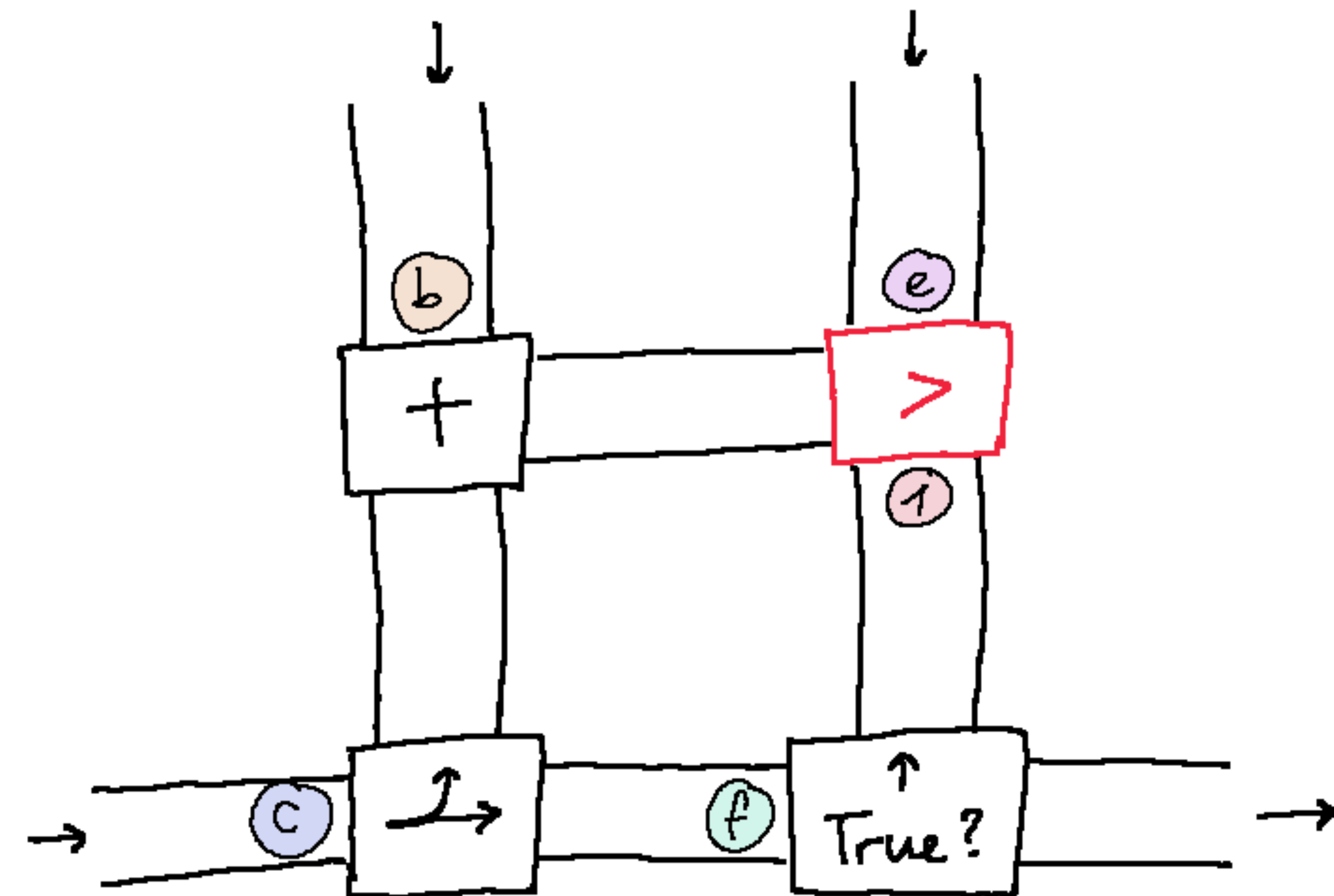
- transfers tokens between components
- **unbounded** queue

Semantics

- execute any component **that has enough inputs**
- scheduling is implementation-dependent

Dataflow execution model [Campbell, Krishna, Ballance 1993]

Network of **components** connected by **channels**



Component

- **independent** computing unit
- consumes / produces tokens

Channel

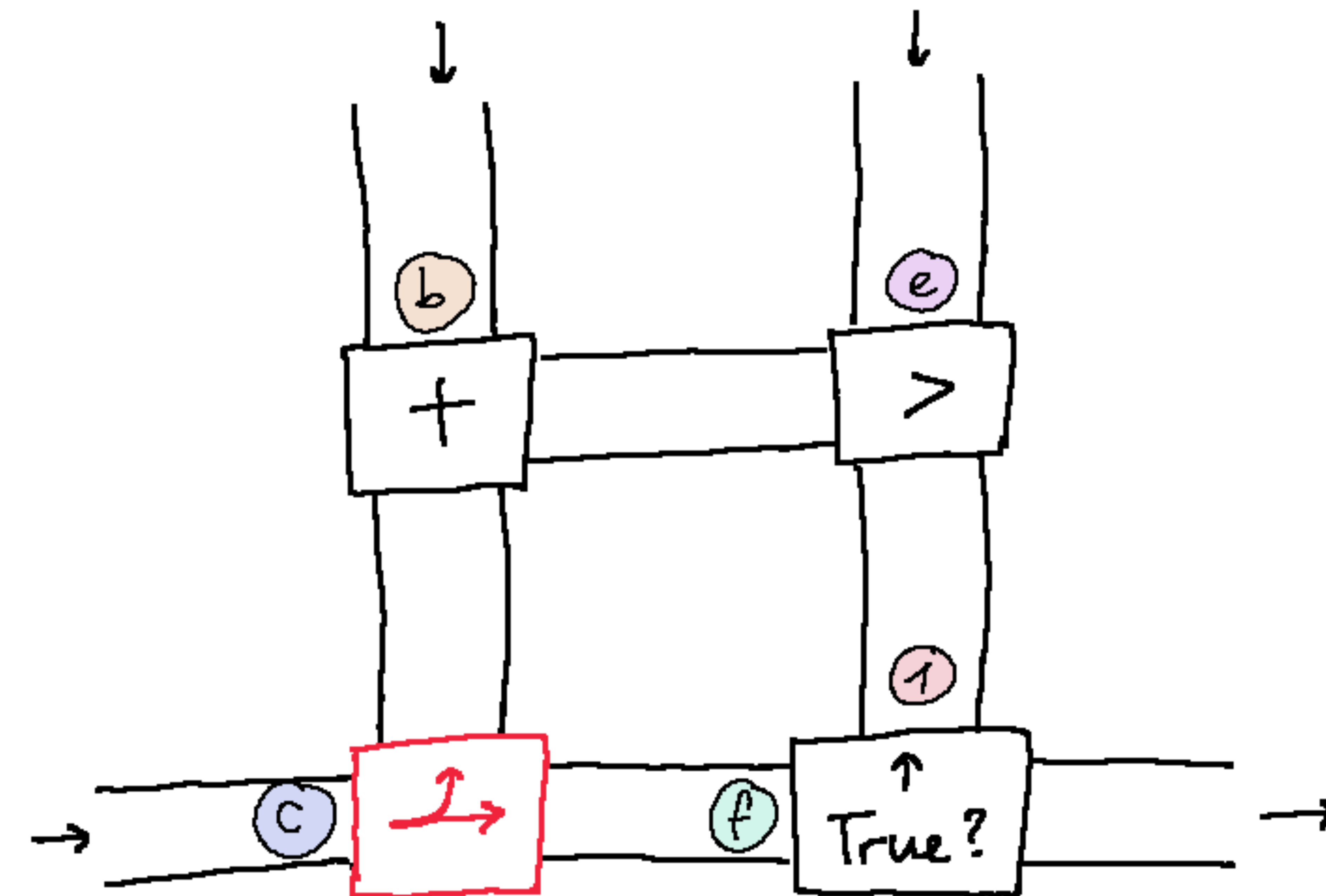
- transfers tokens between components
- **unbounded** queue

