Algebraic Reasoning on Graphical Models: the Case for String Diagrams

Fabio Zanasi

UCL

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

In a Nutshell

- There is a long-established tradition of understanding **programming languages** in terms of *algebraic structures*.

- Non-classical paradigms of computation (concurrent, probabilistic, quantum, …) demand a resource-sensitive representation, often expressed via **graphical models**.

- **String diagrams** have emerged in the last two decades as a formalism in which graphical models can be analysed *as if they were programming languages*.

- This talk is an introduction to string diagrams.

Part I:

The Algebraic Structure of Programming Languages

Programming Language Theory

```python
import random

guess = eval(input("Enter a three digit number: "))
guessD3 = str(guess % 10)
guessD2 = str((guess // 10) % 10)
guessD1 = str(guess // 100)

number = random.randrange(100, 1000)
numberD3 = str(number % 10)
numberD2 = str((number // 10) % 10)
numberD1 = str(number // 100)

print("The lottery number is", number)

if number == guess:
    print("You have won $ 10, 000.")

elif number == int(guessD1 + guessD3 + guessD2) or number ==
    or number == int(guessD2 + guessD3 + guessD1) or number
    or number == int(guessD3 + guessD2 + guessD1):
    print("You have won $ 3, 000.")
```

```scala
import java.io.{BufferedReader,
InputStreamReader}
object Main {
  def main(args: Array[String]) {
    System.out.println("Hello, World!");
    var reader = new BufferedReader(new
InputStreamReader(System.in));
    var n = Integer.parseInt( reader.readLine());
    for (i <- 0 to n) {
      var r = Math.exp(Math.log(2)*i);
      System.out.print("%.0f ".format(r));
    }
  }
}
```

```java
package com.nakotojava.intro;

import java.util.logging.Logger;

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class PersonTest {

  @Test
  public void testPerson() {
    Person p = new Person("Joe Q Author", 42, 17
    Logger l = Logger.getLogger(Person.class.getN
    l.info("Name: " + p.getName());
    l.info("Age:" + p.getAge());
    l.info("Height (cm):" + p.getHeight());
    l.info("Weight (kg):" + p.getWeight());
    l.info("Eye Color:" + p.getEyeColor());
    l.info("Gender:" + p.getGender());
    assertEquals("Joe Q Author", p.getName());
    assertEquals(42, p.getAge());
    assertEquals(173, p.getHeight());
    assertEquals(82, p.getWeight());
    assertEquals("Brown", p.getEyeColor());
    assertEquals("MALE", p.getGender());
  }
```

Can we study programming languages **uniformly**?

Can we **mathematically describe** their similarities and differences?

Can we give a **formal proof** of the fact that they are terminating, correct, trustworthy, etc.?

Can we identify which are `**good**' principles in language **design**?

Programming Language Theory

Syntax

Rules to write a program.

Semantics

Mathematical description of what the program does

Program Syntax

$$
\begin{array}{lllllllllll}
 & b & ::= & True & | & x = y & | & x = n & | & \neg b & | & b \wedge b & | & b \vee b \\
p & ::= & skip & | & x := y & | & x := s(y) & | & x := n & | & while\ b\ do\ p & | & p\ ;\ p
\end{array}
$$

Example:

$$x := 0\ ;\ y := 5\ ;\ while\ (\neg\ x{=}y)\ do\ x := s(x)$$

Why Syntax is Good

Compositional Semantics

Syntax

Semantics

```
var n = Integer.parseInt( reader.readLine());
for (i <- 0 to n) {
  var r = Math.exp(Math.log(2)*i);
  System.out.print("%.0f ".format(r));
}
```

$$f : A \to B$$

```
var n = Integer.parseInt( reader.readLine());
for (i <- 0 to n) {
  var r = Math.exp(Math.log(2)*i);
  System.out.print("%.0f ".format(r));
}
```

$$g : A \to C$$

$$h : C \to B$$

$$f = h \circ g : A \to C$$

Why Syntax is Good

Definitions and proofs by induction

Induction on natural numbers

$$n \quad ::= \quad 0 \mid n+1$$

$$\begin{array}{lcl} b & ::= & True \mid x = y \mid x = n \mid \neg b \mid b \wedge b \mid b \vee b \\ p & ::= & skip \mid x := y \mid x := s(y) \mid x := n \mid while\ b\ do\ p \mid p\,;\,p \end{array}$$

A property Q is true for every program if:

- Q is true for ***skip***, ***x:=y***, ***x:=s(y)***, ***x:=n***.
- Assuming Q is true for programs ***p***, ***p'***,
  Q is true for ***while b do p*** and for ***p;p'***.

Why Syntax is Good

Identify the algebraic laws governing program behaviour

$$
\begin{array}{lllllllllllll}
 & b & ::= & True & | & x = y & | & x = n & | & \neg b & | & b \wedge b & | & b \vee b \\
p & ::= & skip & | & x := y & | & x := s(y) & | & x := n & | & while\ b\ do\ p & | & p\,;\,p
\end{array}
$$

; is associative.

$$[\![(p_1; p_2); p_3]\!] = [\![p_3]\!] \circ [\![p_2]\!] \circ [\![p_1]\!] = [\![p_1; (p_2; p_3)]\!]$$

(Prog, skip, ;) is a monoid.

$$[\![p; skip]\!] = [\![p]\!] = [\![skip; p]\!]$$

…

Programs with the same semantics may be treated as **equivalent** for certain tasks. **Rewriting** of programs into equivalent ones may be used for various purposes, e.g. optimisation.

Summing up

Syntax and semantics are the lenses through which we rigorously define a programming language.

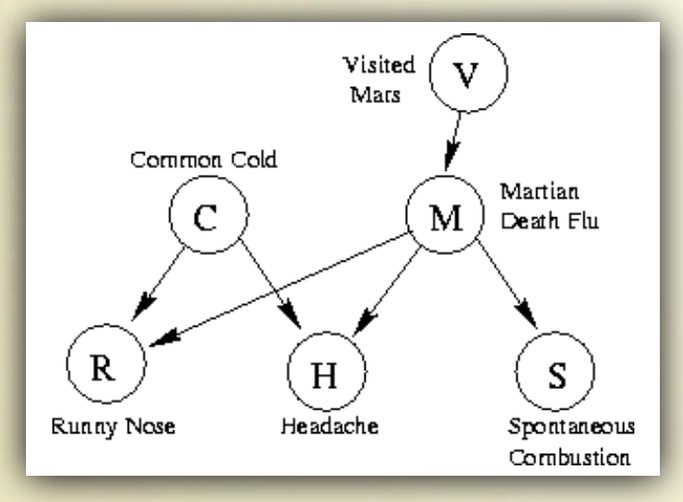
This perspective gives us powerful tools of analysis (**algebraic reasoning**) and fundamental tenets (**compositionality**).

Part II:

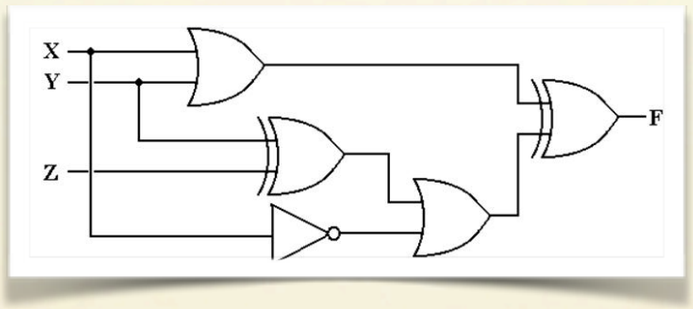
From Graphical Models to String Diagrams

Graphical Models

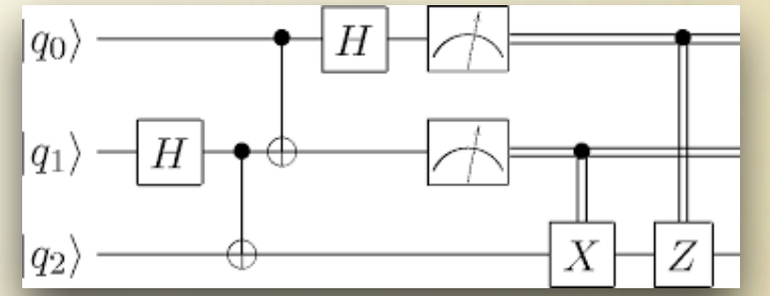
Bayesian networks



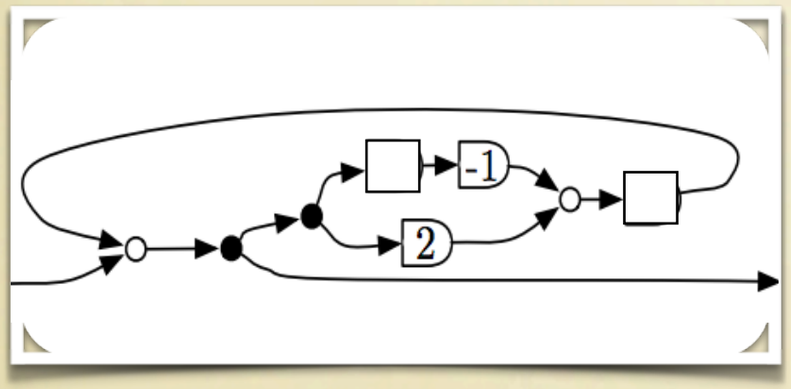
Digital Circuits



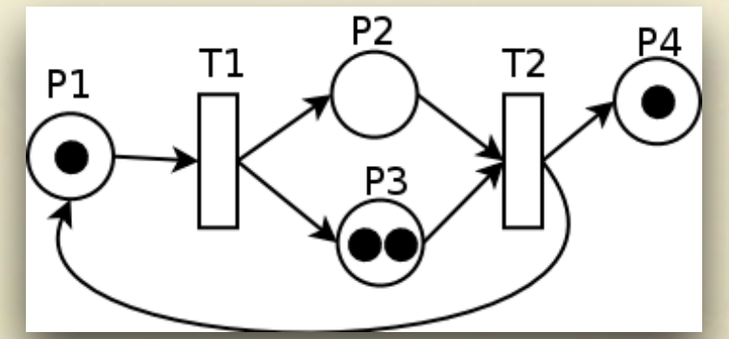
Quantum circuits



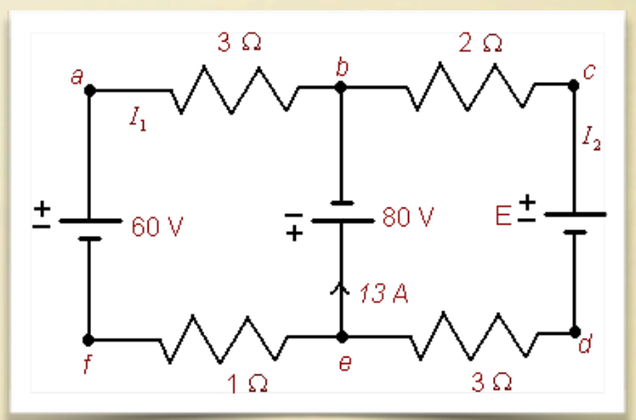
Signal flow graphs



Petri nets



Electrical Circuits



Key Ideas

Graphical models become **syntax**
when formalised as **string diagrams**

We can then analyse them using the methodologies
of programming language theory.

