From Proof Theory to Proof Assistants

Challenges of Responsible Software and AI

Klaus Mainzer

Emeritus of Excellence
Technical University of Munich

Senior Professor
Eberhard Karls University of Tübingen

Incorrectness of Programs leads to Catastrophies

Dramatic accidents highlight the dangers of safety-critical systems without software verification :

- Killed by a machine by massive overdoses of radiation - Therac-25 1985-87

- Crash of Ariane 5 by software failure 1996

- Software failure of Boing 737 Max 2019

1. **Mathematical Proofs and Intuitionistic Type Theory**
2. **Intuitionistic Type Theory and Proof Assistants**
3. **Verification of Circuits in Proof Assistants: Basics**
4. **Verification of Circuits in Proof Assistants: Application**
5. **Verification of Machine Learning in Proof Assistants**
6. **Perspectives of Responsible Artificial Intelligence**

1. Mathematical Proofs and Intuitionistic Type Theory

Curry-Howard Correspondance

In 1969, the logician W.A. Howard observed that Gentzen's *proof system of natural deduction* can be directly interpreted in its *intuitionistic version* as a *typed variant* of the mode of *computation* known as *lambda calculus*.

According to Church, $\lambda a.b$ means a *function* mapping an element a onto the function value b with $\lambda a.b[a] = b$. In the following, *proofs* are represented by terms $a, b, c, \dots$; *propositions* are represented by $A, B, C, \dots$.

Examples:

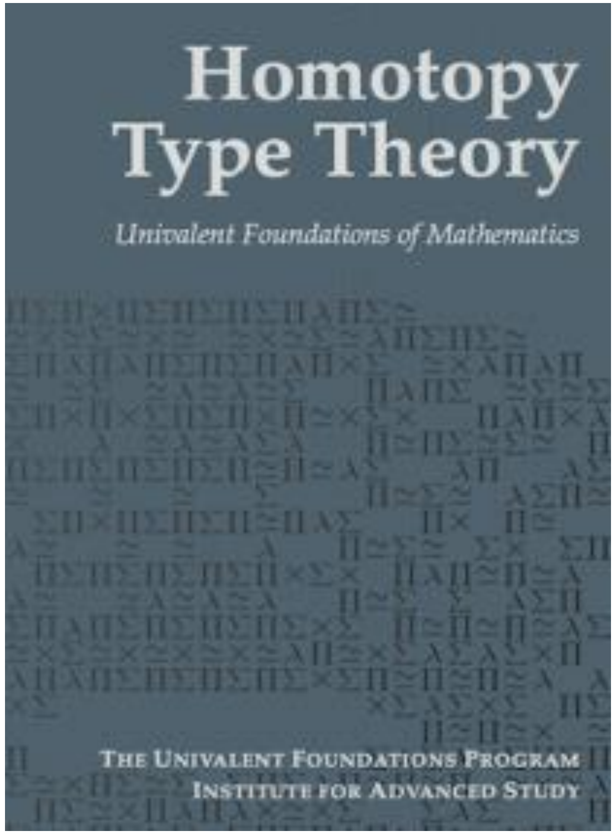
$$\begin{array}{cc} & [A] \\ \lambda a(\lambda b.a) & \vdots \\ & \underline{B \to A} \\ (\to \mathrm{I}) & A \to (B \to A) \end{array} \qquad\qquad \begin{array}{cc} & [A] \\ \lambda a.b & \vdots \\ & \underline{B} \\ (\to \mathrm{I}) & A \to B \end{array}$$

A proof is a program, and the formula it proves is the type for the program.

Martin-Löf's Intuitionistic Type Theory

In addition to the *type formers* of the *Curry-Howard interpretation*, the logician and philosopher P. Martin-Löf extended the *basic intuitionistic type theory* (containing *Heyting's arithmetic of higher types* HA$^{\omega}$ and *Gödel's system* T *of primitive recursive functions of higher type*) with *primitive identity types*, *well founded tree types*, *universe hierarchies* and general notions of *inductice* and *inductive–recursive definitions*.

His extension increases the <u>proof-theoretic strength</u> of the theory and its application to <u>programming</u> as well as to <u>formalization of mathematics</u>.



Since their very beginning, *data types* play a crucial role in *computer languages*:

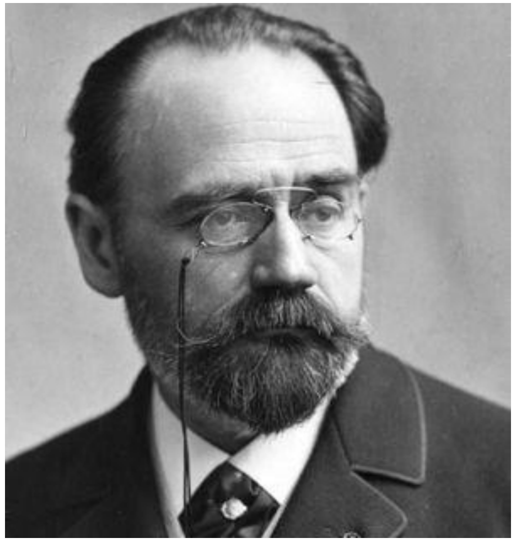
How far can mathematical objects be represented with types of computer languages?

***Homotopy theory* is an outgrowth of *algebraic topology* and *homological algebra* with relationships to higher *category theory* which can be considered as *fundamental concepts of mathematics*.**

***Type theory* is a branch of *mathematical logic* and *theoretical computer science*.**

***Homotopy type theory* (HoTT) interprets *types* as *objects of abstract homotopy theory*. Therefore, HoTT tried to develop a *universal* („*univalent*“) *foundation of mathematics* as well as *computer language* with respect to the *proof assistant Coq*.**

Trust & Security in Mathematics



Nowadays, ***mathematical arguments*** had become so ***complicated*** that a *single mathematician* rarely can examine them in detail: They trust in the *expertise of their colleagues*. The situation is completely similar to ***modern industrial labor world***: According to the French sociologist Emile Durkheim (1858-1917), modern industrial production is so *complex* that it must be organized on the ***principle of division of labor and trust in expertise***, but *nobody* has the *total survey*.

On the background of ***critical flaws*** overlooked by the *scientific community*, Vladimir Voevodsky (1966-2017) *no longer trusted* in the principle of "job-sharing". Humans could not keep up with the *ever-increasing complexity of mathematics*. ***Are computers the only solution***? Thus, his foundational program of univalent mathematics is inspired by the idea of a ***proof-checking software*** to *guarantee trust & security in mathematics*.