Semantics

- execute any component **that has enough inputs**
- scheduling is implementation-dependent

Dataflow execution model [Campbell, Krishna, Ballance 1993]

Network of **components** connected by **channels**



Component

- **independent** computing unit
- consumes / produces tokens

Channel

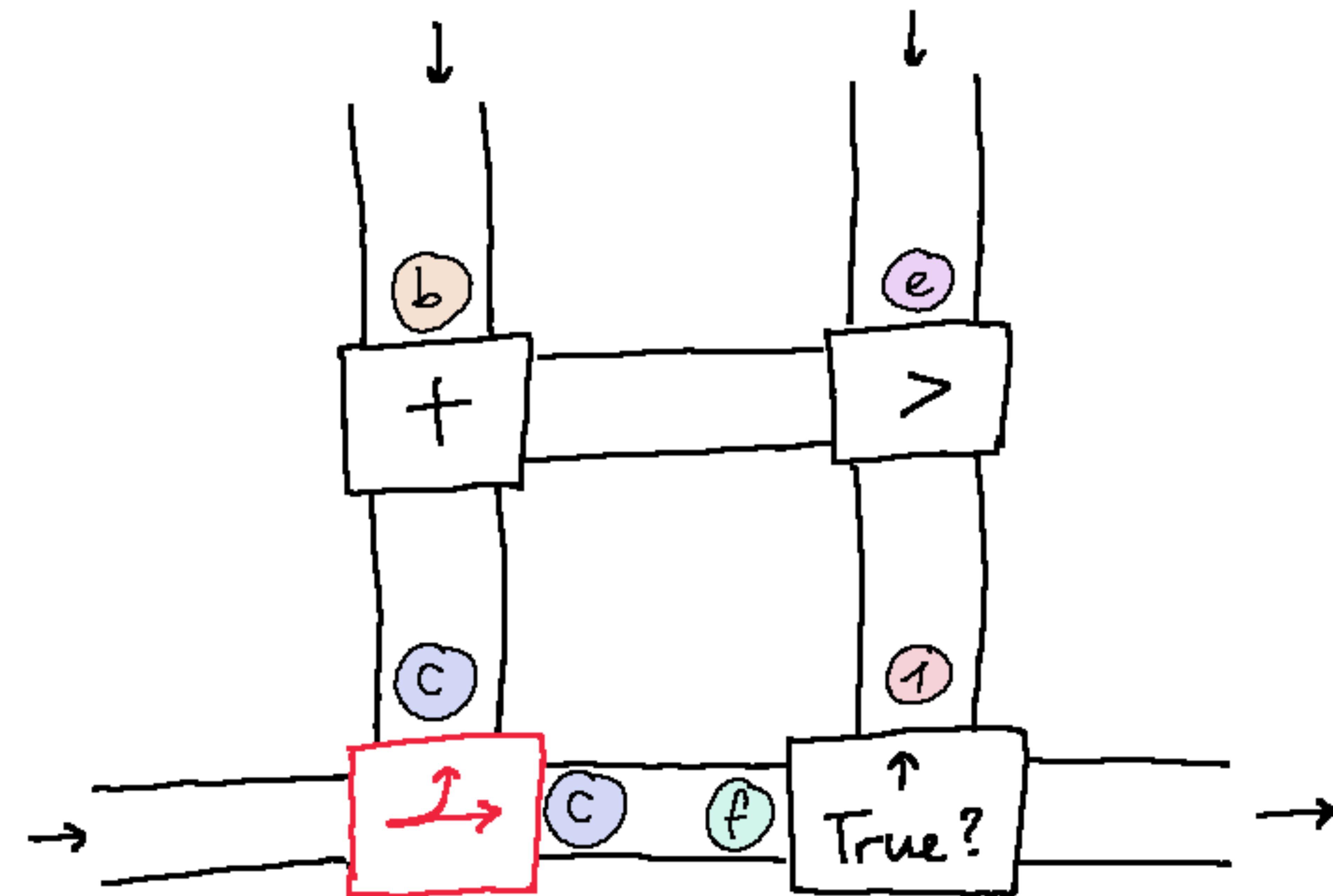
- transfers tokens between components
- **unbounded** queue

Semantics

- execute any component **that has enough inputs**
- scheduling is implementation-dependent

Dataflow execution model [Campbell, Krishna, Ballance 1993]