Semantics

Equational reasoning

Completeness

Full Abstraction

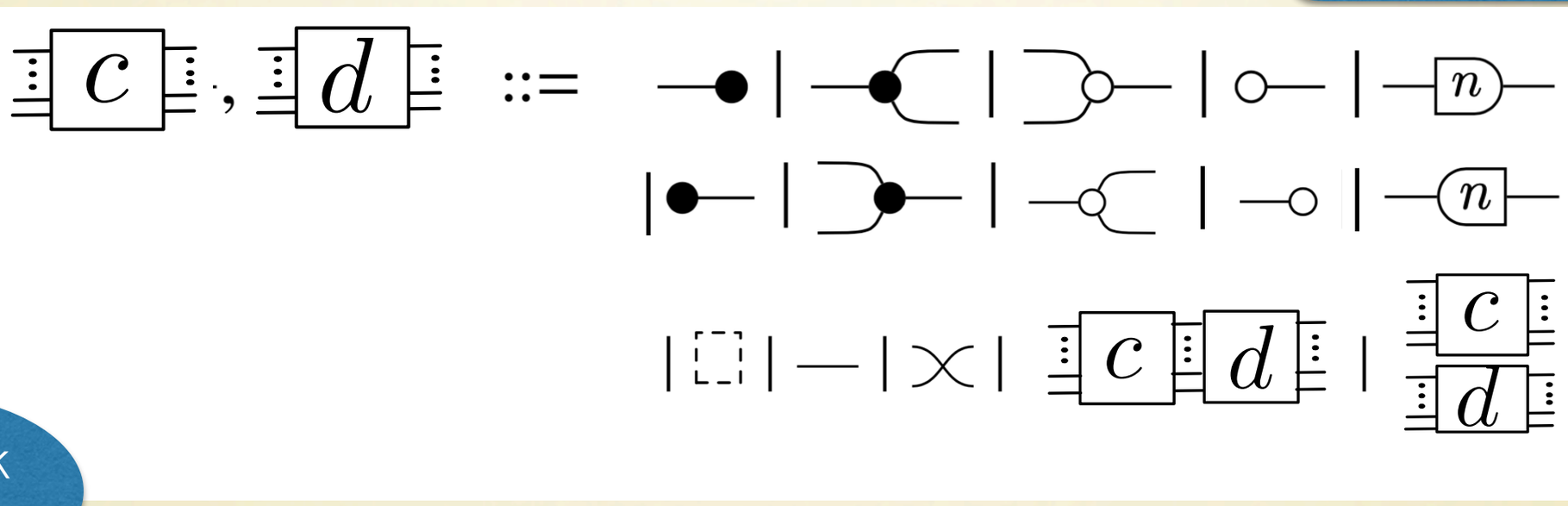
Rewriting

…

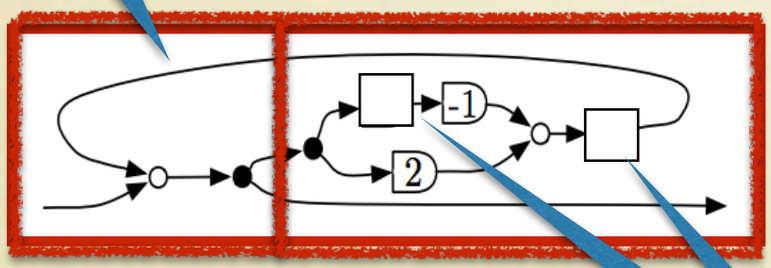
Example: Signal Flow Graphs

String Diagrammatic Syntax

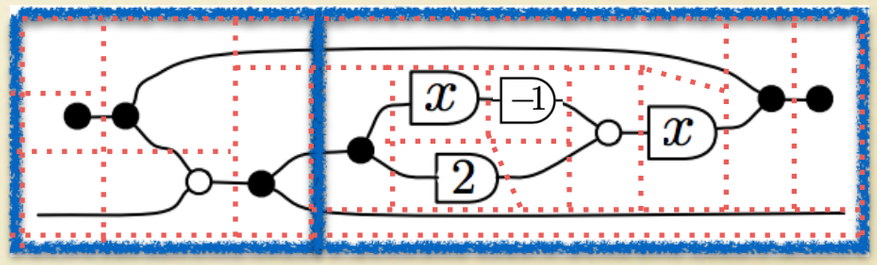
$n \in R$
In this case, $R = \mathbb{R}[x]$



feedback loop



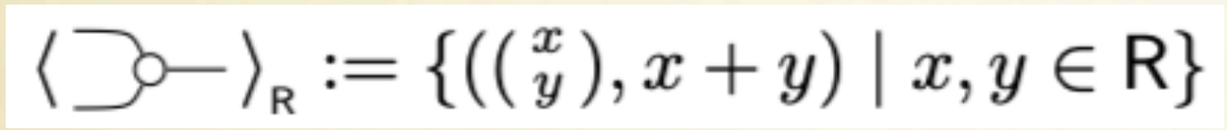
registers



? ? R_1 ; R_2 = R

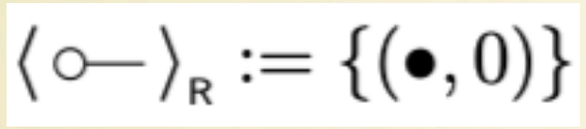
Signal Flow diagrams: semantics

The semantics of a signal flow diagram is a relation over R-vectors



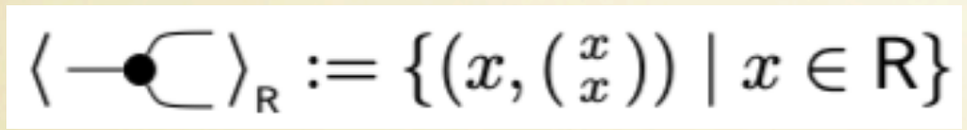
$$\langle \rangle_{\mathsf{R}} := \{((\tbinom{x}{y}), x+y) \mid x, y \in \mathsf{R}\}$$

Addition



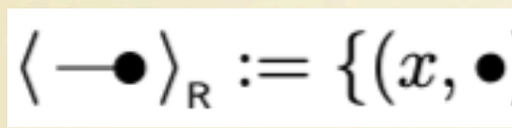
$$\langle \rangle_{\mathsf{R}} := \{(\bullet, 0)\}$$

Zero



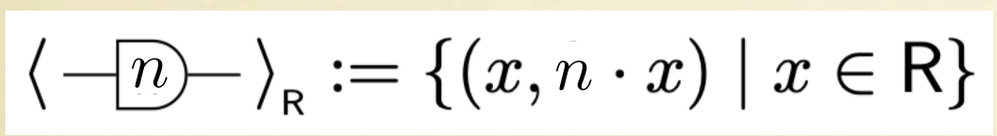
$$\langle \rangle_{\mathsf{R}} := \{(x, (\tbinom{x}{x})) \mid x \in \mathsf{R}\}$$

Duplicate



$$\langle \rangle_{\mathsf{R}} := \{(x, \bullet) \mid x \in \mathsf{R}\}$$

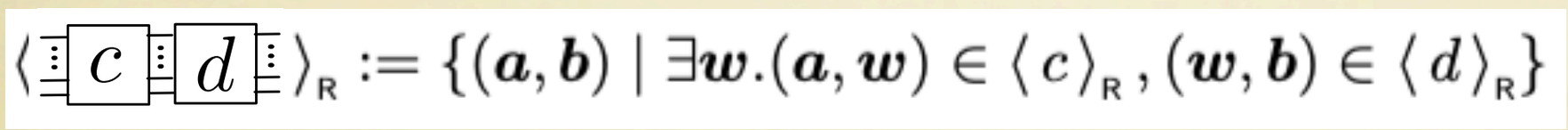
Discard



$$\langle \rangle_{\mathsf{R}} := \{(x, n \cdot x) \mid x \in \mathsf{R}\}$$

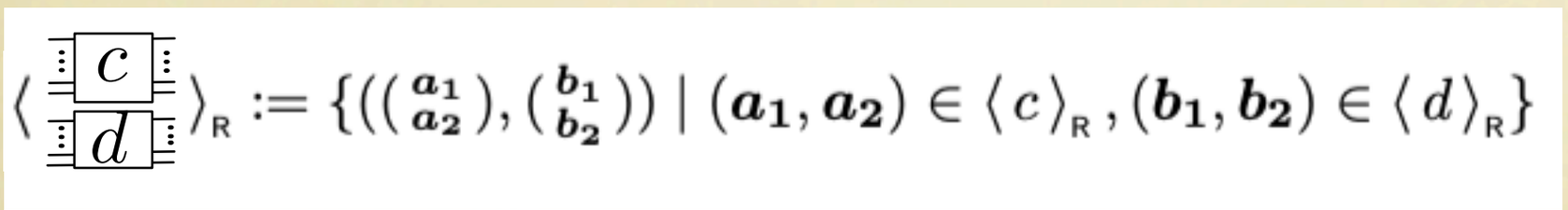
Multiply by n

$$\vdots$$



$$\langle \rangle_{\mathsf{R}} := \{(\boldsymbol{a}, \boldsymbol{b}) \mid \exists \boldsymbol{w}.(\boldsymbol{a}, \boldsymbol{w}) \in \langle c \rangle_{\mathsf{R}}, (\boldsymbol{w}, \boldsymbol{b}) \in \langle d \rangle_{\mathsf{R}}\}$$