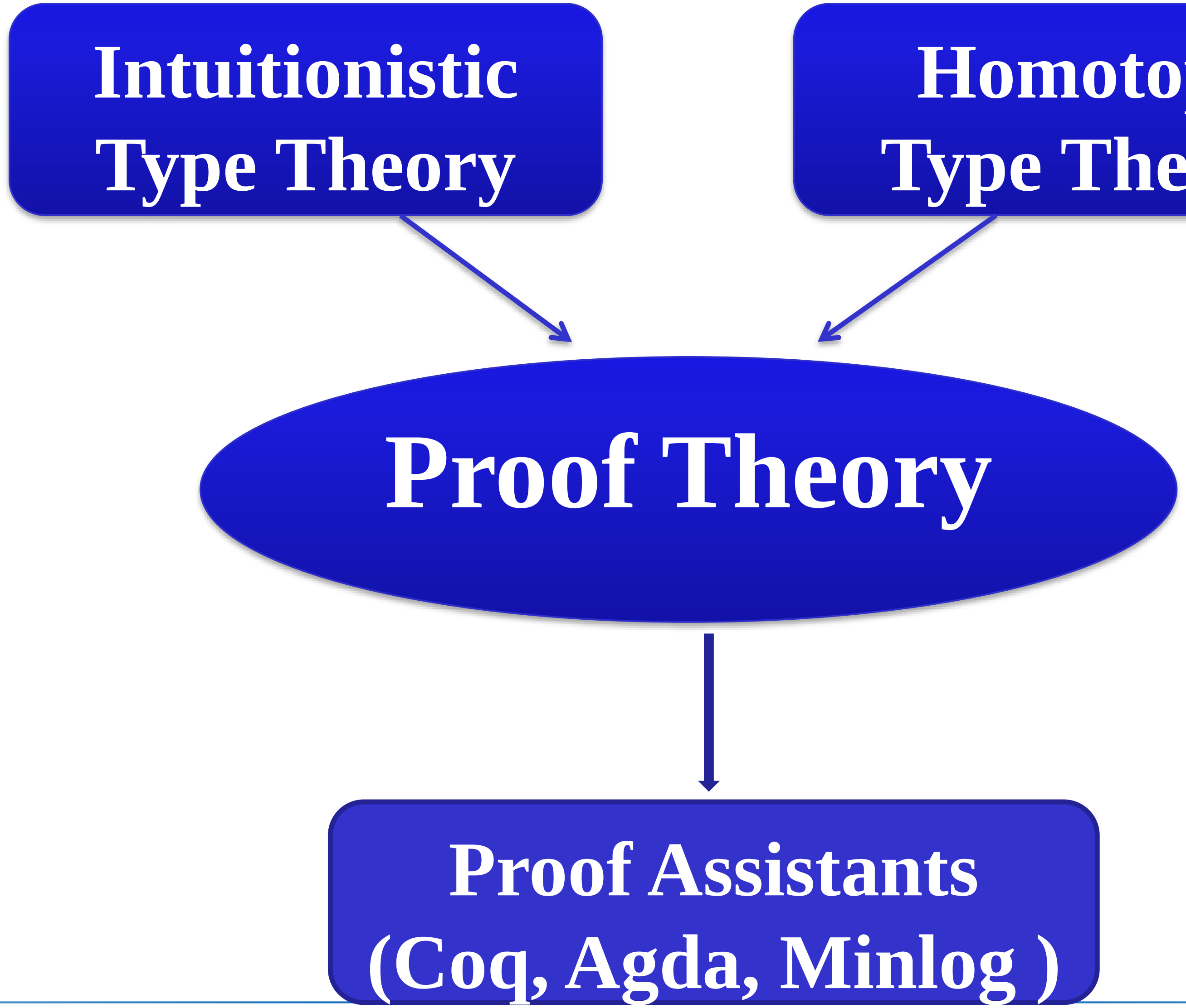
Verification of Proofs in HoTT

HoTT allows mathematical proofs to be translated into a computer programming language for *computer proof assistants* (e.g., Coq) even for advanced mathematical categories with "*isomorphism as equality*"(UA). Therefore, an essential goal of HoTT is :

type checking ⇒ proof checking in higher categories („difficult proofs")

Besides UA, HoTT is extended by *higher inductively defined structures* (e.g. *inductively defined spaces* with collections of *points*, *paths*, *higher paths* et al.) which can be characterized by appropriate *induction principles*. HoTT is *consistent* with respect to a *model* in the *category of Kan complexes* (V. Voevodsky). Thus, it is consistent relative to ZFC (with as many *inaccessible cardinals* which are necessary for *nested univalent universes*).

But it is still an *open question* whether it is possible to provide a constructive justification of the Univalence Axiom (UA).


Intuitionistic Type Theory
Homotopy Type Theory
Proof Theory
Proof Assistants (Coq, Agda, Minlog)

2. Intuitionistic Type Theory and Proof Assistant

Terms of the Calculus of Constructions (CoC)

CoC is a *type theory* of Thierry Coquand et al. which can serve as *typed programming language* as well as *constructive foundation of mathematics*. It extends the *Curry-Howard isomorphism* to *proofs* in the *full intuitionistic predicate calculus.* Coc has very few *rules of construction* for terms:

- **T is a *term* (*Type*).**
- **P is a *term* (*Prop*).**
- ***Variables* $(x, y, z, \ldots)$ are *terms*.**
- **If A and B are *terms*, then (AB) is a *term*.**
- **If A and B are *terms* and x is a *variable*, then $\lambda x\, A.B$ and $\forall x\, A.B$ are *terms*.**

The *objects* of CoC are *proofs* (terms with propositions as types), *propositions* (small types), *predicates* (functions that return propositions), *large types* (types of predicates, e.g., P), T (type of large types).

Inference Rules of CoC

Γ is a sequence of type assignments $x_1 : A_1, x_2 : A_2, \ldots$; K is either T or P :

$$\frac{}{\Gamma \vdash P : T} \qquad \frac{\Gamma \vdash A : K}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash B : K \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash (\lambda x\, A.\, N) : (\forall x\, A.\, B) : K}$$

$$\frac{\Gamma \vdash M : (\forall x : A.\, B) \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

$$\frac{\Gamma \vdash M : A \qquad A =_\beta B \qquad B : K}{\Gamma \vdash M : B}$$

Logical Operators and Data Types in CoC

Coc has very few basic operators. The *only logical operator* for forming *propositions* is ∀ :

logical operators:

$$A \Rightarrow B \equiv \forall x{:}A.B \quad (x \notin B)$$
$$A \wedge B \equiv \forall C{:}P.(A \Rightarrow B \Rightarrow C) \Rightarrow C$$
$$A \vee B \equiv \forall C{:}P.(A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C$$
$$\neg A \equiv \forall C{:}P.(A \Rightarrow C)$$
$$\exists x{:}A.B \equiv \forall C{:}P.(\forall x{:}A(B \Rightarrow C)) \Rightarrow C$$

data types:

booleans: $\forall A{:}P.A \Rightarrow A \Rightarrow A$

naturals: $\forall A{:}P.(A \Rightarrow A) \Rightarrow (A \Rightarrow A)$

product $A \times B$: $A \wedge B$

disjoint union $A + B$: $A \vee B$

Calculus of Inductive Constructions (CiC)

CiC is based on CoC enriched with *inductive* and *co-inductive definitions* with the following *rules for constructing terms*:

- ***identifiers* refer to *constants* or *variables*.**
- **(AB) *application* of a *functional object* A to B**
- **$[x: A]B$ *abstraction* of variable x of type A in term B to construct a *functional object* $\lambda x \in A.B$**
- **$(x: A)B$ *term* of type Set corresponds to $\prod_{x \in A} B$ product of sets.**
  $(x: A)B$ *term* of type Prop corresponds to $\forall x \in A\ B$.

  If x does *not* occur in B, $A \rightarrow B$ is an abbreviation which corresponds to
  - ***set of all functions* from A to B**
  - ***logical implication***

Inductive Types in CiC*

An *inductive type* is *freely generated* by a certain number of *constructors*.

Examples: a) Type $\mathbb{N}$ of natural numbers with *constructors*
- $0 : \mathbb{N}$
- $\mathrm{succ} : \mathbb{N} \to \mathbb{N}$

b) Type $\mathrm{List}(A)$ of finite lists of elements of type A with *constructors*
- $\mathrm{nil} : \mathrm{List}(A)$
- $\mathrm{cons} : A \to \mathrm{List}(A) \to \mathrm{List}(A)$

Inductive proofs make it possible to prove statements for *infinite collections* of objects (e.g., integers, lists, binary trees), because all these *objects* are constructed in a *finite number of steps*.

An *induction principle* of an *inductive type* proves a *statement* for a *type freely generated by its constructors*.

* C. Paulin-Mohring (1993), Inductive Definition in the System Coq: Rules and Properties (Research Report 92-49, LIP-ENS Lyon)

Co-Inductive Types in CiC*

Besides *inductive types*, there are *co-inductive types* concerning *infinite objects* (e.g., potentially infinite lists, potentially infinite trees with infinite branches).

***Terms* are still be obtained by *repeated uses of constructors* such as in *inductive types*. However, there is *no induction principle* and the *branches* may be *infinite*.**

In *practical domains* such as *telecommunication*, *energy*, or *transportation*, *streams* are examples with *infinite execution* which are defined by constructor `Cons`:

```
CoInductive Stream (A : Set) : Set :=
Cons : A → Stream A → Stream A
```

Contrary to the *inductive type* of a `list`, there is *no constructor* of the empty list. Thus, *finite lists cannot* be constructed.

* E. Giménez (1996), Un calcul de constructions infinies et son application à la vérification de systèmes communicants (PhD thesis Lyon)

Equivalence of Streams in CiC

Accessors of a *stream* `l` are defined by functions on the structure of the stream with *head* `hd` and *tail* `tl`:

```
Definition Head: Stream → A := [l] Cases l of (Cons hd _ ) ⇒ hd end.
Definition Tail: Stream → Stream := [l] Cases l of (Cons  _ tl) ⇒ tl
end.
```

Two *streams* `l` and `l`' are *equivalent* iff their *heads* are *equal* and their *tails* are *equivalent*. In CiC, *equivalence of streams* is represented by a *co-inductive definition*:

```
CoInductive EqS : Stream → Stream → Prop := eqs : (l , l' : Stream)
                                                  (Head l) = (Head l') →
                                                  (EqS (Tail l)( Tail l')) →
                                                  (EqS l l').
```

Production of Streams in CiC

The *mapping* of a given function f on *two streams* l and l' is *co-recursively defined* in CiC:

```
CoFixpoint Map2 : (A, B, C : Set)
                  (A → B → C) → (Stream A) → (Stream B) →(Stream C) :=
  [A, B, f, l, l‘]
  (Cons (f (Head l)(Head l‘))(Map2 f (Tail l)(Tail l‘)))
```

The function $Prod$ builds the *stream of the pairs*, element by element, *of two streams* of type $(Stream\ A)$ and $(Stream\ B)$ respectively. $Prod$ is the result of the *application* $Map2$ to the function $(pair\ A\ B)$, where $pair$ is the *constructor* of the *cartesian product* $A * B$. In CiC, $Prod$ is represented by:

```
Definition Prod := [A, B : Set] (Map2 (pair A B ))
```

The Coq Proof Assistant*

Coq implements a *program specification* which is based on the *Calculus of Inductive Constructions* (CiC) combining both a *higher-order logic* and a *richly-typed functional language*.