Network of **components** connected by **channels**



Component

- **independent** computing unit
- consumes / produces tokens

Channel

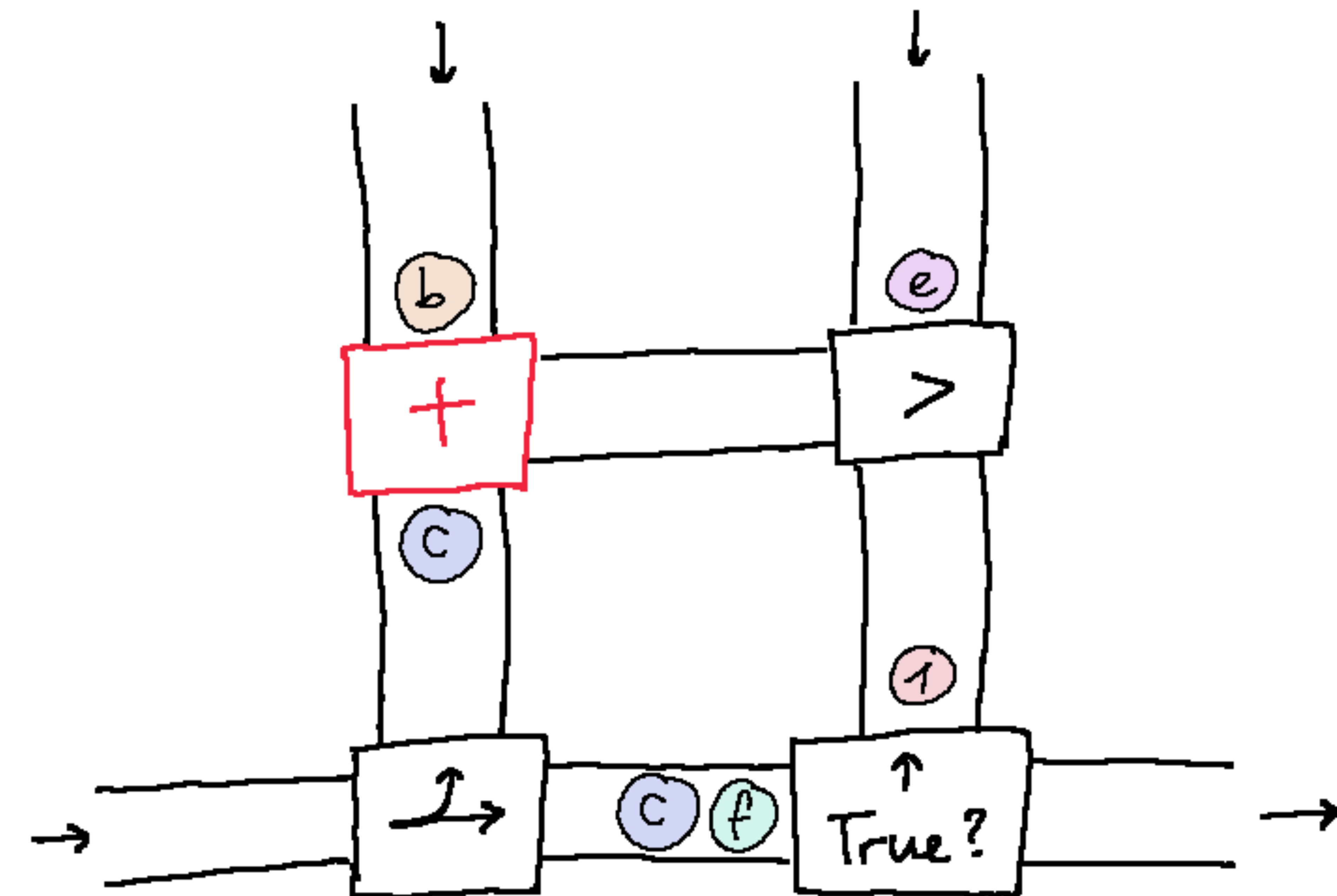
- transfers tokens between components
- **unbounded** queue

Semantics

- execute any component **that has enough inputs**
- scheduling is implementation-dependent

Dataflow execution model [Campbell, Krishna, Ballance 1993]

Network of **components** connected by **channels**



Component

- **independent** computing unit
- consumes / produces tokens

Channel

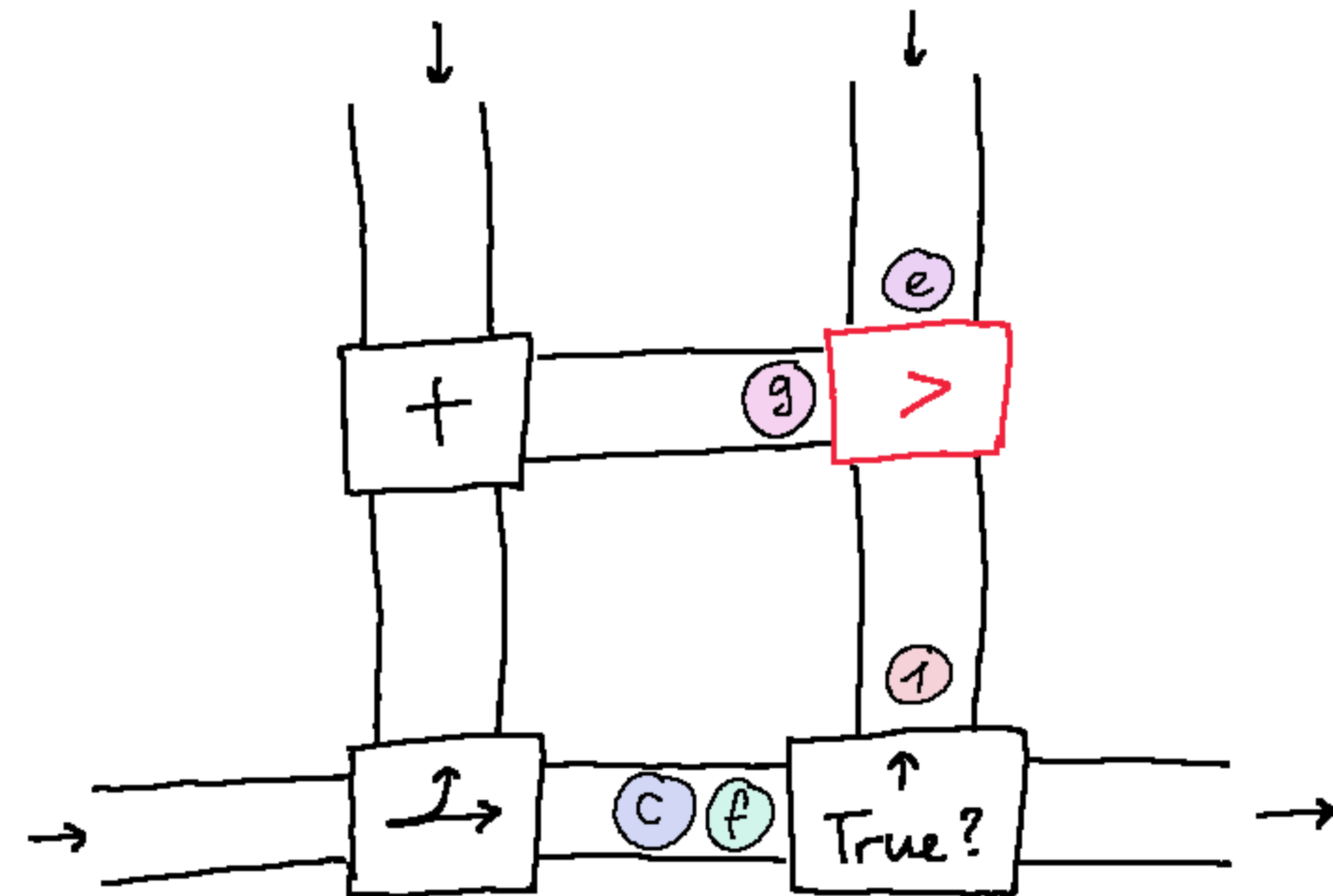
- transfers tokens between components
- **unbounded** queue