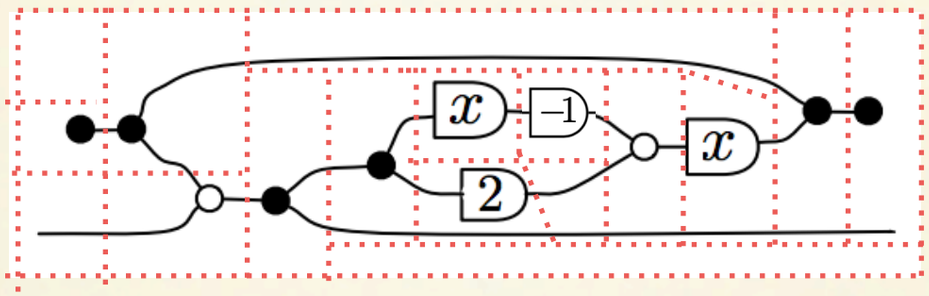
Synch



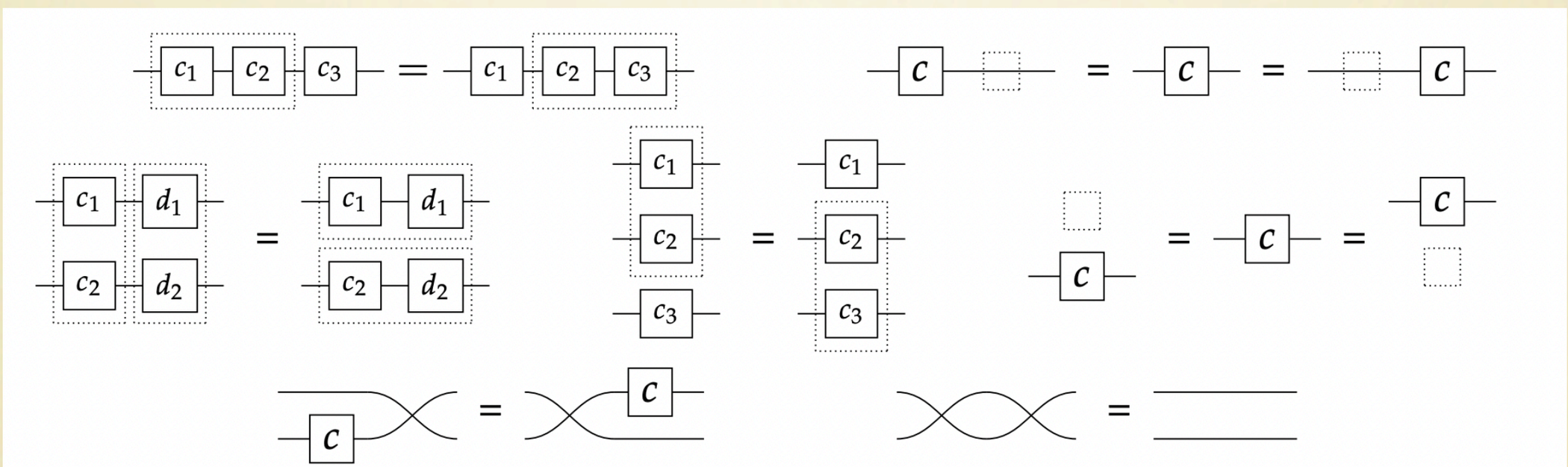
$$\langle \rangle_{\mathsf{R}} := \{((\tbinom{\boldsymbol{a_1}}{\boldsymbol{a_2}}), (\tbinom{\boldsymbol{b_1}}{\boldsymbol{b_2}})) \mid (\boldsymbol{a_1}, \boldsymbol{a_2}) \in \langle c \rangle_{\mathsf{R}}, (\boldsymbol{b_1}, \boldsymbol{b_2}) \in \langle d \rangle_{\mathsf{R}}\}$$

Parallel

Equational Reasoning



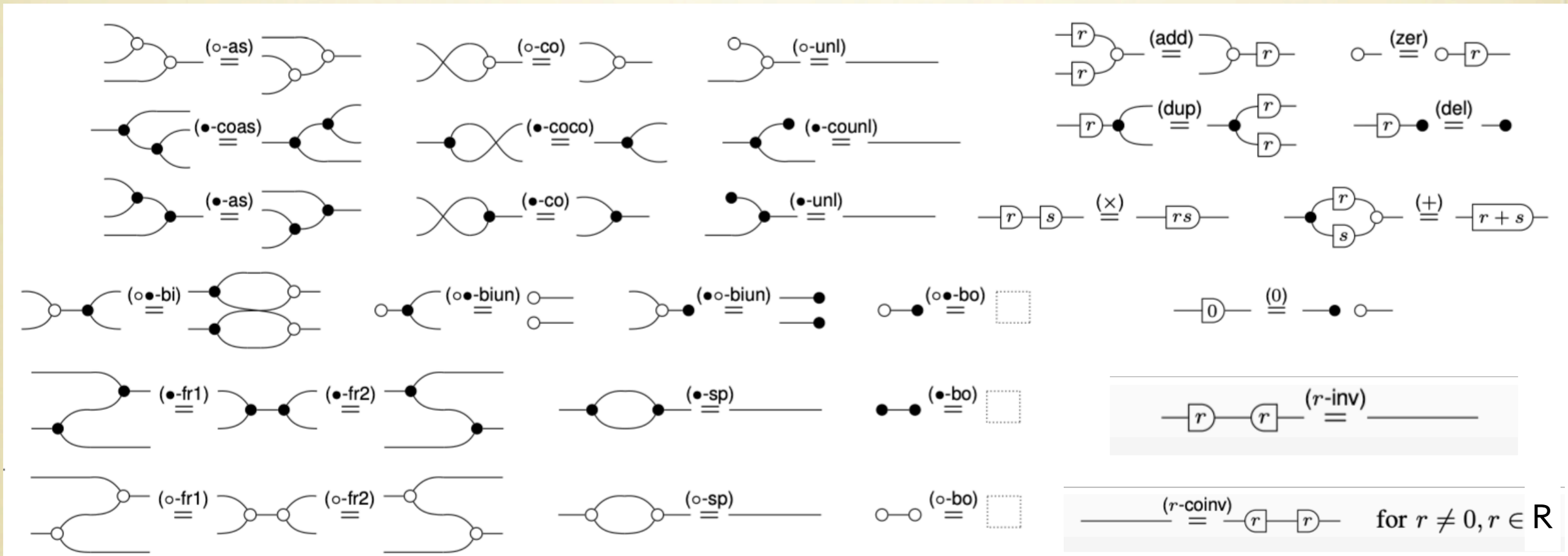
What equations allow us to switch between syntactic and combinatorial perspective?



String diagrams define morphisms in a **symmetric monoidal category**.

Equational Reasoning

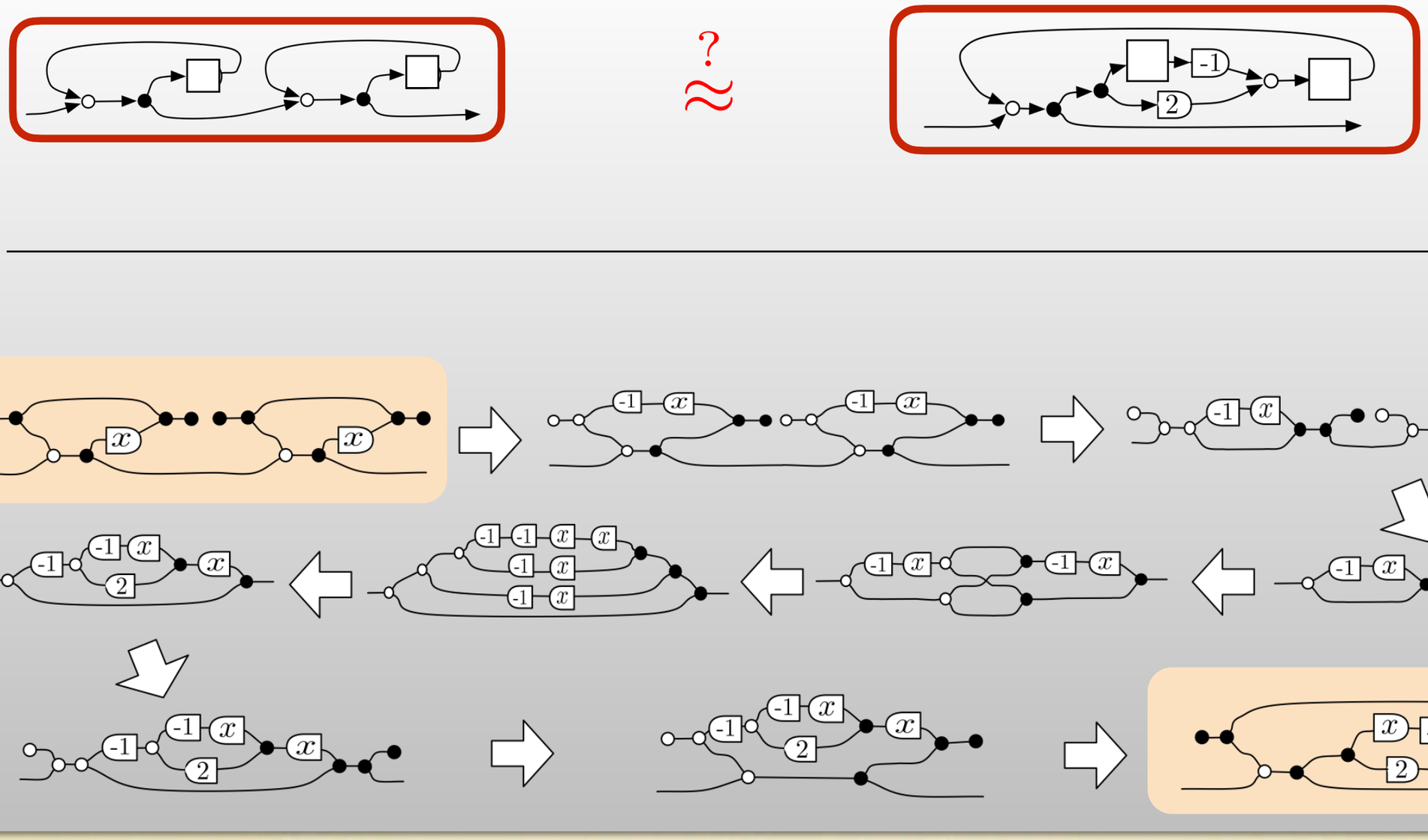
Equational theory **IH** of signal flow diagrams