The commands of Coq allow

- **to *define functions* or *predicates* (that can be evaluated efficiently)**
- **to *state mathematical theorems* and *software specifications***
- **to *interactively develop formal proofs* of these *theorems***
- **to *machine-check* these *proofs* by a relatively small certification (kernel)**
- **to *extract certified programs* to languages (e.g., Objective Caml, Haskell, Scheme)**

Coq provides *interactive proof methods, decision* and *semi-decision algorithms*. Connections with *external theorem provers* is available.

Coq is a platform for the verification of mathematical proofs as well as the verification of computer programs in CiC.

* Y. Bertot, P. Castéran (2004), Interactive Theorem Proving and Program Development: Coq'Art: CiC (Springer)

3. Verification of Circuits in Proof Assistants: Basics

Verification of Circuits with Co-Induction in Coq

A *hardware* or *software program* is correct („*certified by Coq*“) if it can be *verified* to follow a given *specification* in CIC.

Example: Verification of circuits*

The *structure* and *behaviour of circuits* can mathematically be described by *interconnected finite automata* (e.g., Mealy machines). In circuits, one has to cope *with infinitely long temporal sequences* of data (*streams*).

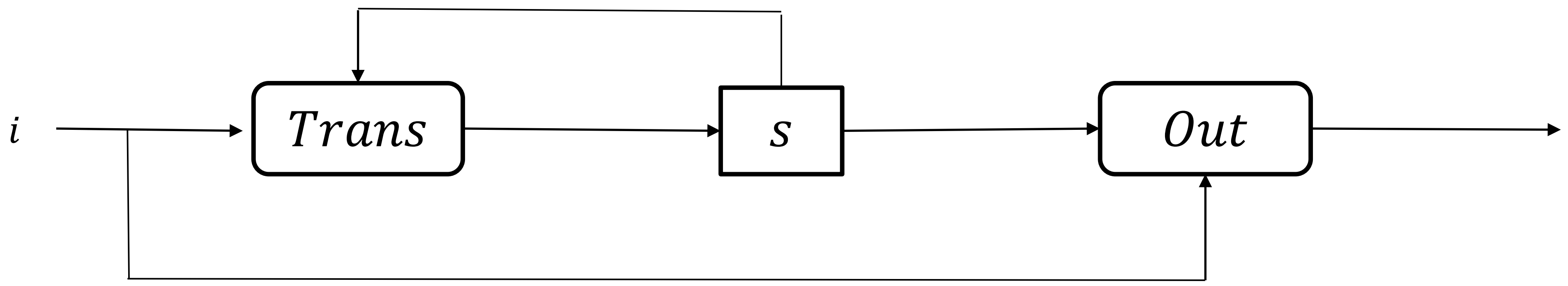
A *circuit* is correct iff, under certain conditions, the *output stream* of the *structural automaton* is *equivalent* to that of the *behavioural automaton*.

Therefore, *automata theory* must be *implemented* into CiC with the *co-inductive type of streams*.

* S. Coupet-Grimal, L. Jakubiec (1996): Coq and Hardware Verification: a Case Study (TPHOLs ‚96, LCNS 1125, 125-139)

Specification of Mealy Automata

A Mealy automaton is a 5-tuple $(I, O, S\ Trans, Out)$ with *input set* I, *output set* O, *state set* S, *transition function* $Trans : I \times S \rightarrow S$, and *output function* $Out : I \times S \rightarrow O$.



Given an *initial state s* , the *Mealy machine* computes an *infinite output sequence* („*stream*“) in response to an *infinite input sequence* („*stream*“).

Implementation of Mealy Automata in CiC

```
Variables I, O, S : Set .
Variable Trans : I → S → S.
Variable Out : I → S → O.

CoFixpoint Mealy : (Stream I) → S → (Stream O) := [inp, s]
    (Cons (Out (Head inp) s) (Mealy (Tail inp)(Trans (Head inp) s)).
```

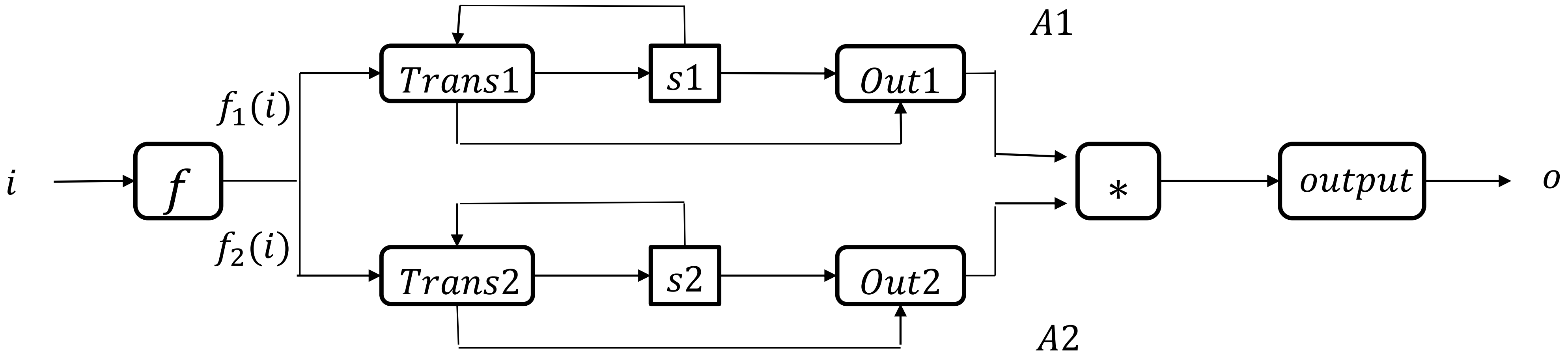
The first element of the *output stream* is the result of the *application* of the *output function Out* to the first input (the *head* of the *input* stream *inp*) and to the *initial state s*. The *tail* of the *output stream* is then computed by a *recursive call* to *Mealy* on the *tail* of the *input stream* and the *new state*. This new state is given by the function *Trans*, applied to the *first input* and the *initial state*.

The *streams* of *all the successive* states from the *initial one s* is obtained similarily:

```
CoFixpoint States : (Stream I) → S → (Stream S) := [inp, s]
    (Cons s (States (Tail inp)(Trans (Head inp) s))).
```

Network of Automata

In a network, *automata* are *inter-connected* by *parallel composition, sequential composition*, and *feedback composition of synchronous sequential devices*.



In the *parallel composition* of two *Mealy automata* $A1$ and $A2$, $f = (f_1, f_2)$ builds from the current input i the *pair of inputs* $(f_1(i), f_2(i))$ for $A1$ and $A2$, $output$ computes the *global outputs* of $A1$ and $A2$.

Implementation of Parallel Automata in CiC

```
Variables I1, I2, O1, O2, S1, S2, I. O : Set
Variable Trans1 : I1 → S1 → S1. Variable Trans2 : I2 → S2 → S2.
Variable Out1 : I1 → S1 → O1.   Variable Out2 : I2 → S2 → O2.
Variable f : I → I1*I2.         Variable f : O → O1*O2.
Local A1 := (Mealy Trans1 Out1). Local A2 := (Mealy Trans2 Out2).

Definition parallel : (Stream I) → S1 → S2 := [inp, s1, s2]
    (Map output (Prod (A1 (Map Fst (Map f inp)) s1)
                      (A2 (Map Snd (Map f inp)) s2))).
```

The *initial states* of automata $A1$ and $A2$ are $s1$ and $s2$. The *input* of $A1$ is obtained by mapping the first projection Fst on the stream resulting from the mapping of the function f on the *global stream* inp. Then $(A1(Map\ Fst\ (Map\ f\ inp))s1)$ is the *output stream* $A1$. That of $A2$ is defined similarly. Finally, the *parallel composition* is obtained by mapping the function $output$ on the *product* of the *output streams* of $A1$ and $A2$.

Invariant Relations of Mealy Automata*

The *equivalence* of *structure* and *behaviour of circuits* can be proved by certain *invariant relations* of *states* and *streams* in the corresponding Mealy automata.