Semantics

- execute any component **that has enough inputs**
- scheduling is implementation-dependent

Dataflow execution model [Campbell, Krishna, Ballance 1993]

Network of **components** connected by **channels**



Component

- **independent** computing unit
- consumes / produces tokens

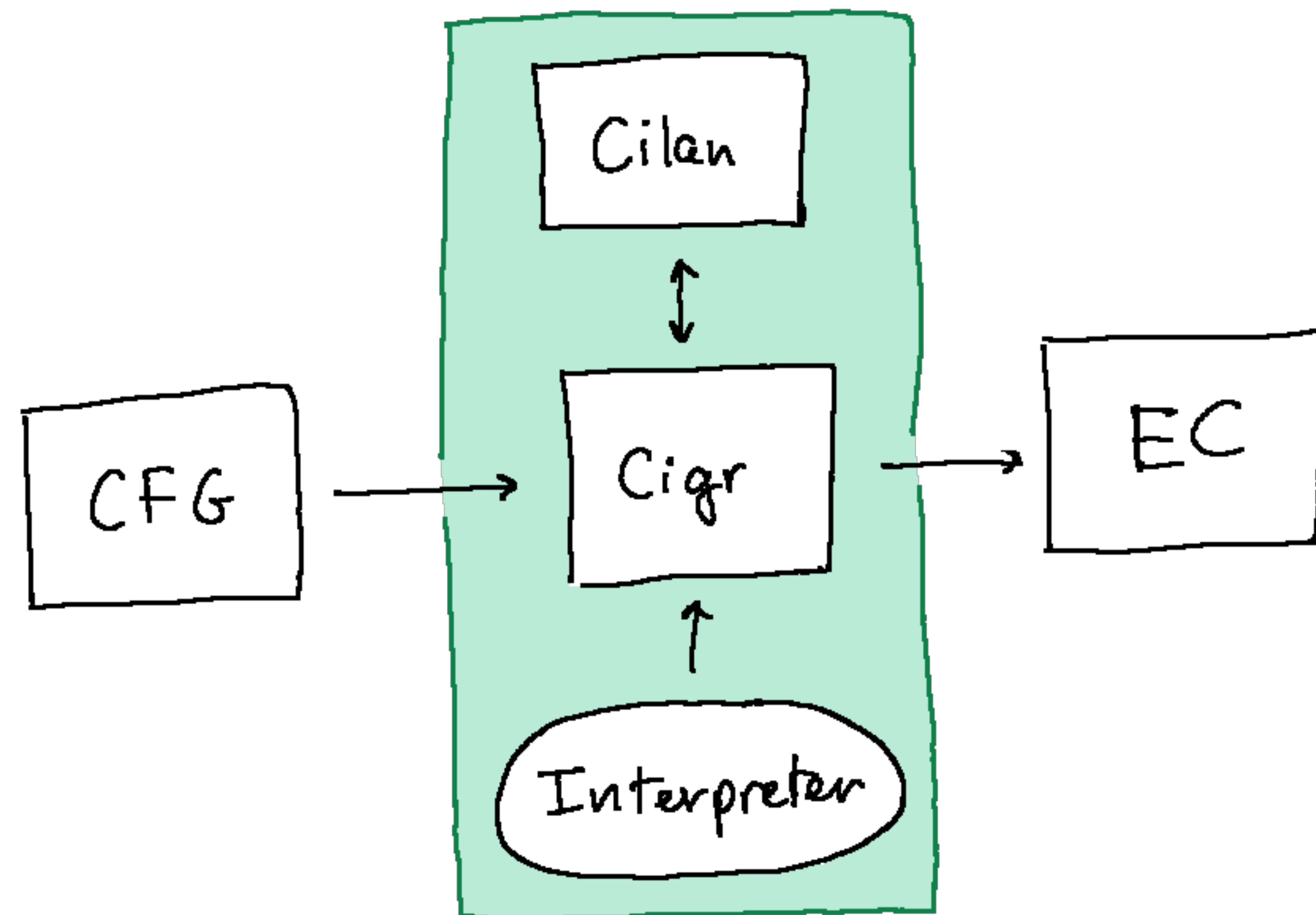
Channel

- transfers tokens between components
- **unbounded** queue

Semantics

- execute any component **that has enough inputs**
- scheduling is implementation-dependent