Completeness Theorem

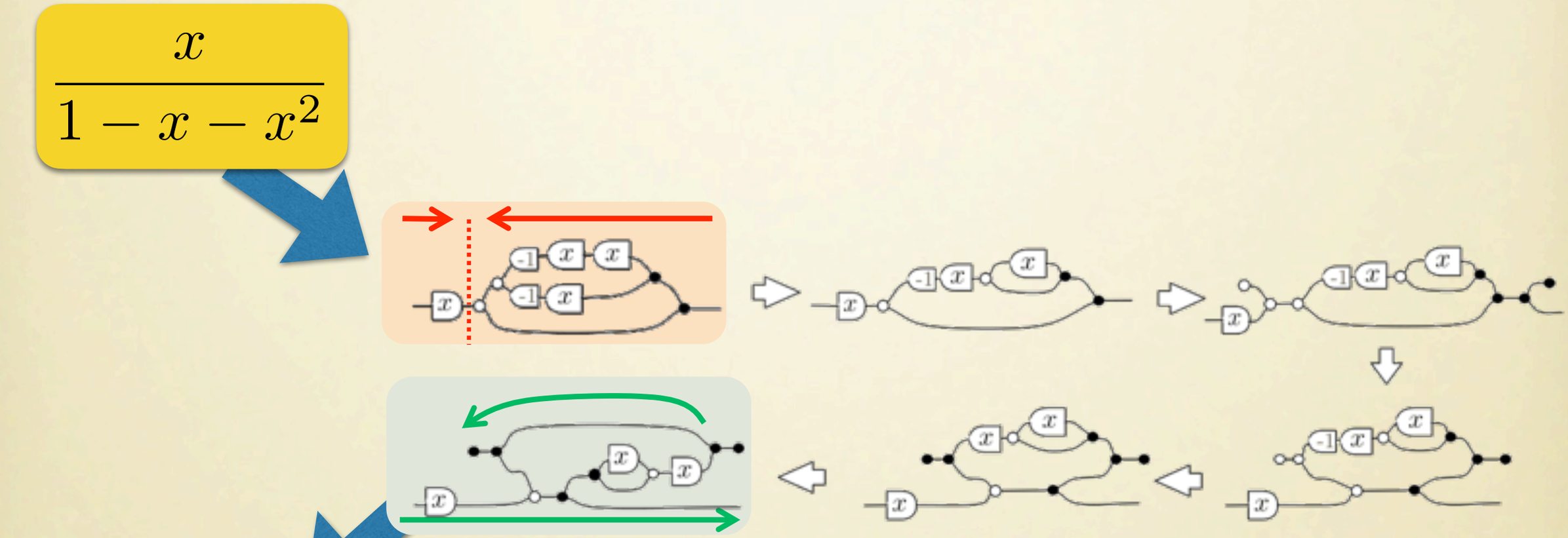
$\langle c \rangle_{\mathsf{R}} = \langle d \rangle_{\mathsf{R}}$ if and only if $c \overset{\mathbf{IH}}{=} d$

Equational Reasoning at Work



Equational Reasoning at Work

$$\frac{x}{1 - x - x^2}$$



Realisability Theorem

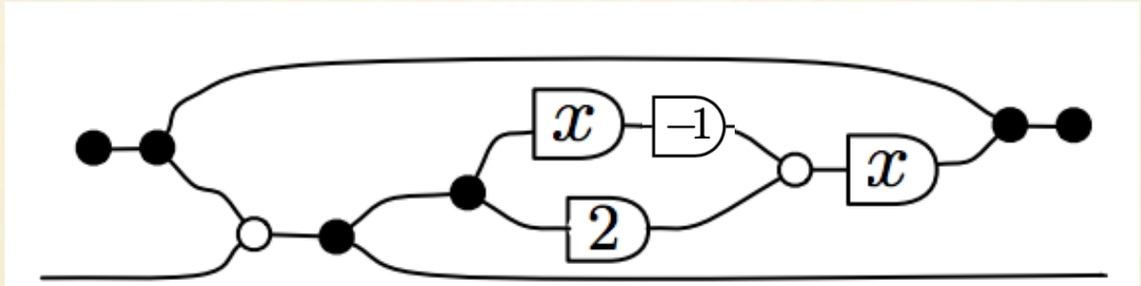
Any signal flow diagram can be transformed into an equivalent executable circuit.

…and more

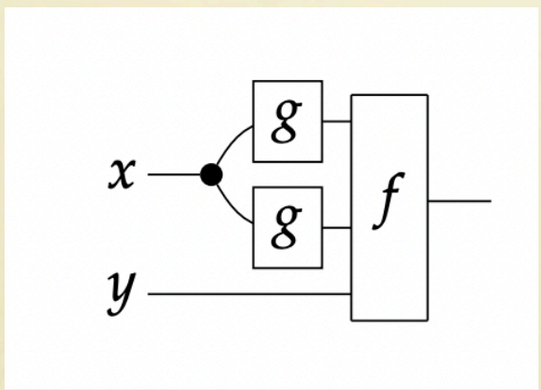
- Structural Operational Semantics
- Contextual Equivalence
- Full Abstraction
- Controllability (Control theory)
- Behaviour Refinement

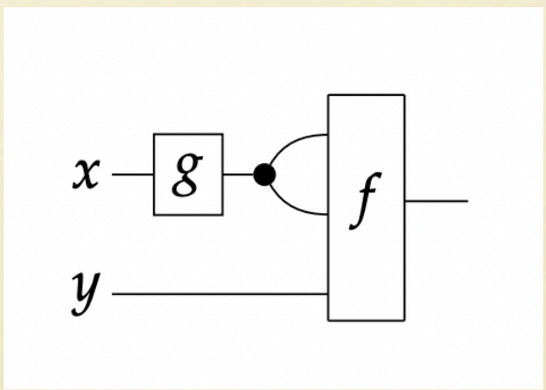
Coda: why not textual syntax?

- Preserve visual component-based reasoning



- Resource-sensitivity





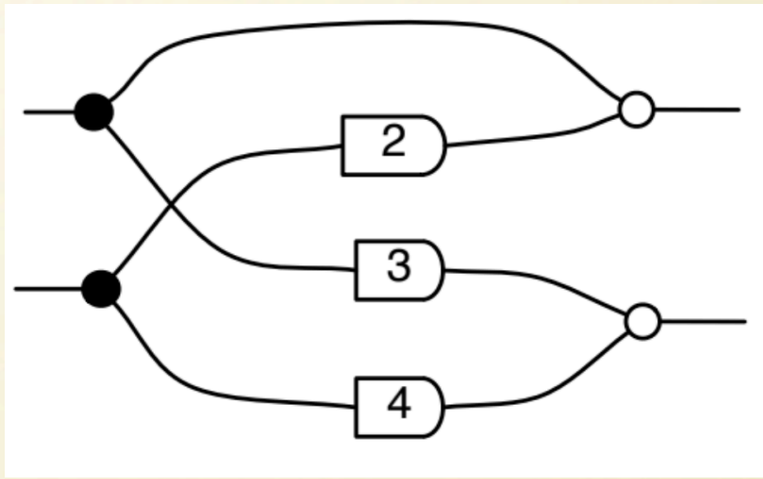
$$f(g(x), g(x), y)$$

Part III: a Tour on Diagrammatic Reasoning

Example I: Linear Algebra

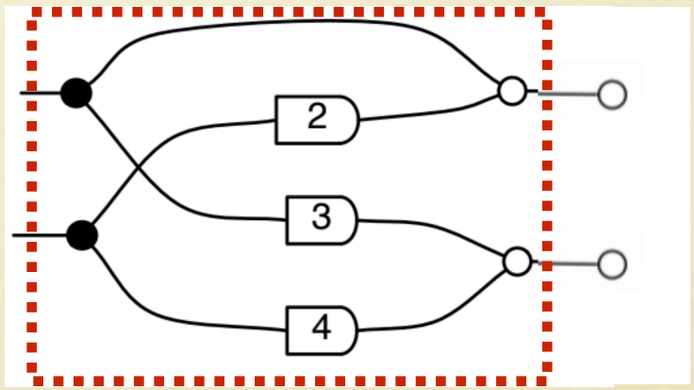
Picturing matrices with coefficients in **R**

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \longmapsto$$



Picturing **R** -linear subspaces (if **R** is a field)

Every subspace is the kernel of some matrix.

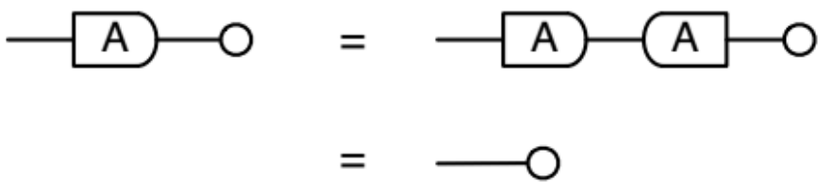


Example I: Linear Algebra

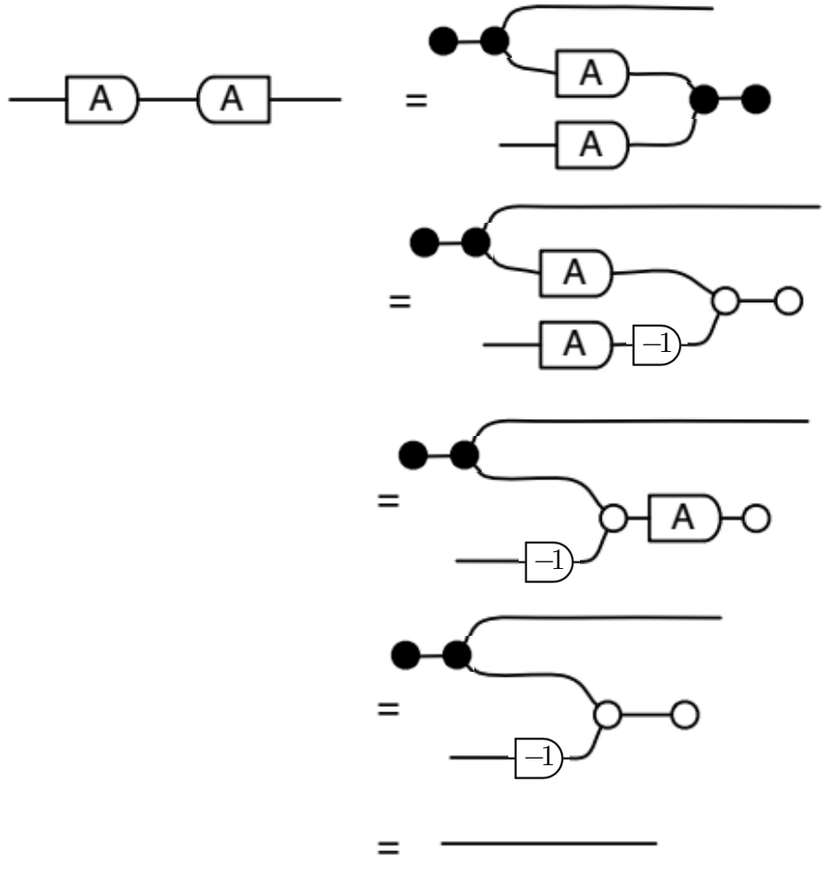
Proposition: a matrix A is injective iff its kernel is 0.