Consider two *Mealy automata* $A1 = (I, O, S_1, Trans1, Out1)$ and $A2 = (I, O, S_2, Trans2, Out2)$ with the *same input set* and the *same output set*. Given p streams, a *relation* which holds for all p-tuples of elements at the same rank is called an *invariant* of these p streams.

In CiC, an *invariant relation P* with respect to *input set I* and the *state sets* S_1 and S_2 can be defined by co-induction:

```
CoInductive Inv [P : I → S1 → S2 → Prop] :
                (Stream I) → (Stream S1) → (Stream S2) → Prop :=
  C_Inv : (inp : (Stream I))(st1 : (Stream S1))(st2 : (Stream S2))
          (P (Head inp) (Head st1) (Head st2)) →
          (Inv P (Tail inp) (Tail st1) (Tail st2)) →
          (Inv P inp st1 st2).
```

*S. Coupet-Grimal, L. Jakubier, Hardware Verification using co-induction in Coq (Laboratoire d'Informatique de Marseille, URA CNRS 1787)

Invariant State Relation of Mealy Automata in CiC

Let R be a relation on the state space $S_1 \times S_2$ and P a relation on $I \times S_1 \times S_2$.

***R* is *invariant* under *P* for the *automata A*1 and *A*2 iff**

$$\forall i \in I\ \forall s_1 \in S_1\ \forall s_2 \in S_2$$
$$(P(i, s_1, s_2) \wedge R(s_1, s_2)) \Rightarrow R(\, Trans1\ (i, s_1), Trans2\ (i, s_2)).$$

The *invariance* of relation *R* can be implemented into CIC :

```
Definition Inv_under := [P : I→ S1 → S2 → Prop][R : S1 → S2 → Prop]
    (i : I)(s1 : S1)(s2 : S2)
    (P i s1 s2) → (R s1 s2) →(R (Trans 1 i s1)(Trans2 i s2).
```

An *output relation* is strong enough to induce the *equality of the outputs* of two automata:

```
Definition Output_rel := [R : S1 → S2 → Prop]
                         (i : I)(s1 : S1) (s2 : S2)
                         (R s1 s2) →(Out1 i s1)=(Out2 i s2).
```

Proof Scheme for Circuit Correctness.

The correctness of a circuit is proved by the *equivalence* of its *structure* and *behaviour* which are represented by two *composed Mealy automata*. The *equivalence of composed Mealy automata* can be proved by the *equivalence lemma of invariant relations* (which is also represented in CiC) :

If *R* is an *output relation invariant* under *P* that holds for the *initial states*, if *P* is an *invariant* for the *common input stream* and the *state streams* of each automata, then the *two output streams* are *equivalent*.

```
Lemma Equiv_2_Mealy :
(P : I → S1 → S2 → Prop)(R : S1 → S2 → Prop)
(Output_rel R) → (Inv_under P R) → (R s1 s2) →
(inp : (Stream I)) (s1 : S1) (s2 : S2)
(Inv P inp (States Trans1 Out1 inp s1)(States Trans2 Out2 inp s2)) →
(EqS (A1 inp s1) (A2 inp s2)).
```

Proof by co-induction

4. Verification of Circuits in Proof Assistants: Application

Certification of a 4 by 4 Switch Fabric

A *switch fabric* is a *network topology* in which *nodes* interconnect via one or more *switches*. The *switching element* performs *switching of data* from 4 *input ports* to 4 *output ports* and *arbitrating data clashes* according to the *output port requests* made by the input ports.*

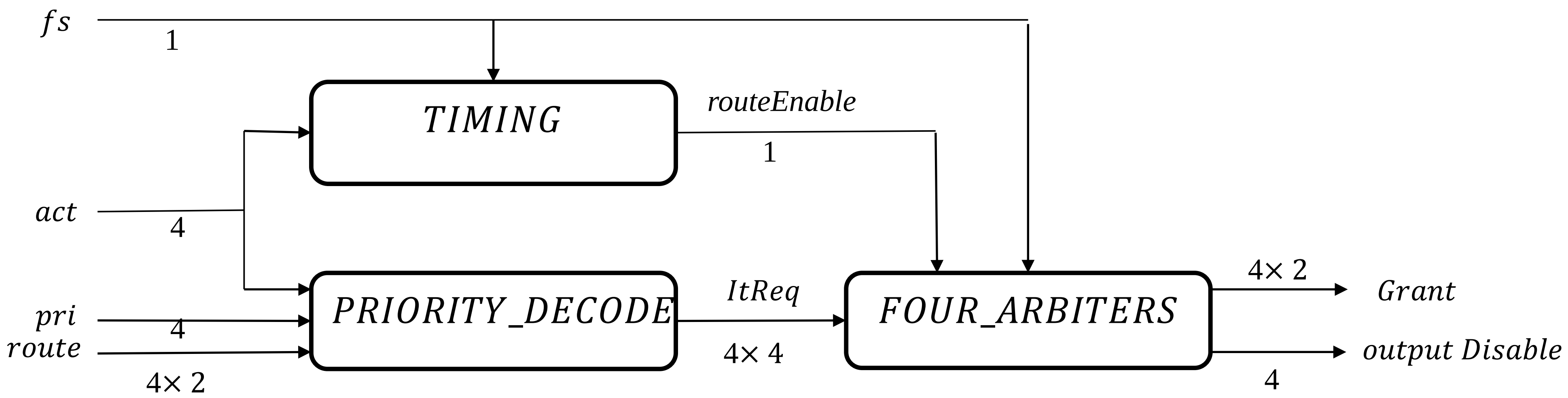
The most significant part for *verification* is the Arbitration Unit. It decodes *requests* from *input ports* and *priorities* between data to be sent, and then performs *arbitration*.

* Local area network based on ATM (Systems Research Group, Cambridge University)

Structure of the Arbitration Unit

The arbiration unit is the interconnection of three modules:

- ***FOUR_ARBITERS*** **performs the *arbitration* for *all output ports* (following Round Robin algorithm)**
- ***TIMING determines when*** **the *arbitration* process can be triggered.**
- ***PRIORITY_DECODE*** **decodes the *requests* and filters them according to their *priority***



Outline of the Proof of Correctness*

	The *correctness* of a switch fabric requires an *equivalence proof* of its *structural automaton* and *behavioural automaton*. It follows from the *verification* of its *modules* that compose the *Arbitration* unit.
(1)	<u>Proof</u> that the *behavioural automata* for *TIMING*, *FOUR_ARBITERS*, and *PRIORITY_DECODE* are <u>*equivalent*</u> the three corresponding *structural automata*.
(2)	<u>Construction</u> of the *global structural automaton structure_ARBITRATION* by *interconnecting* the *structural automata* of the the three *modules TIMING*, *FOUR_ARBITERS*, and *PRIORITY_DECODE* .
(3)	<u>Construction</u> of the *global behavioural automaton Composed_Behaviours* by *interconnecting* the *behavioural automata* of the the three *modules TIMING*, *FOUR_ARBITERS*, and *PRIORITY_DECODE* .
(4)	<u>Proof</u> that *Composed_Behaviours* and *structure_ARBITRATION* are *equivalent* (which follows from (1) and by applying the *lemmas* stating that the *equivalence of automata* is a *congruence* for the *composition rules*).
(5)	<u>Proof</u> that *Composed_Behaviours* is *equivalent* to the *expected behaviour Behaviour_ARBITRATION*. (*Composed_Behaviours* is more abstract than *structure_ARBITRATION* .)
(6)	The <u>equivalence</u> of *Behaviour_ARBITRATION* and *structure_ARBITRATION* is obtained from (4) and (5) by using the *transitivity* of of the *equivalence* on the *streams*.

* S. Coupet-Grimal, L. Jakubier, Hardware Verification using co-induction in Coq (Laboratoire d‘Informatique de Marseille, URA CNRS 1787)

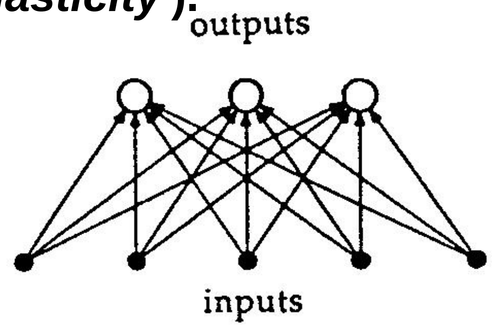
Advantages of the Coq Proof Assistent for Verification of Software/Hardware

- In Coq, a ***verification of a computer program*** is as *strong* and *save* as a ***mathematical proof in a constructive formalism***.
- The use of Coq ***dependent types*** provide *precise* and *reliable specifications*.
- The use of Coq ***co-inductive types*** provide a *clear modelling* of *streams* in *circuits* (without introducing any temporal parameter).
- The use of Coq ***co-induction*** allows to capture the *temporal aspects* of the *proof processes* in one *lemma*.
- The ***hierarchical*** and ***modular approach*** allows *correctness results* in a ***complex verification process*** related to *pre-proven components*.