Semantics of dataflow circuits: our two IR

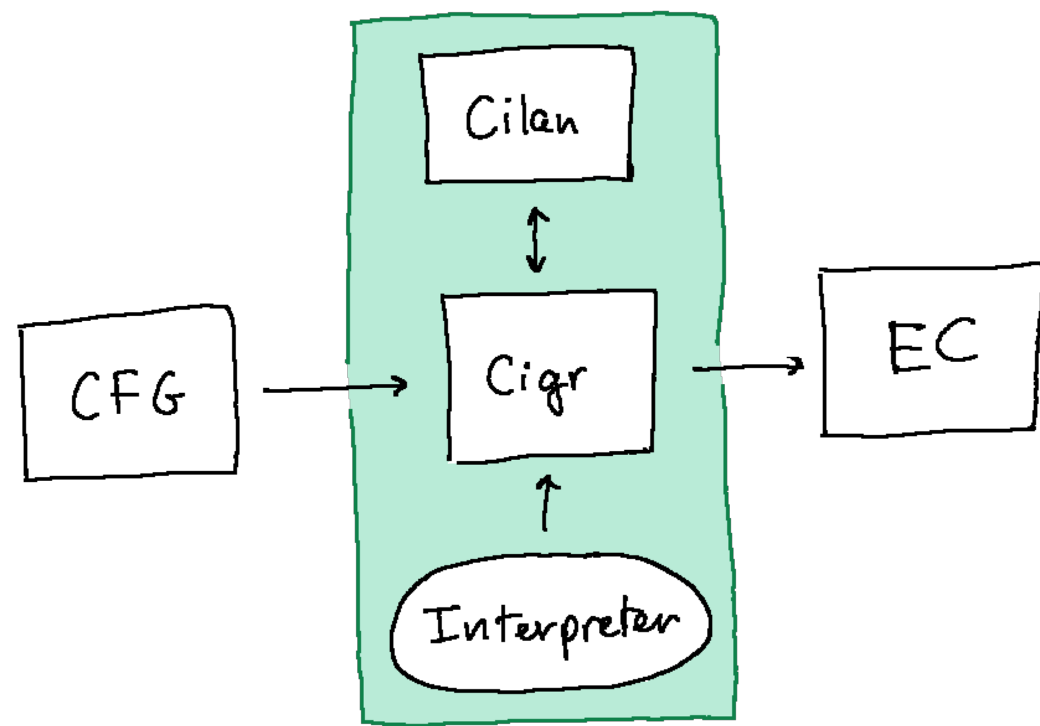


Both fit the dataflow execution model

Each is suited for a specific task: verification and compilation

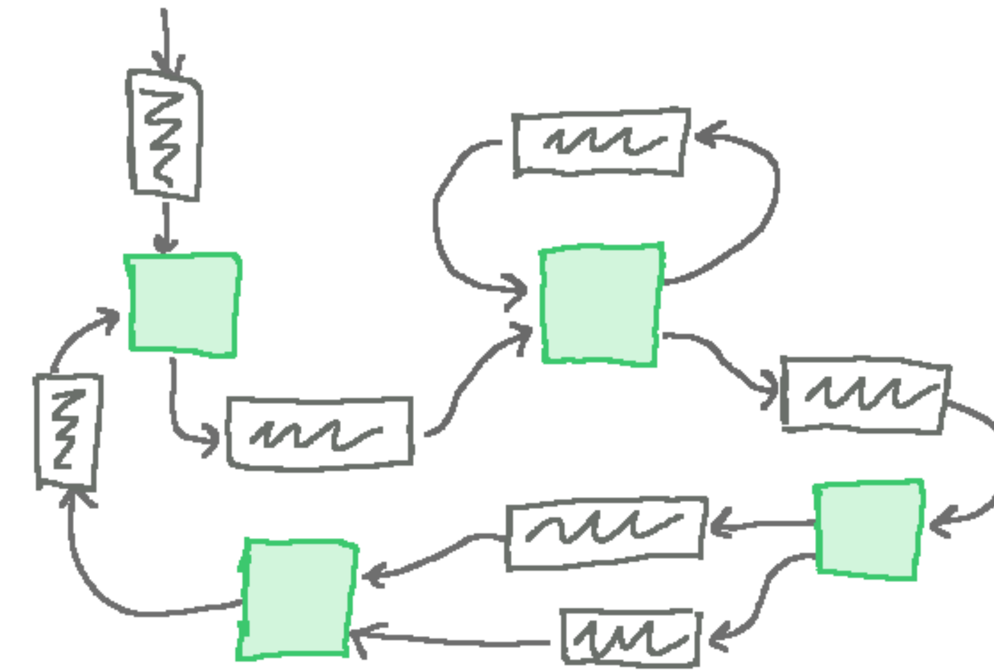
Semantics tested w.r.t. an interpreter (extracted + correct)

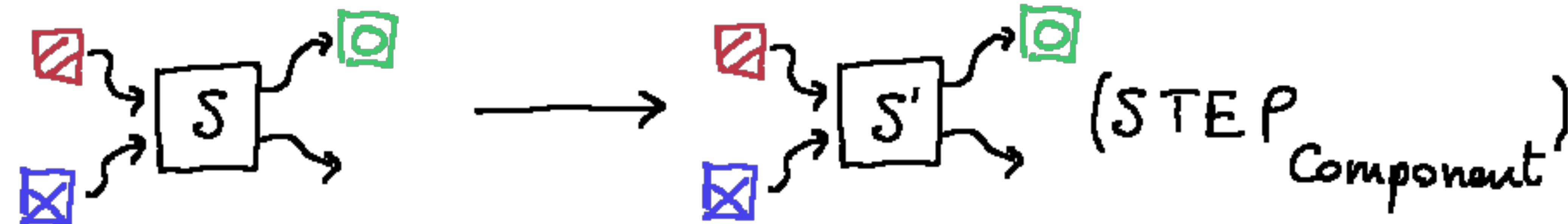
Cigr: circuit graphs
Circuits regarded as a closed system