Proof

$\Rightarrow$

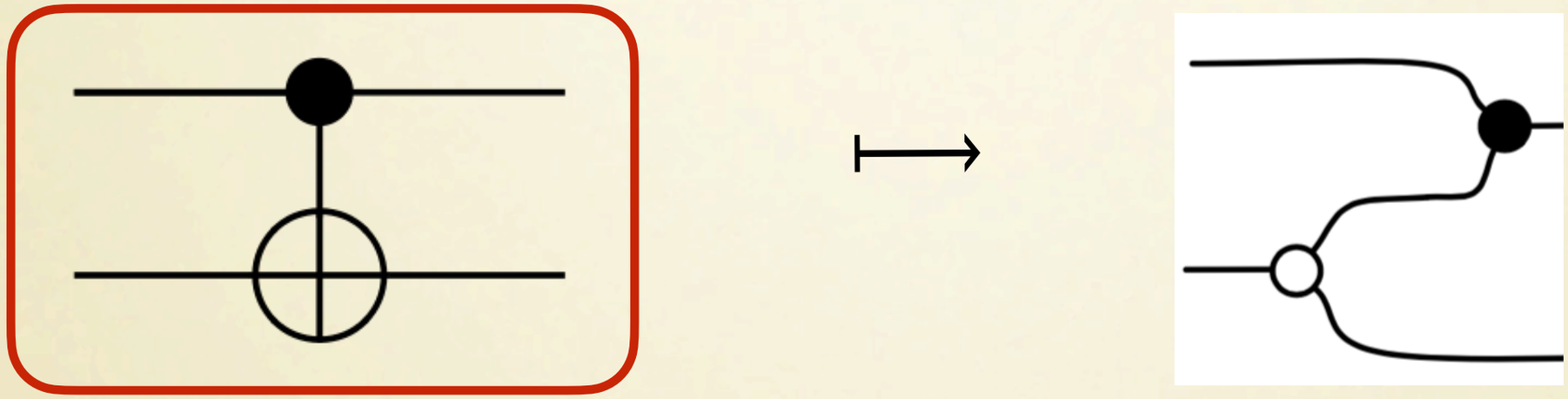


$\Leftarrow$

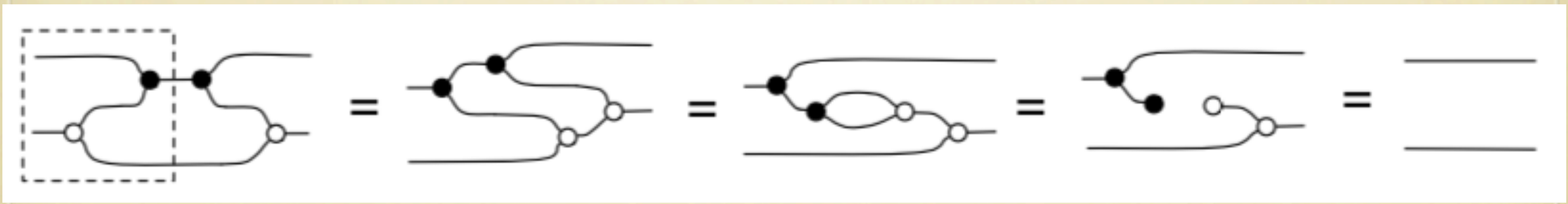


Example II: Quantum Circuits

CNOT gate interpreted as a signal flow diagram on $\mathbb{Z}/2$

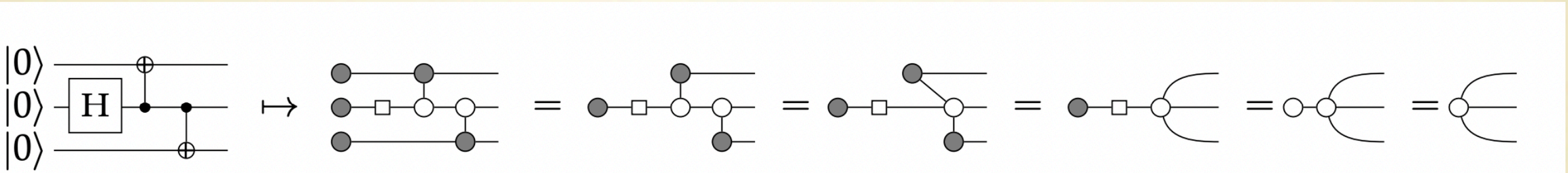


Proof by equational reasoning: CNOT gate is unitary



Example II: Quantum Circuits

A quantum circuit (left) preparing the GHZ state $|000\rangle + |111\rangle$



State-of-the-art algorithms for quantum circuits:

Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus

Ross Duncan, Aleks Kissinger, Simon Perdrix, John van de Wetering

We present a completely new approach to quantum circuit optimisation, based on the ZX-calculus. We first interpret quantum circuits as ZX-diagrams, which provide a flexible, lower-level language for describing quantum computations graphically. Then, using the rules of the ZX-calculus, we give a simplification strategy for ZX-diagrams based on the two graph transformations of local complementation and pivoting and show that the resulting reduced diagram can be transformed back into a quantum circuit. While little is known about extracting circuits from arbitrary ZX-diagrams, we show that the underlying graph of our simplified ZX-diagram always has a graph-theoretic property called generalised flow, which in turn yields a deterministic circuit extraction procedure. For Clifford circuits, this extraction procedure yields a new normal form that is both asymptotically optimal in size and gives a new, smaller upper bound on gate depth for nearest-neighbour architectures. For Clifford+T and more general circuits, our technique enables us to to `see around'

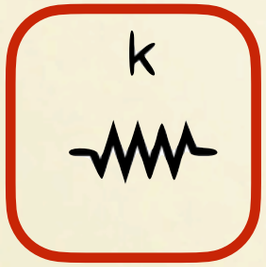
Graphical quantum Clifford-encoder compilers from the ZX calculus

Andrey Boris Khesin, Jonathan Z. Lu, Peter W. Shor

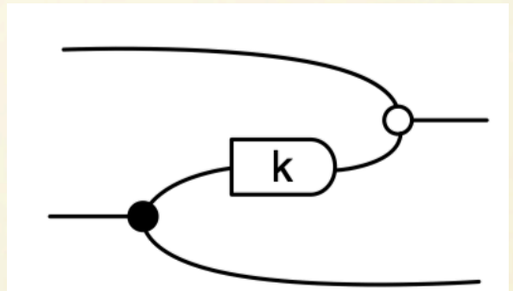
We present a quantum compilation algorithm that maps Clifford encoders, an equivalence class of quantum circuits that arise universally in quantum error correction, into a representation in the ZX calculus. In particular, we develop a canonical form in the ZX calculus and prove canonicity as well as efficient reducibility of any Clifford encoder into the canonical form. The diagrams produced by our compiler explicitly visualize information propagation and entanglement structure of the encoder, revealing properties that may be obscured in the circuit or stabilizer-tableau representation.

Example III: Electrical Circuits

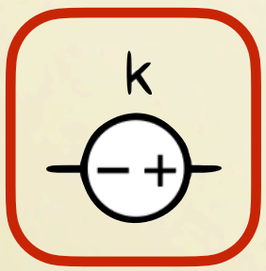
Encode electrical components as circuit diagrams on $\mathbb{R}$