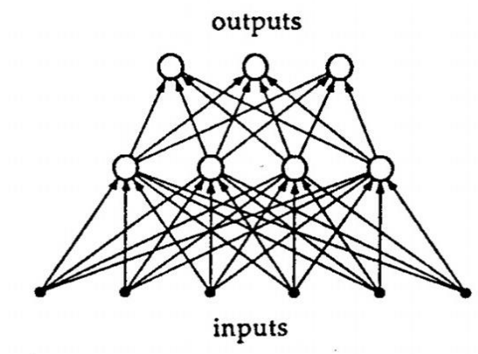
5. Verification of Machine Learning in Proof Assistants

Neural Networks and Learning Algorithms

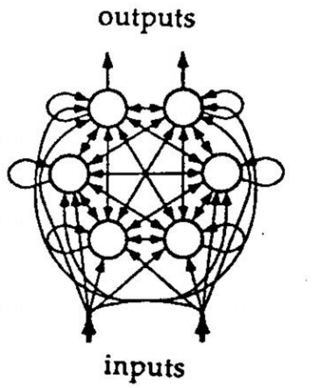
Neural networks **are** ***complex systems*** **of** ***firing*** **and** ***non-firing neurons*** **with** ***topologies*** **like brains. There is no central processor (‚mother cell'), but a** ***self-organizing information flow*** **in cell-assemblies according to rules of synaptic interaction (‚*synaptic plasticity*').**



Feedforward with one synaptic layer



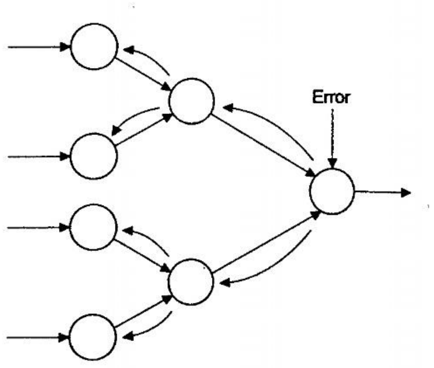
Feedforward with two synaptic layers (Hidden Units)

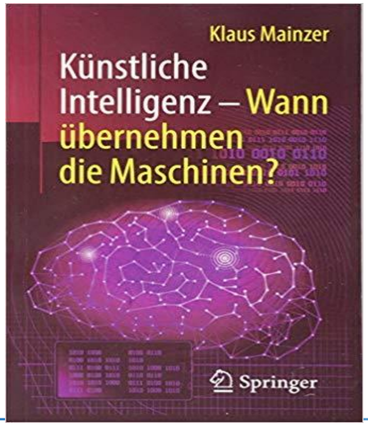


Feedback of recurrent neural network (RNN)

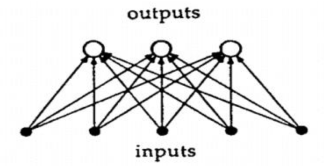
Learning algorithms:

- ***supervised***
- ***non-supervised***
- ***reinforcement***
- ***deep learning***

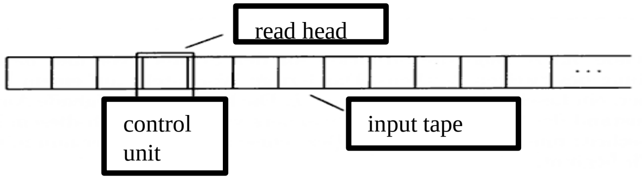




Equivalence of Neural Networks, Automata, and Machines

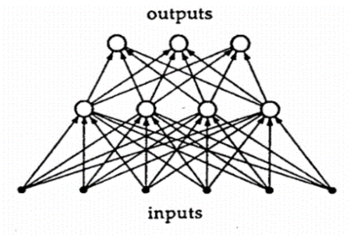


digital **McCulloch-Pitts net with** ***integer weights***

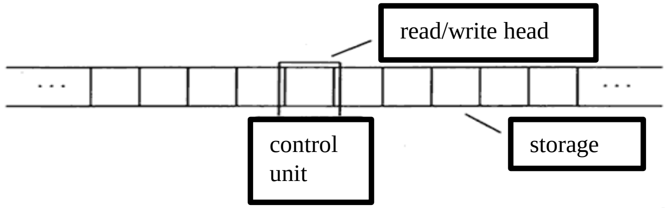


finite automaton

recognition of ***computable („regular") languages*** *

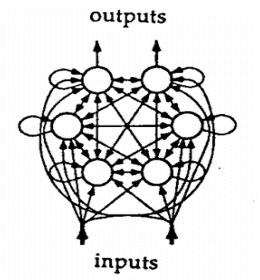


digital **net with** ***rational weights***

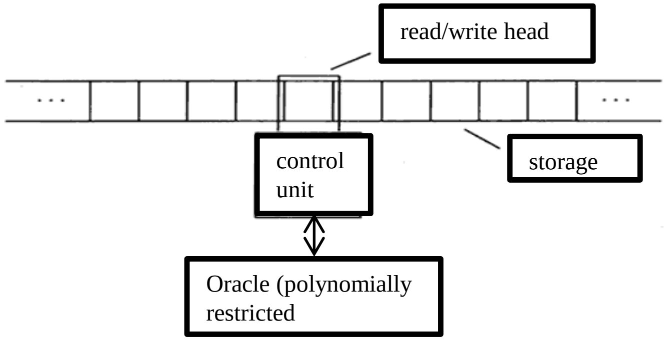


Turing machine

recognition of ***computable („recursive") languages*** **(Chomsky grammar)** **



analog **recurrent net with** ***real weights***



Turing oracle machine

recognition of ***incomputable (non-recursive) languages*** **(natural languages)** ***

* S.C. Kleene (1956); **, *** H.T. Siegelmann, E.D. Sontag (1995), (1994); K. Mainzer (2018)

Verification of Neural Networks and Learning Algorithms

Digital neural networks **are equivalent to appropriate** ***automata*** **(with respect to certain cognitive tasks).**

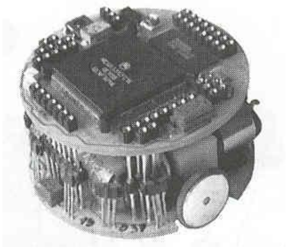
The ***structure*** **and** ***behaviour of automata*** **can be implemented into the** ***Calculus of inductive Constructions*** **(CiC).**

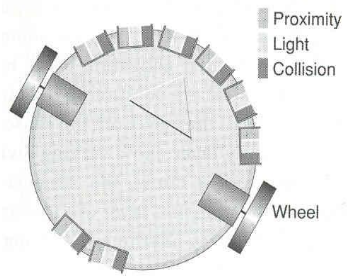
Thus, in principle, their ***equivalence*** **could** ***verify*** **the** ***correctnesss of circuits of automata*** **and, therefore, the** ***correctness of neural networks in Coq*****.**

Even ***analog neural networks*** **(with real weights) could be implemented into CiC extended by** ***higher inductively defined structures*** **in HoTT to** ***verify*** **their** ***correctness in Coq*****.**

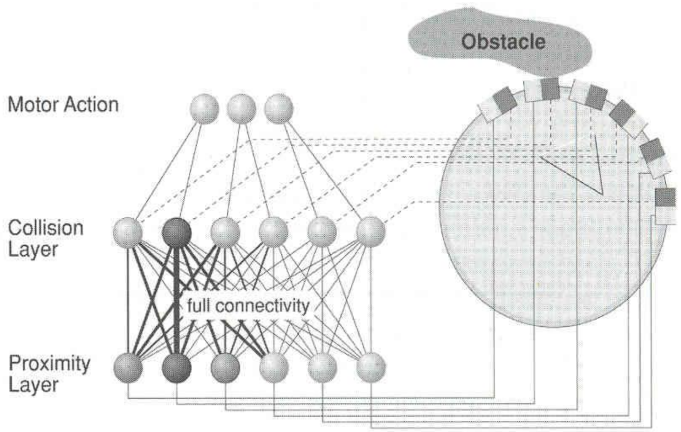
Machine Learning and Autonomous Cars

A simple *robot* with diverse *sensors* (e.g., proximity, light, collision) and *motor equipment* can generate *complex behavior* by a *self-organizing neural network*:





In the case of *collision*, the *connections* between the *active nodes* of *proximity* and *collision layer* are reinforced by *Hebbean learning*: *A behavioral pattern emerges!*



Pfeifer/Scheier 1999

Explosion of Parameters and Big Data generates a Black Box:

"Does your car have any idea why my car pulled it over?"