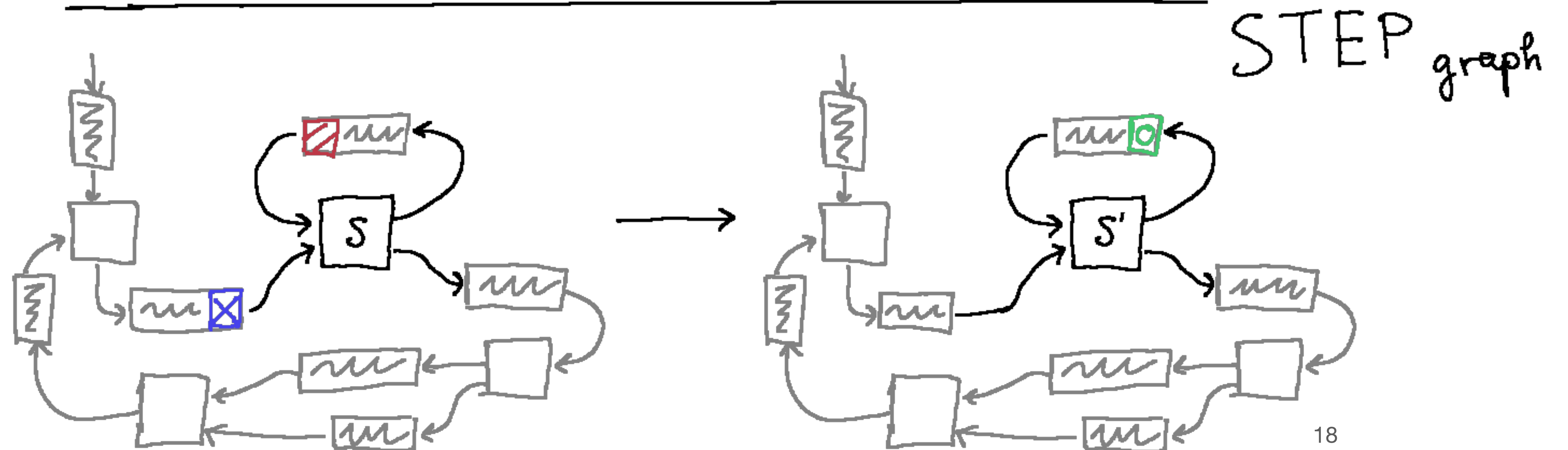


Nodes = components

Edges = channels







Simple semantics

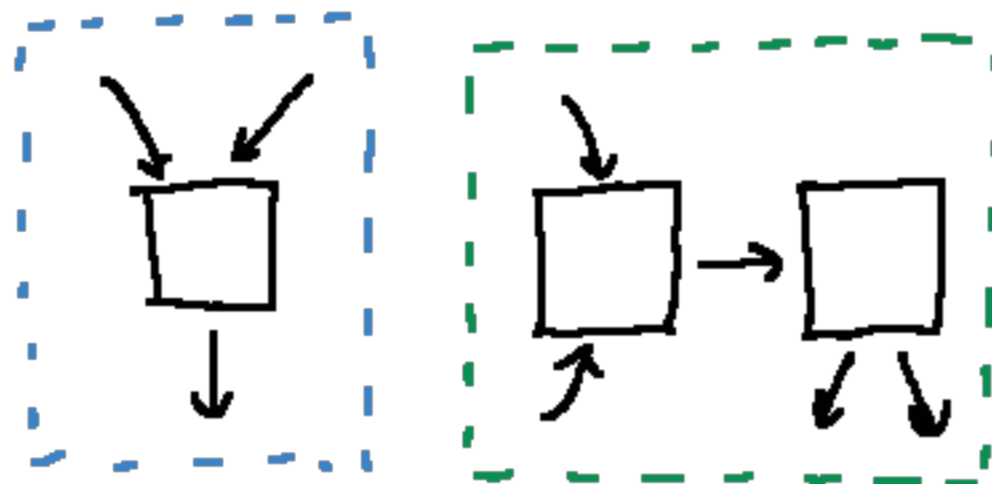
One execution step fires a component + global state update