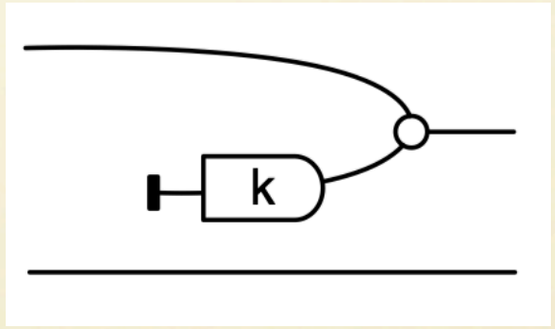
$\longmapsto$



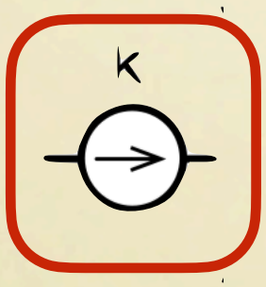
Resistor



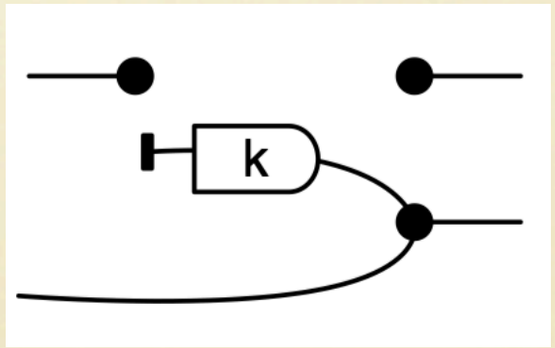
$\longmapsto$



Voltage



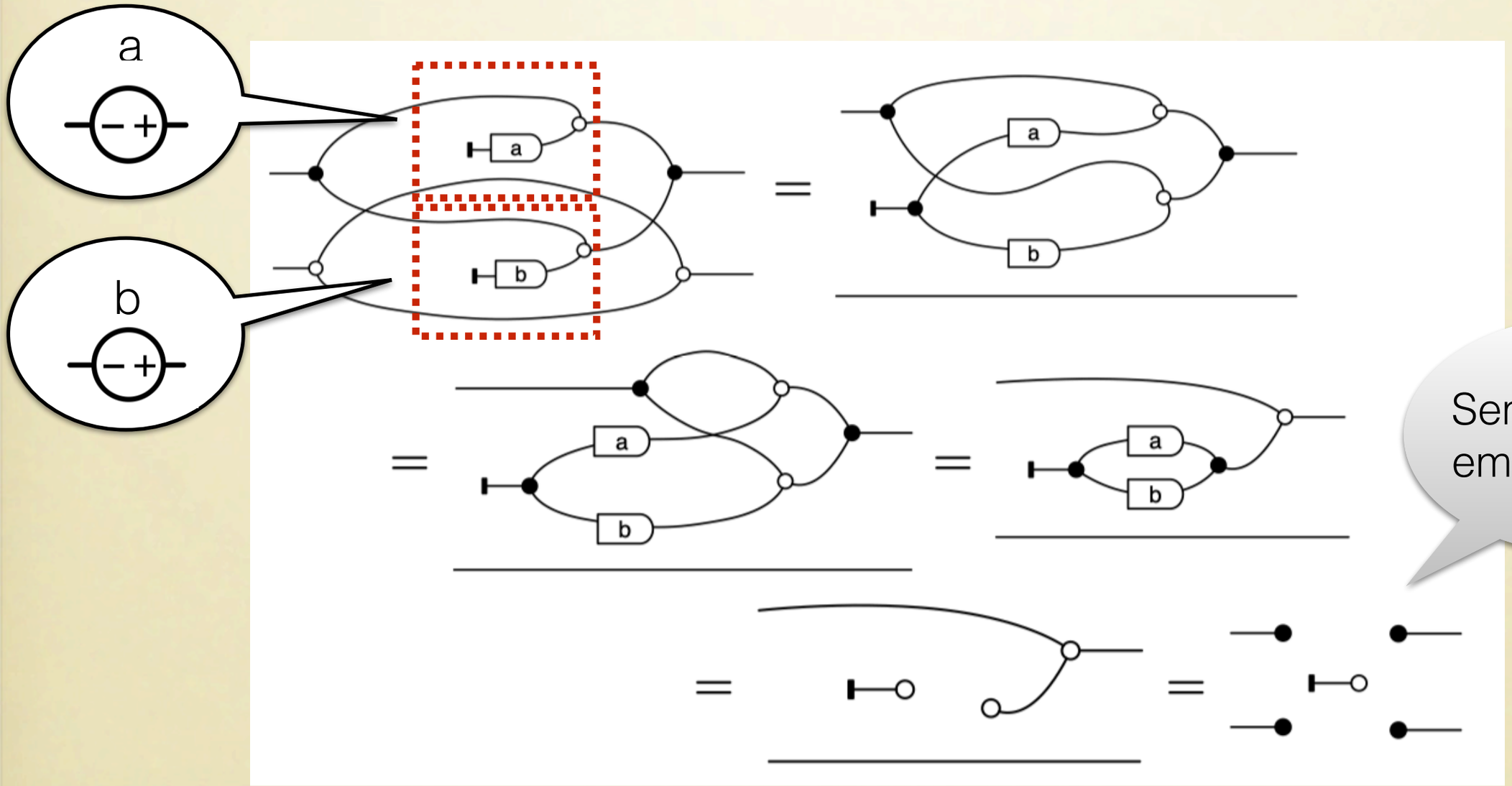
$\longmapsto$



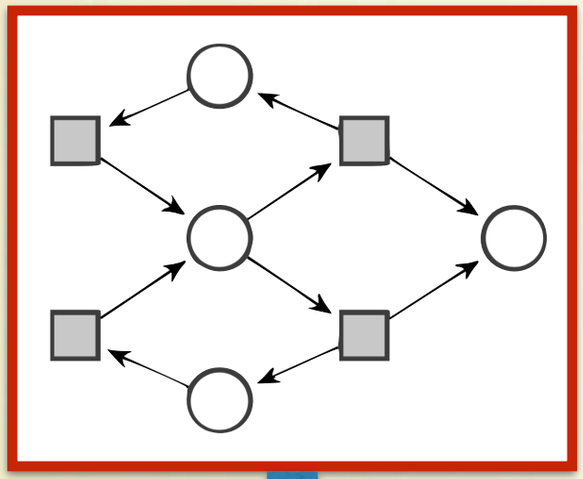
Current

Example III: Electrical Circuits

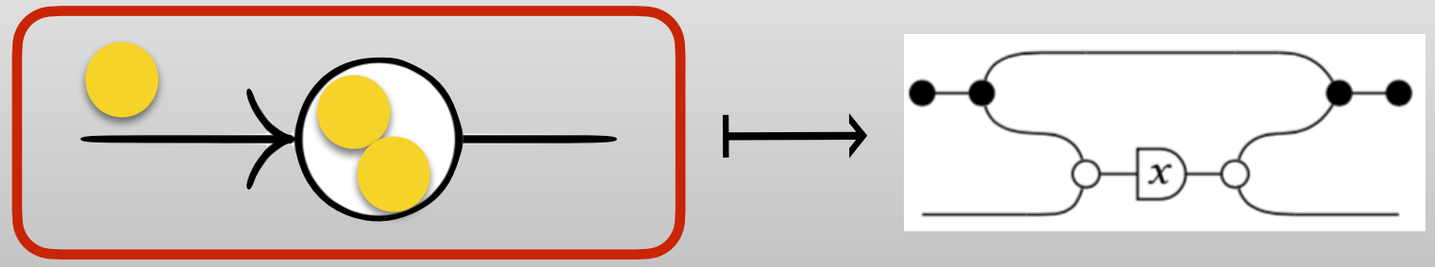
Proof that parallel voltage sources of different voltages are disallowed.

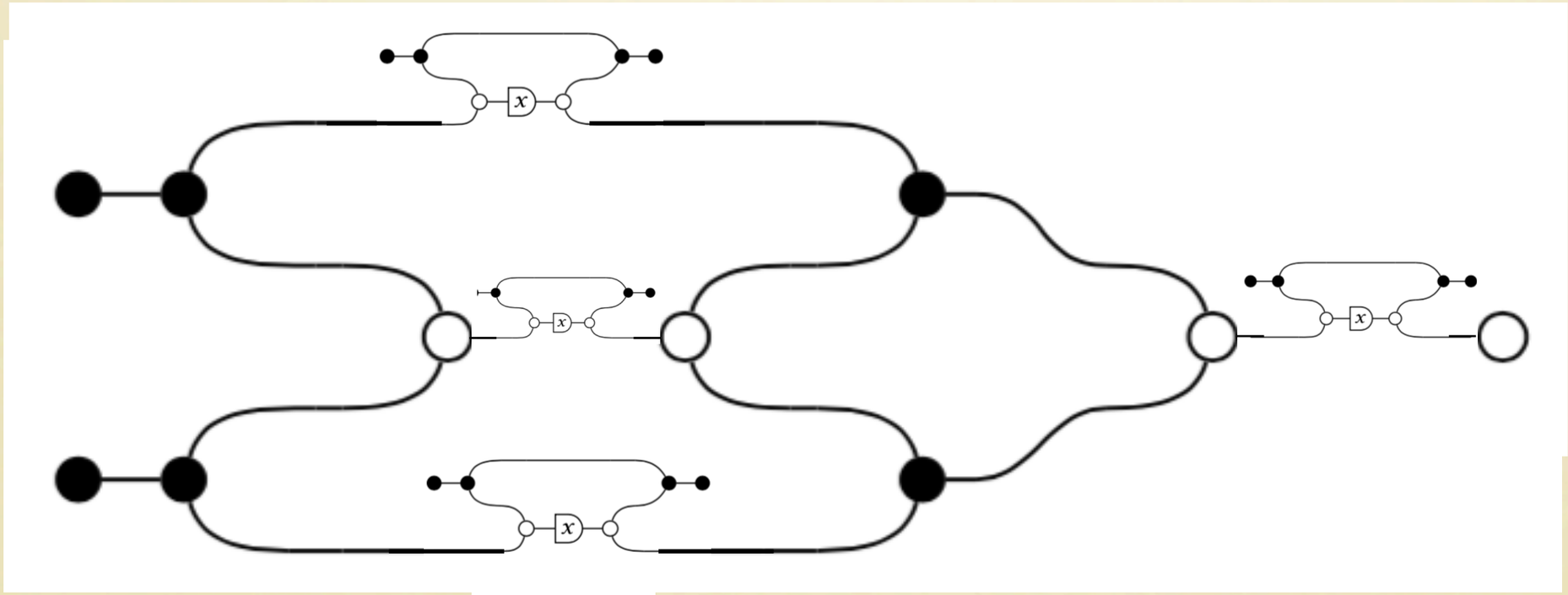


Example V: Petri Nets



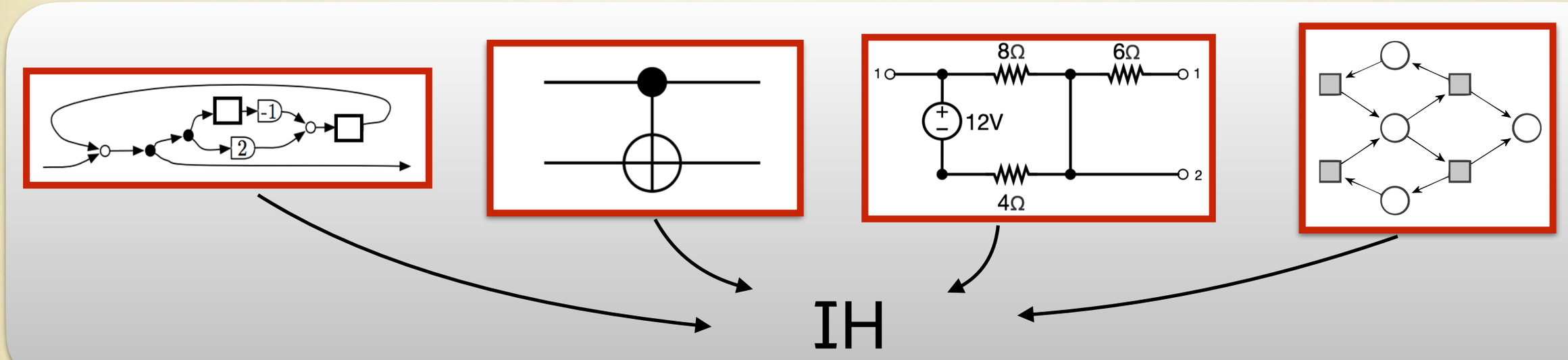
We interpret the **place of a Petri net** as a signal flow diagram over the semiring $\mathbb{N}$.



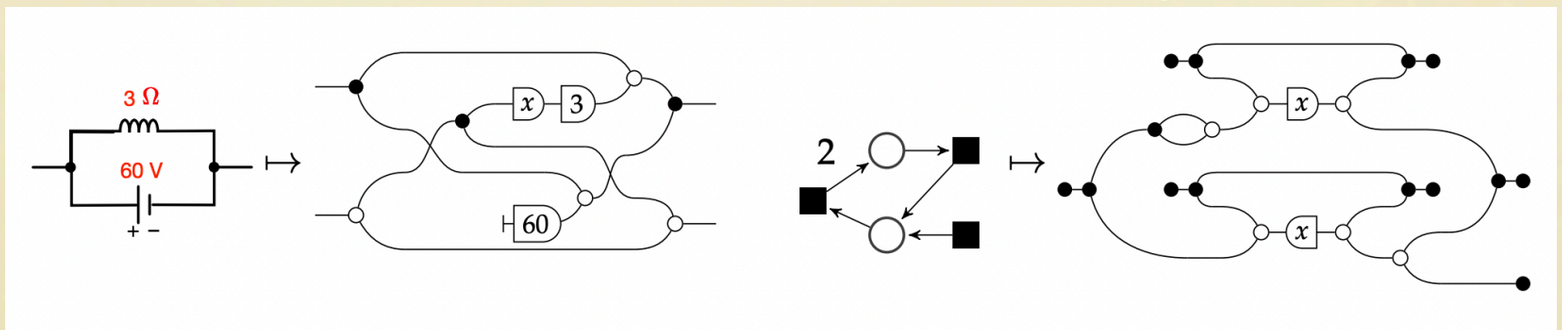


A uniform picture

Signal flow diagrams (with their equational theory IH) acts as an **assembly language** for diverse families of networks.