How many real world accidents **are required to** ***teach machine-learning based autonomous vehicles*****?**

Who should be ***responsible*** **when there is an accident involving autonomous vehicles** ***(ethical and legal challenges)*****?**

We need provability, explainability and accountability of neural networks!

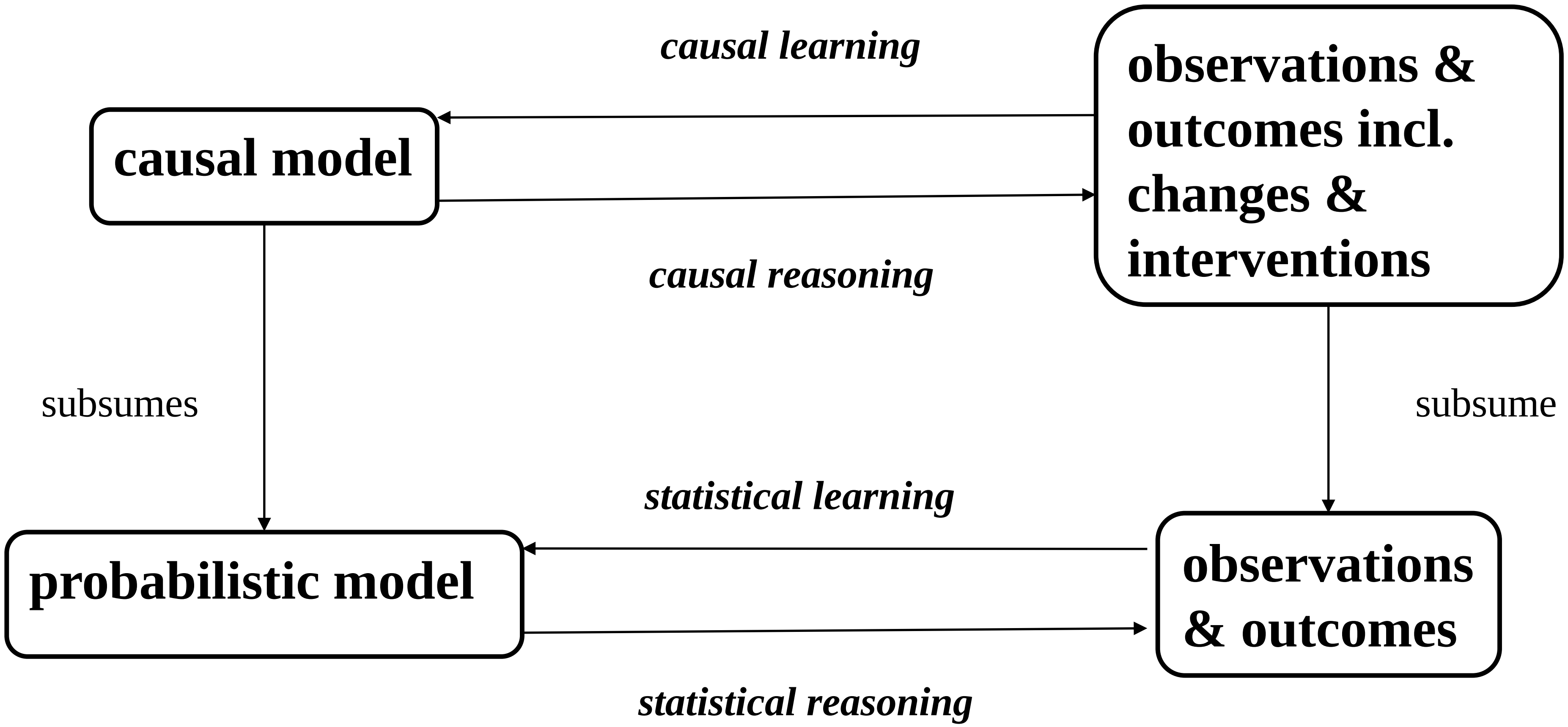
Blindness of Machine Learning and Big Data

Without explanation, **big neural networks with *large statistical training data* (Big Data) are *black boxes*.**

***Statistical data correlations* do not replace *explanations of causes and effects*.**

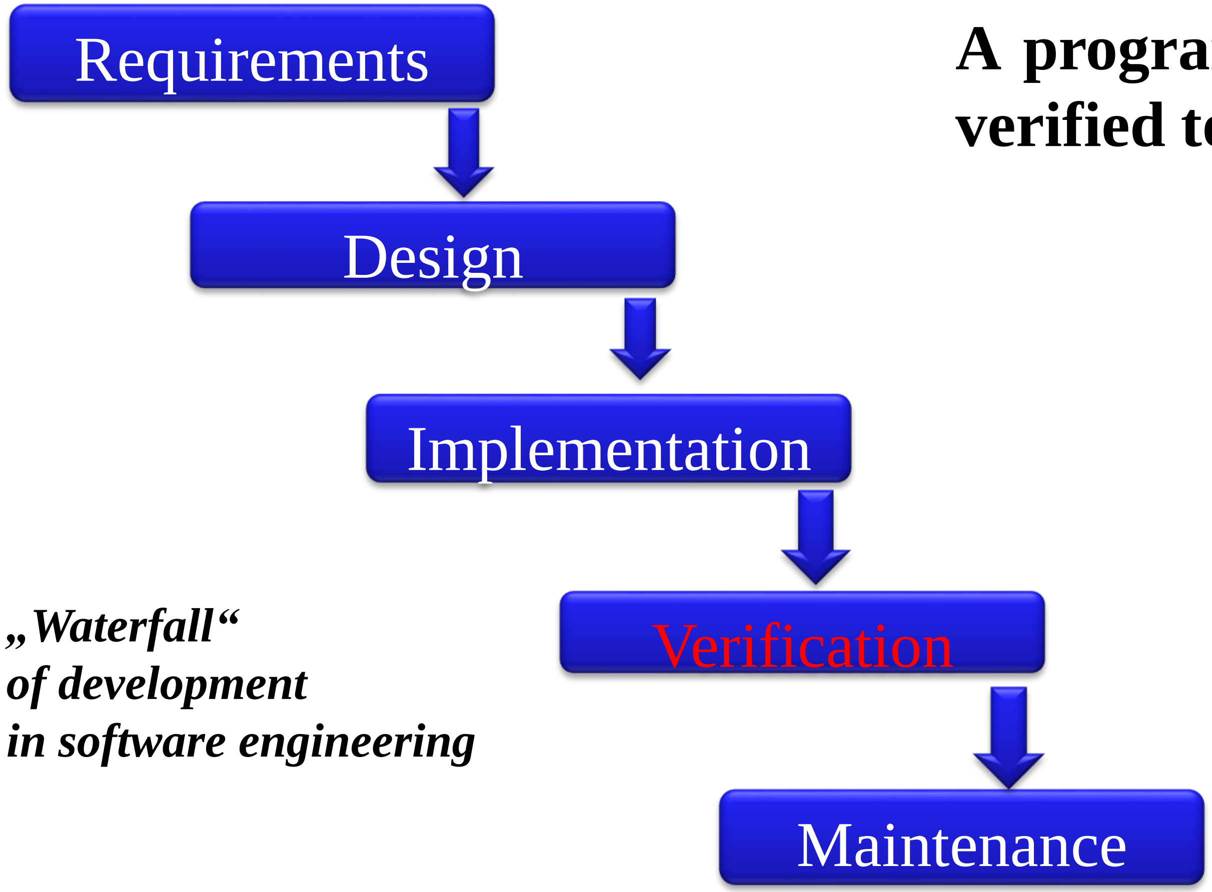
Their *evaluation* needs *causal modeling* for answering questions of *accountability and responsibility*.