Simplifying proofs

Intuition: restrict reasonings to one aspect at a time

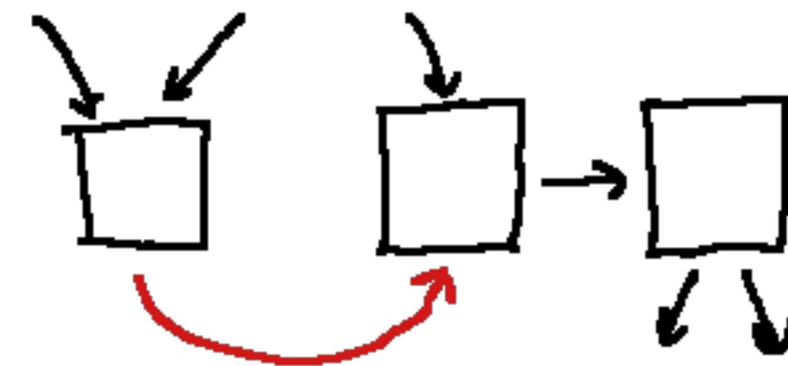
Single component

Parallel composition



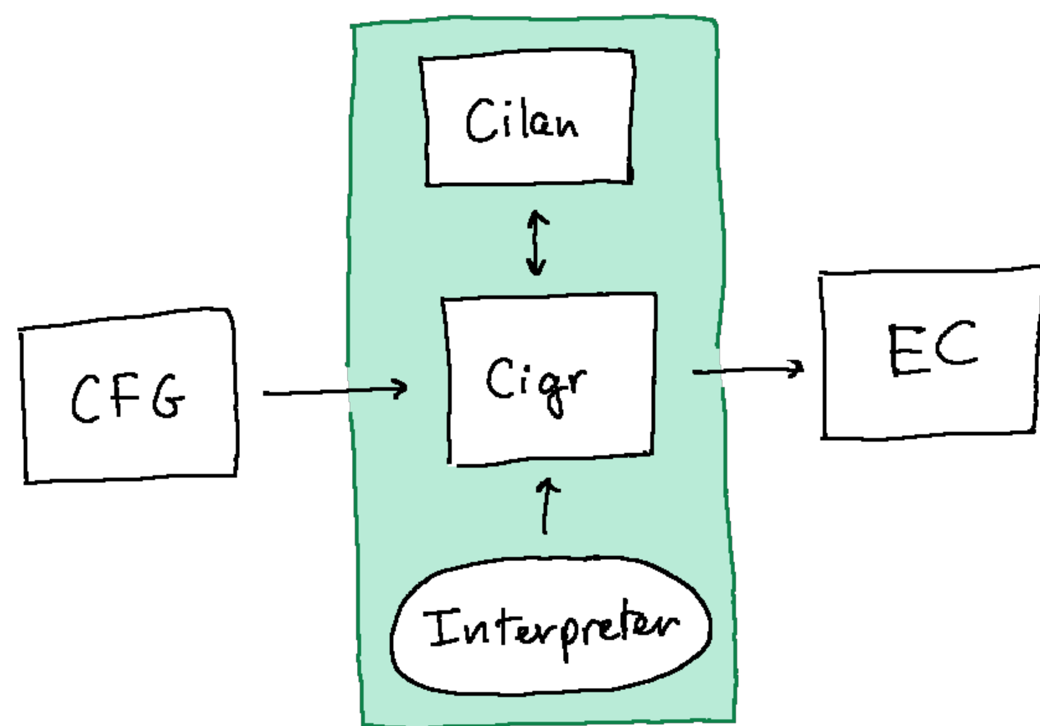
Combination of 2 independent circuits

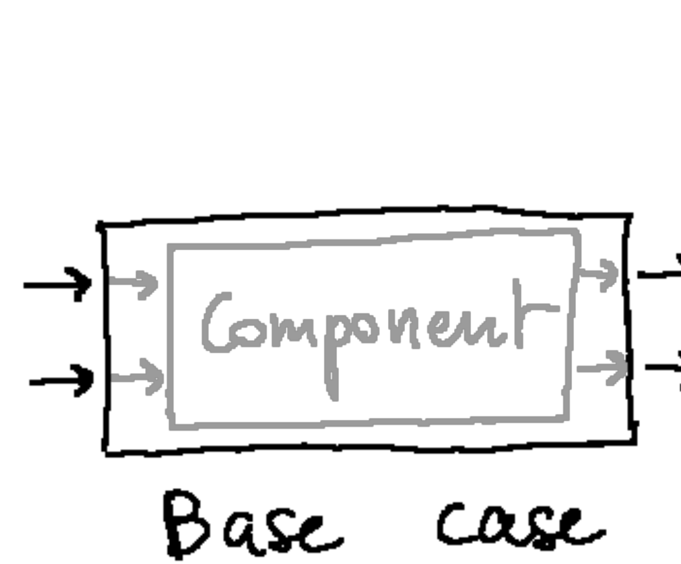
Self composition

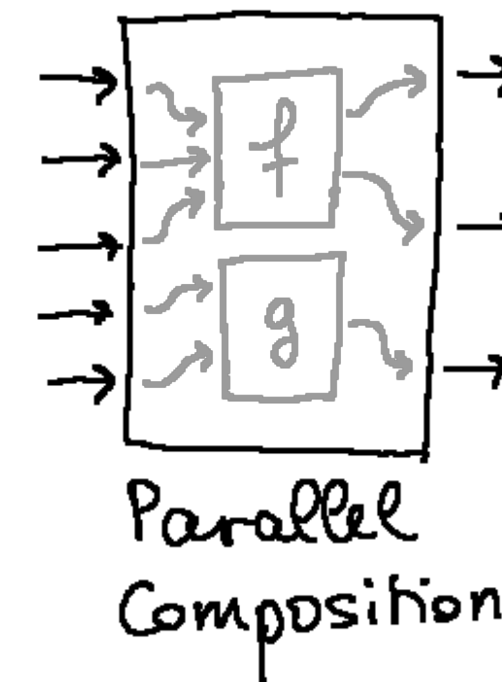


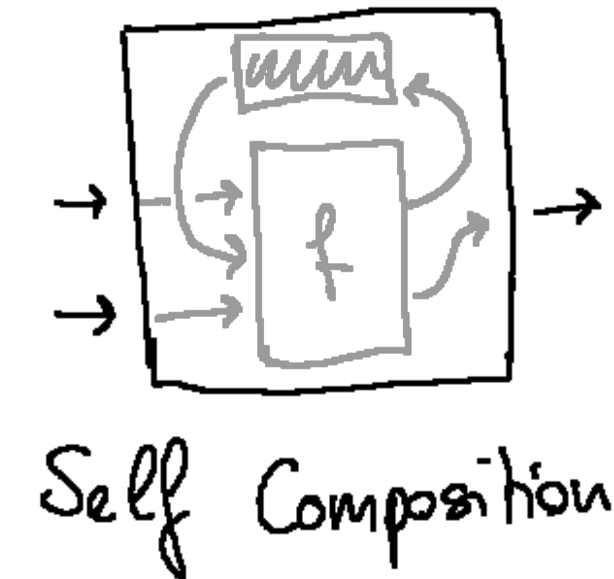
Single channel

Cilan: circuit language
Compositional circuits regarded as components

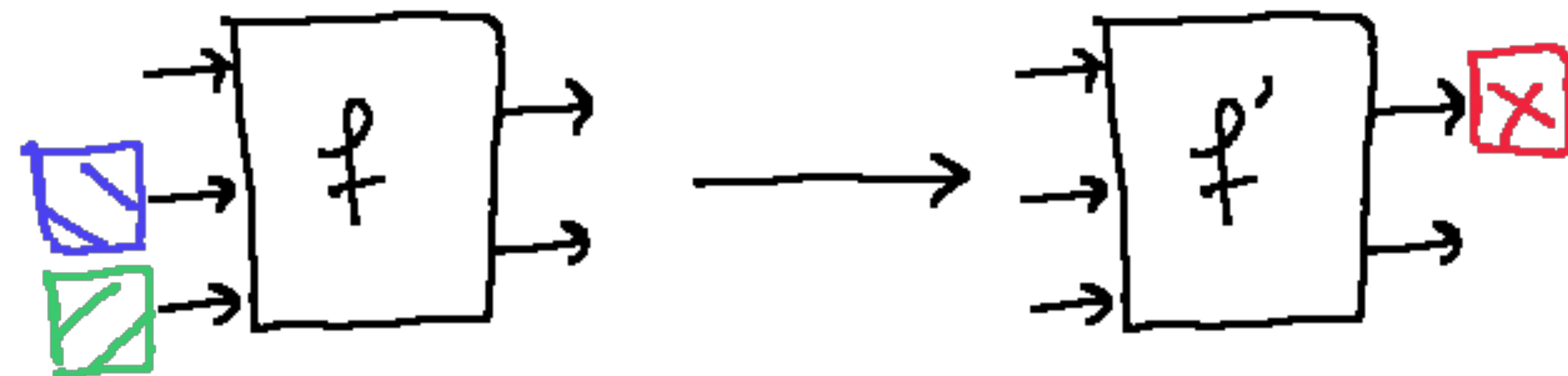




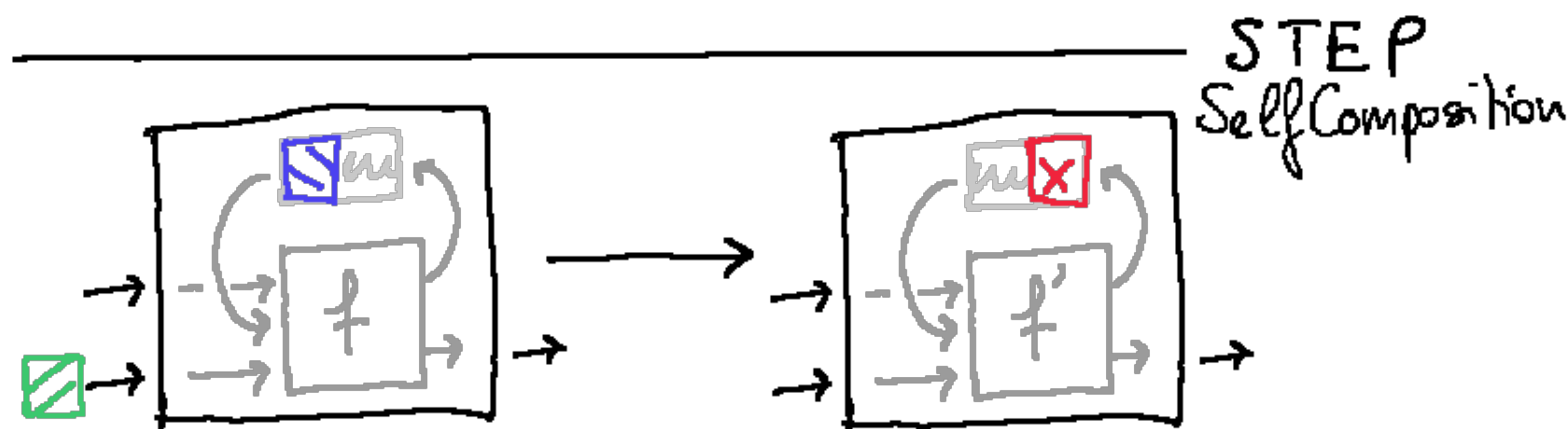




Language defined inductively



Structured term calculus, which enables inductive reasoning on circuits