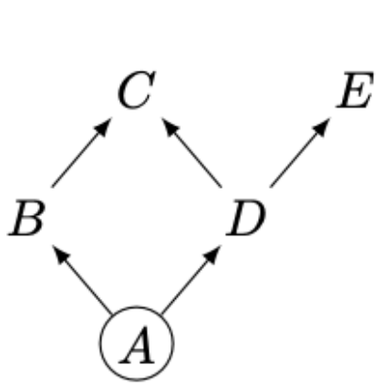


This approach reveals different models are reducible to the same elementary interactions.



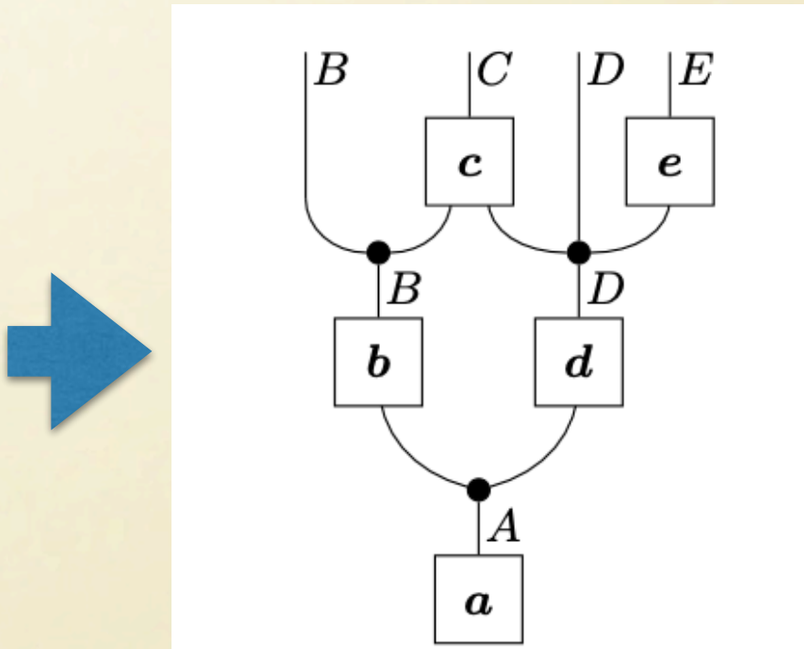
Example VI: Bayesian Probability

Bayesian Networks



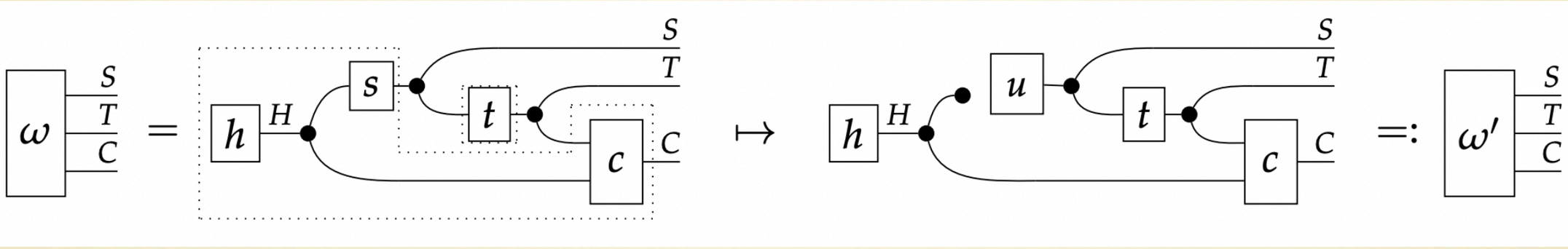
$$P(BCDE) =$$

$$\sum_A P(A)P(B|A)P(D|A)P(C|BD)P(E|D)$$



Example VI: Bayesian Probability

Causal Inference

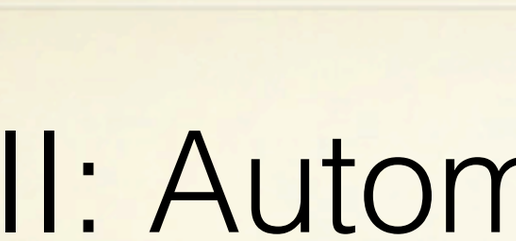


Establish whether statistical correlation between smoking and cancer actually identifies a causal relationship.

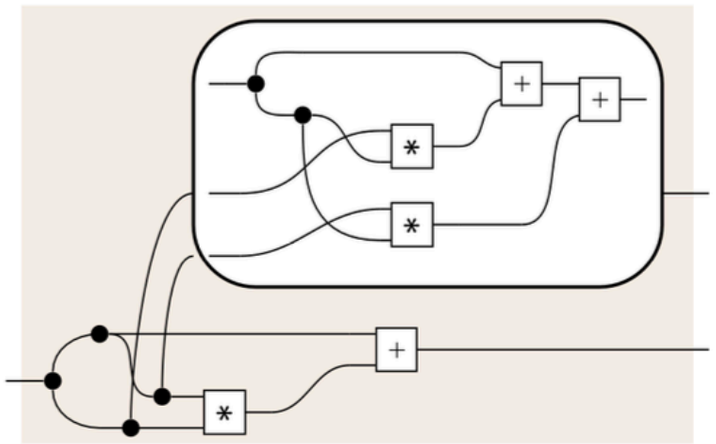
Example VII: Automatic Differentiation

```
def poly(x):
    def mul(y):
        return x * y
    return mul(x) + x
```