Causal Modeling and Machine Learning



Peters et al. 2017, p. 6

Correctness of Certified Programs with Proof Assistants



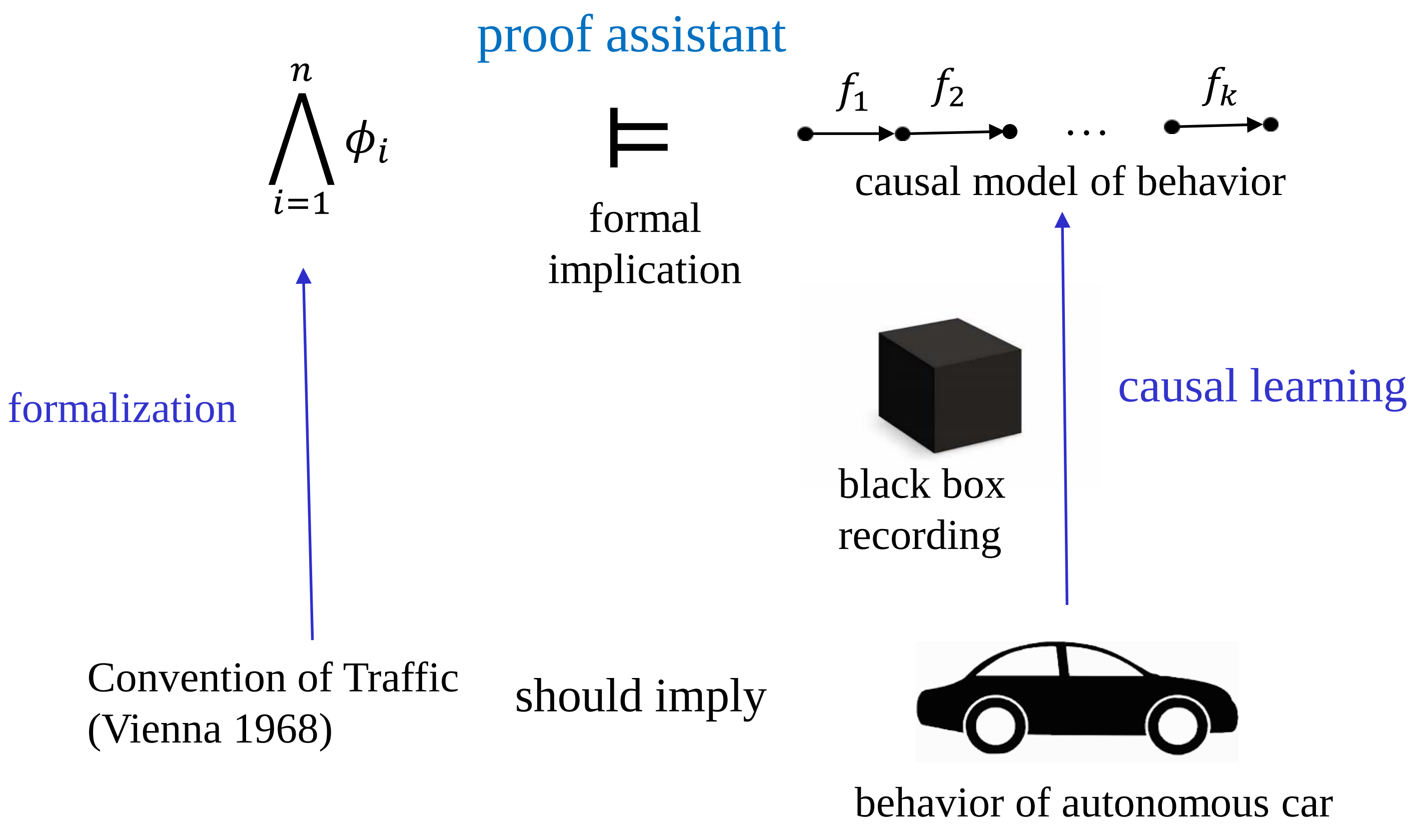
„Waterfall“
of development
in software engineering

A program is <u>correct</u> („*certified*“) if it can be verified to follow a given specification.

A <u>proof assistant</u> proves the *correctness of a computer program* in a *consistent formalism* like a *constructive proof in mathematics* (e.g., Coq, Agda, MinLog).

Therefore, *proof assistants* are the *<u>best formal verification</u>* of *correctness for certified programs*.

Responsible AI in Autonomous Car Driving with Causal Learning and Proof Assistant



Certified Programs with Theorem Proving and Causal Learning

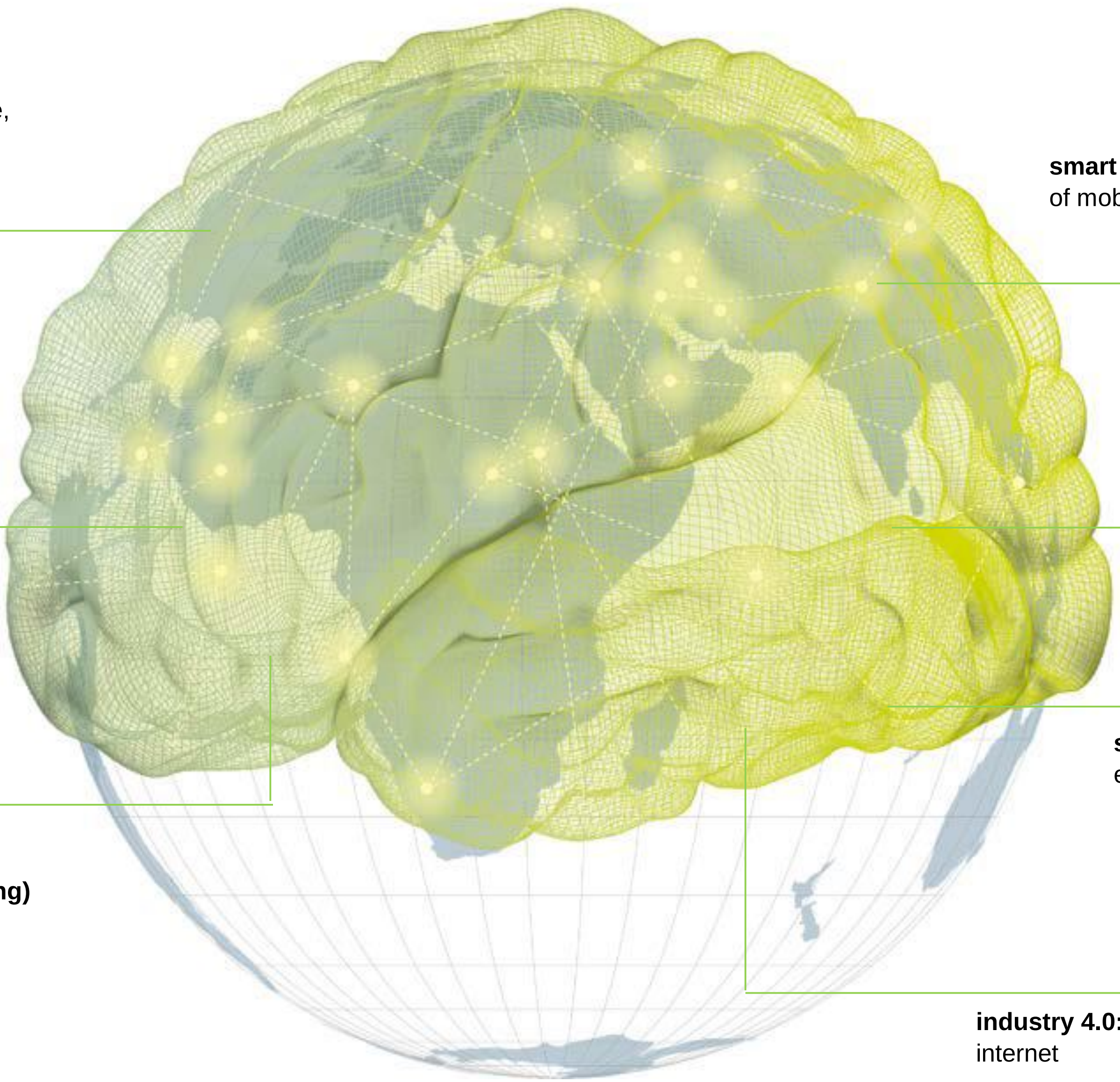
Statistical machine learning works,
but we can't understand the underlying reasoning.

***Machine learning technique* is akin to *testing*,**
but it is *not enough for safety-critical systems.*

⟹ *Combination* of *<u>causal learning</u>* and
<u>constructive AI with certified programs</u>
(theorem proving and causal learning)