Example rule for the connecting channel

Reasoning on Cilan

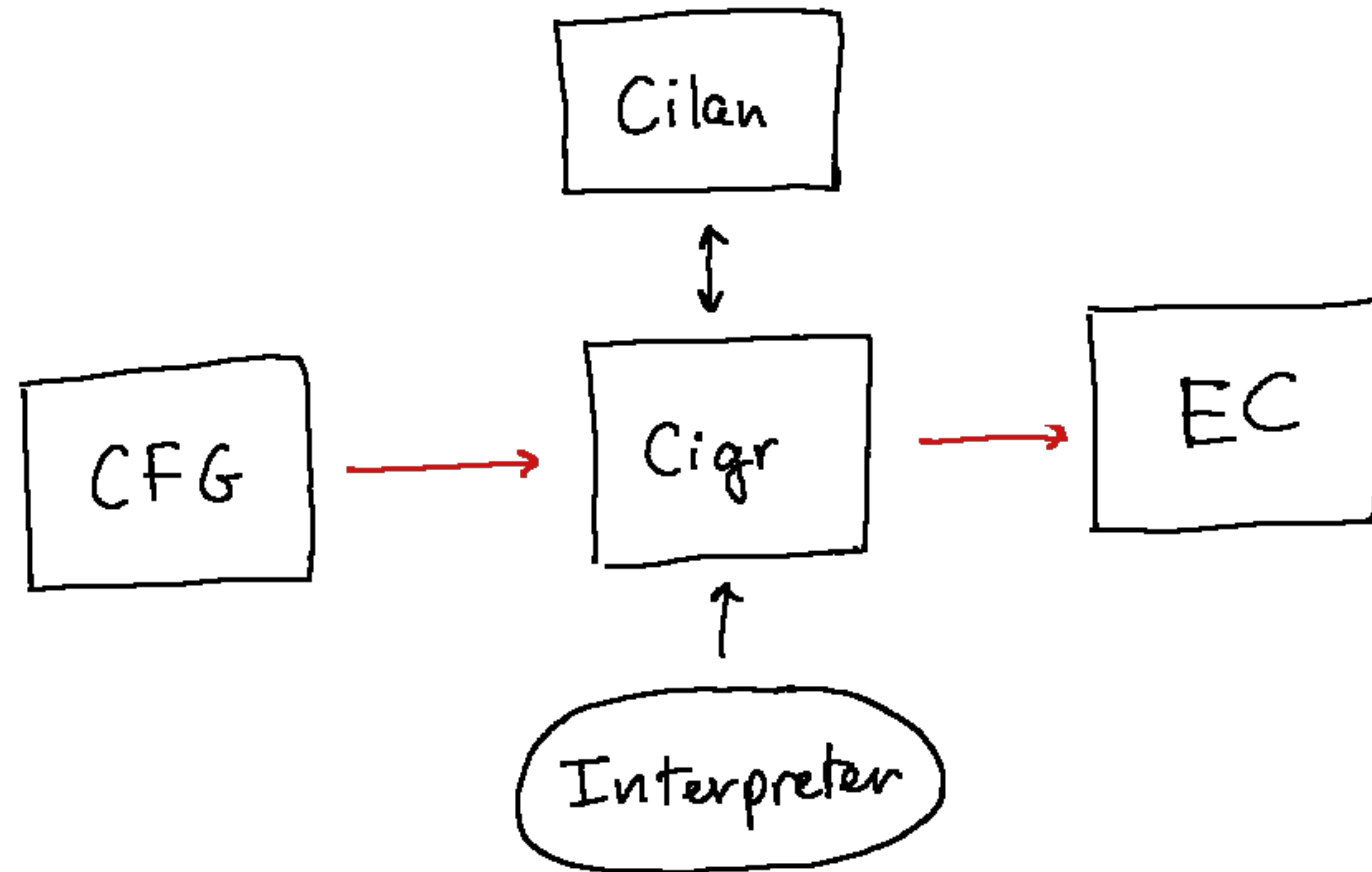
Two **equivalent** IR: each property proved on an IR also holds on the other one

Operations on circuits are triggered in a non-deterministic way
(for well-formed components)

Main property: **determinacy** of circuit executions
Theorem determinacy: regardless of the schedule of components, a circuit should exhibit a single observable behavior

- Lemma order irrelevance: the order in a schedule should not matter
- Lemma trace determinacy: tokens are produced in the same order regardless of the schedule

WIP



Questions ?