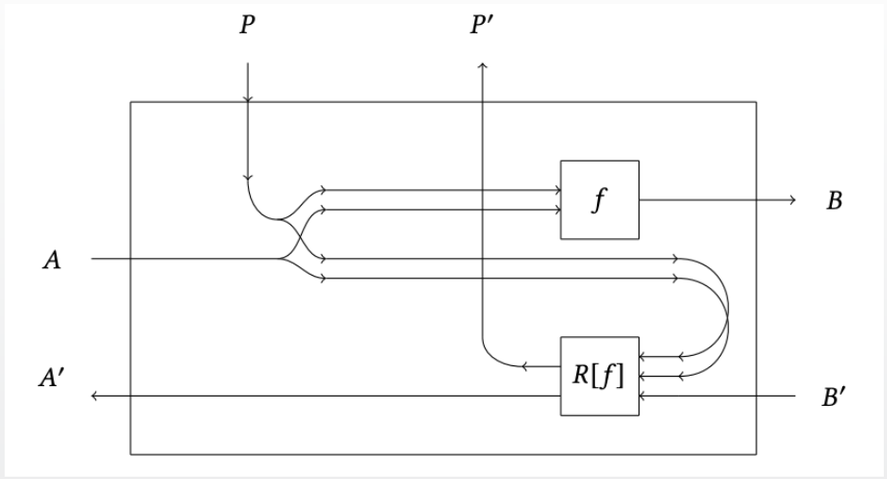
=

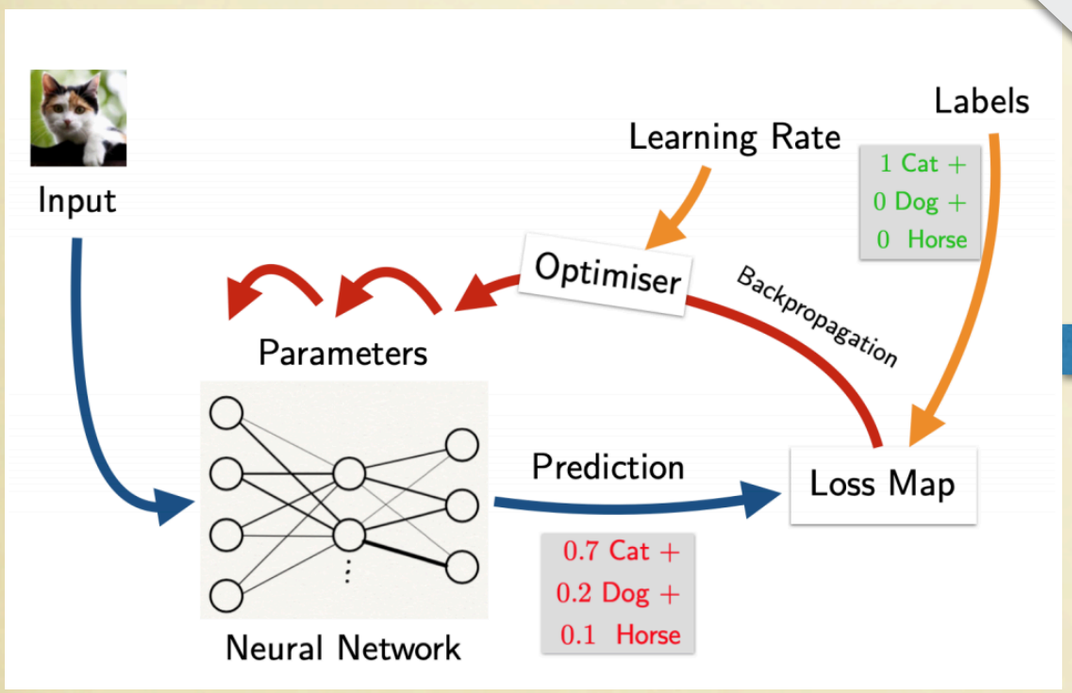


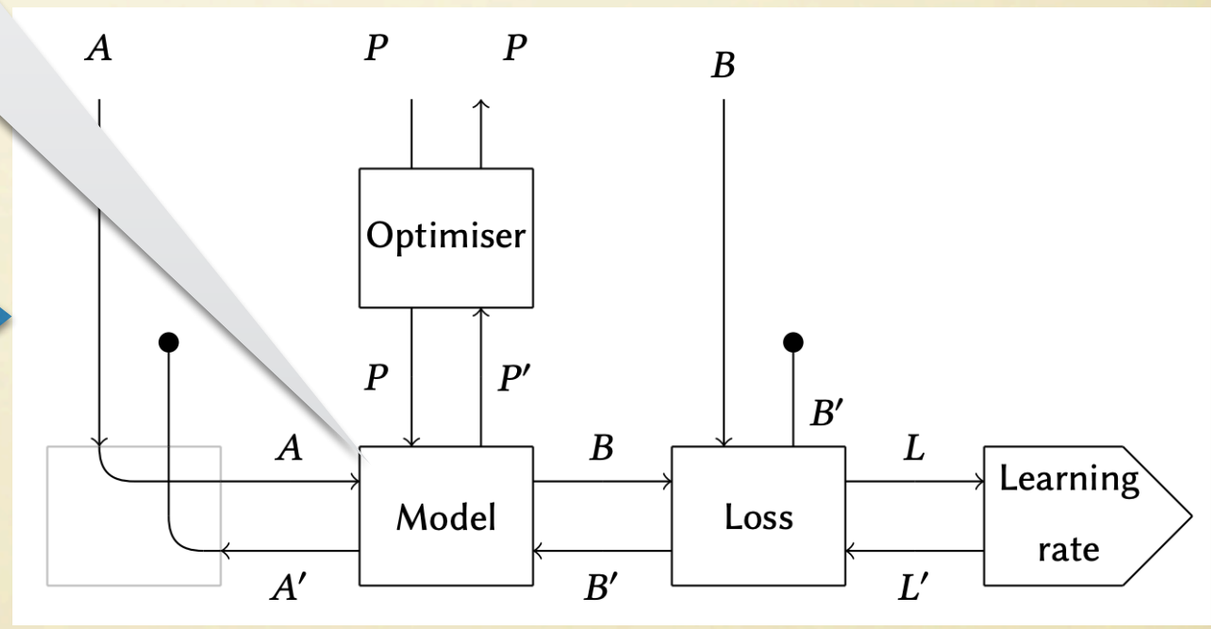
$\Longrightarrow^{\star}$



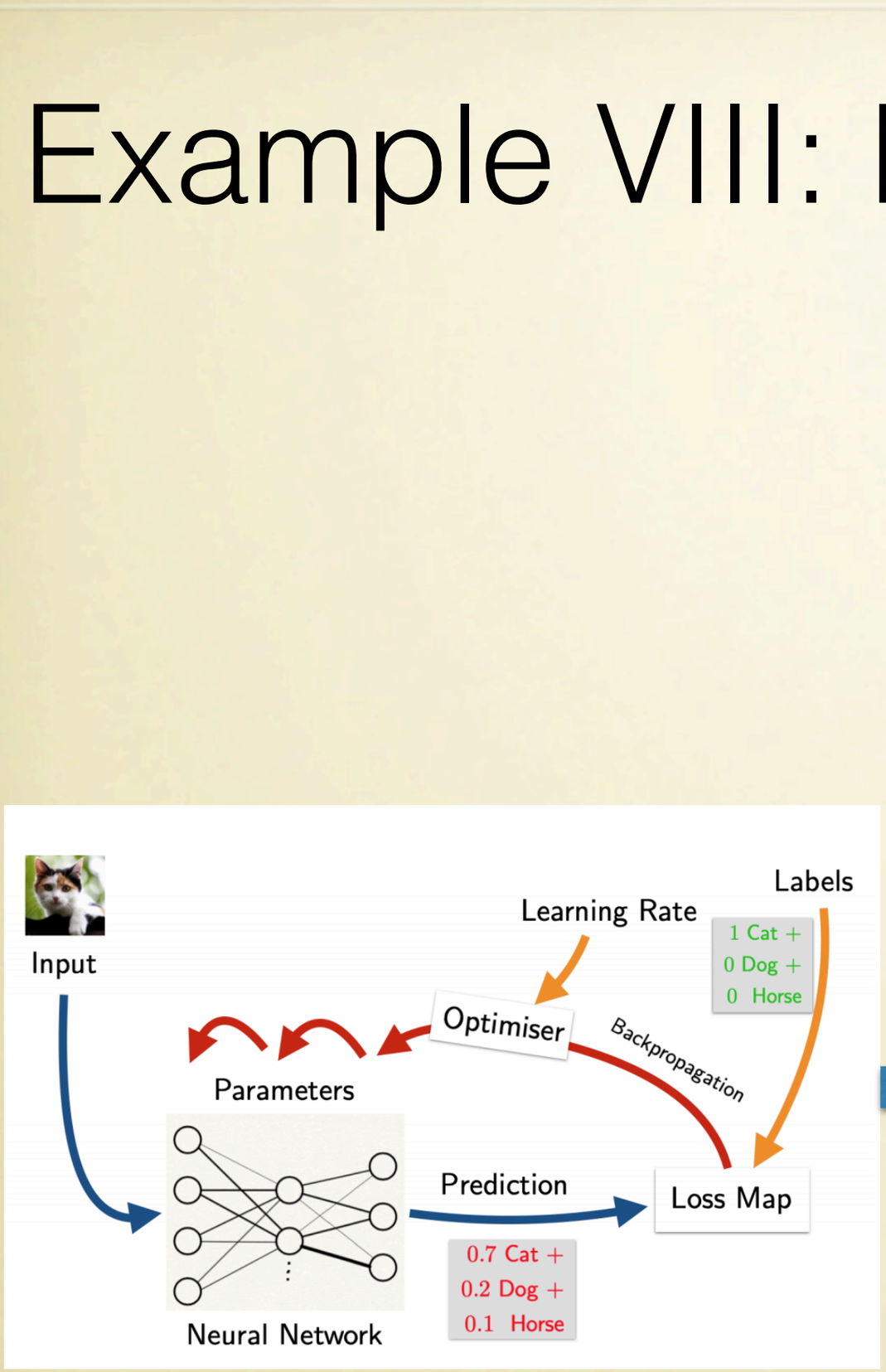
Example VIII: Machine Learning

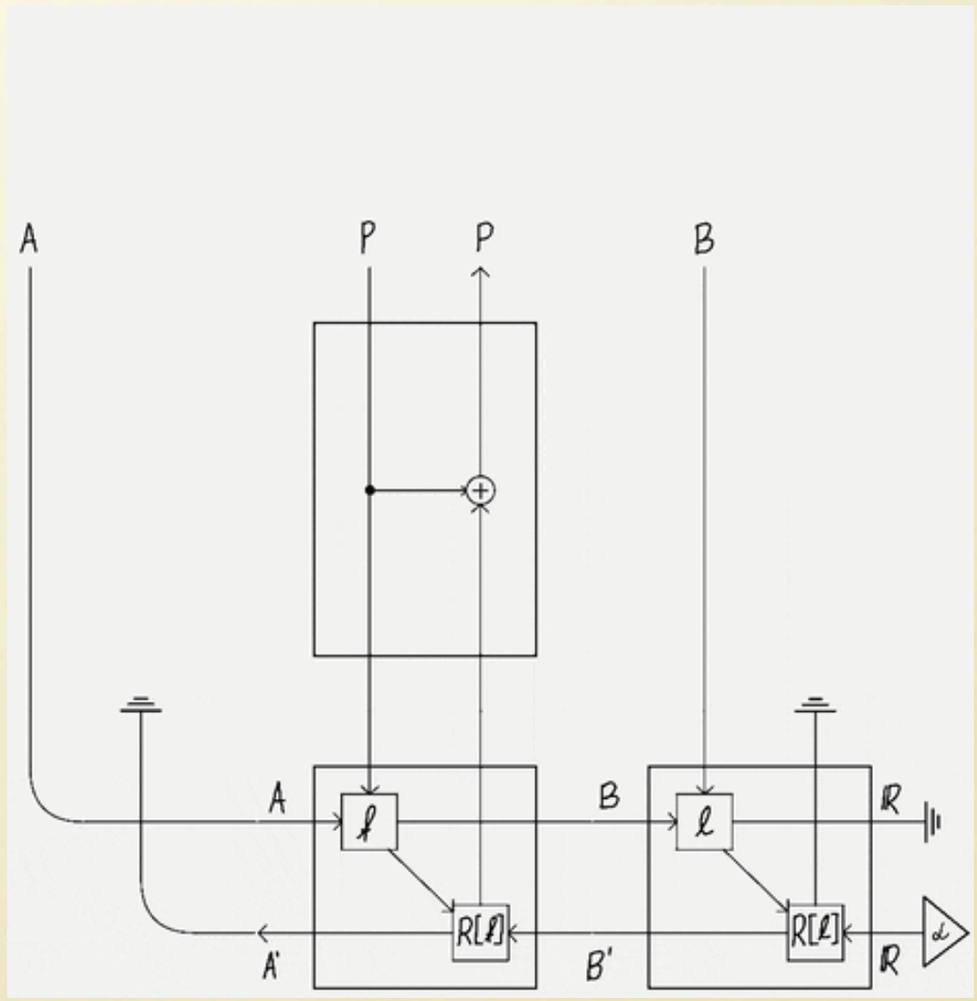






Example VIII: Machine Learning





Example VIII: Machine Learning

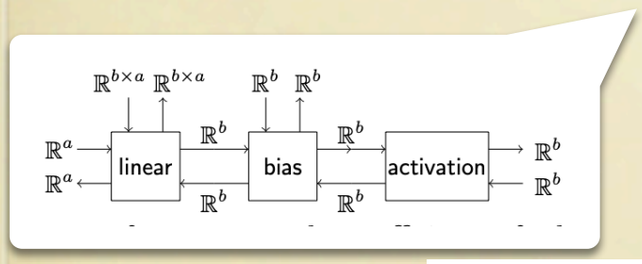
Programming Gradient-Descent Learning with Lenses

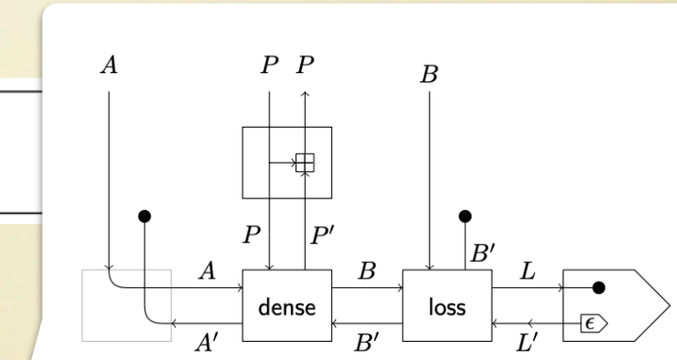
https://github.com/statusfailed/numeric-optics-python/

http://catgrad.com/p/reverse-derivative-ascent

http://yarrow.id

```
def dense(a, b, activation):
  return linear(a, b) >> bias(b) >> activation
```






```
apply_update(basic_update, dense) >> loss >> learning_rate(ϵ)
```

Part IV

Conclusions

Summing Up

String diagrams intend to take the best of two worlds:

- They are a formal syntax, which can be reasoned about in a compositional (algebraic) manner.
- They are a graphical language, which highlights how components in a system exchange resources.

String diagrams are a relatively young methodology, which is starting just now to inform the development of programming practices, as well as tools of analysis (especially compilers).

Want to know more?

- Introduction to String Diagrams for Computer Scientists
  arxiv.org/pdf/2305.08768.pdf
- Hierarchical String Diagrams and Applications
  arxiv.org/pdf/2305.18945.pdf
- A Survey on Compositional Signal Flow Theory
  http://www.zanasi.com/fabio/files/paperIFIP21.pdf
- Blog Tutorial: Graphical Linear Algebra
  www.graphicallinearalgebra.net

Disclaimer

Looking for an internship/PhD/Postdoc on these topics?

www.zanasi.com/fabio

fabio.zanasi@unibo.it

f.zanasi@ucl.ac.uk