6. Perspectives of Responsible Artificial Intelligence

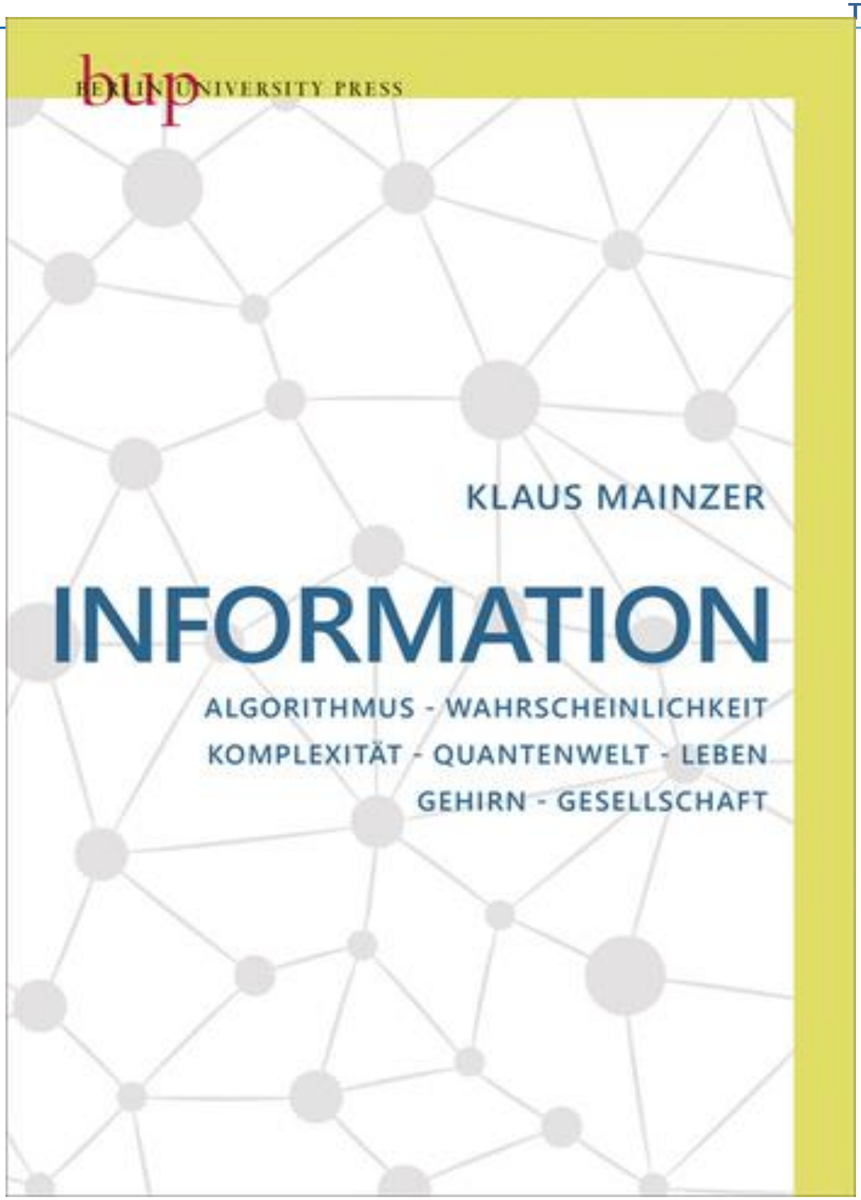
Internet of Things with Exploding Data

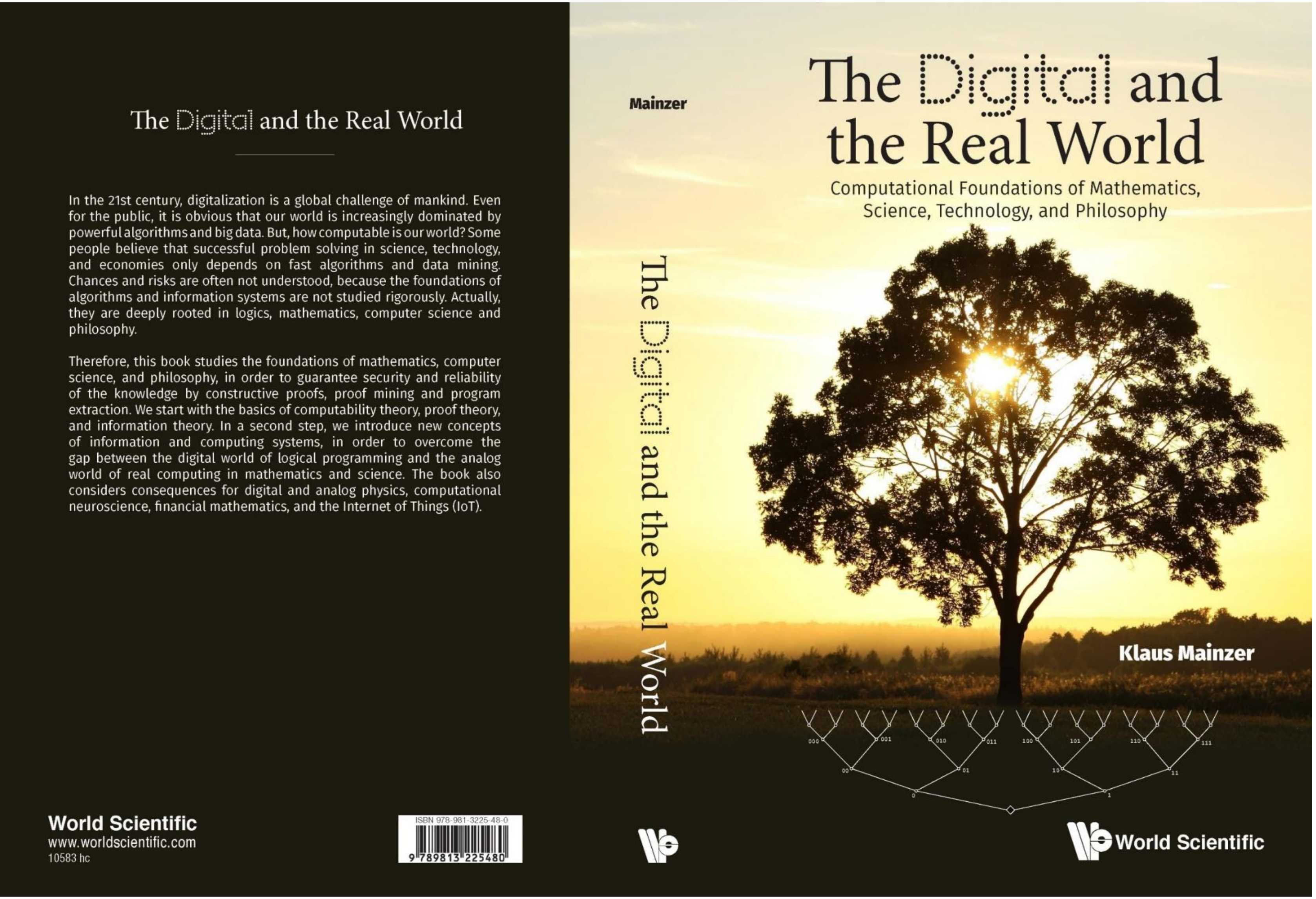


We need more *explainability*, *verification*, and *governance of machine learning and Big Data* to master the *increasing complexity of our civilization*!


Künstliche Intelligenz – Wann übernehmen die Maschinen?
Jeder kennt sie. Smartphones, die mit uns sprechen, Armbanduhren, die unsere Gesundheitsdaten aufzeichnen, Arbeitsabläufe, die sich automatisch organisieren, Autos, Flugzeuge und Drohnen, die sich selber steuern, Verkehrs- und Energiesysteme mit autonomer Logistik oder Roboter, die ferne Planeten erkunden, sind technische Beispiele einer vernetzten Welt intelligenter Systeme. Sie zeigen uns, dass unser Alltag bereits von KI-Funktionen bestimmt ist.
Auch biologische Organismen sind Beispiele von intelligenten Systemen, die in der Evolution entstanden und mehr oder weniger selbstständig Probleme effizient lösen können. Gelegentlich ist die Natur Vorbild für technische Entwicklungen. Häufig finden Informatik und Ingenieurwissenschaften jedoch Lösungen, die sogar besser und effizienter sind als in der Natur.
Seit ihrer Entstehung ist die KI-Forschung mit großen Visionen über die Zukunft der Menschheit verbunden. Löst die „künstliche Intelligenz“ also den Menschen ab? Dieses Buch ist ein Plädoyer für Technikgestaltung: KI muss sich als Dienstleistung in der Gesellschaft bewähren.
ISBN 978-3-662-48452-4
springer.com
Mainzer
Künstliche Intelligenz – Wann übernehmen die Maschinen?
Klaus Mainzer
Künstliche Intelligenz – Wann übernehmen die Maschinen?
Springer


Klaus Mainzer
Die Berechnung der Welt
Von der Weltformel
zu Big Data
C.H.Beck


bup
BERLIN UNIVERSITY PRESS
KLAUS MAINZER
INFORMATION
ALGORITHMUS - WAHRSCHEINLICHKEIT
KOMPLEXITÄT - QUANTENWELT - LEBEN
GEHIRN - GESELLSCHAFT


The Digital and the Real World
In the 21st century, digitalization is a global challenge of mankind. Even for the public, it is obvious that our world is increasingly dominated by powerful algorithms and big data. But, how computable is our world? Some people believe that successful problem solving in science, technology, and economies only depends on fast algorithms and data mining. Chances and risks are often not understood, because the foundations of algorithms and information systems are not studied rigorously. Actually, they are deeply rooted in logics, mathematics, computer science and philosophy.
Therefore, this book studies the foundations of mathematics, computer science, and philosophy, in order to guarantee security and reliability of the knowledge by constructive proofs, proof mining and program extraction. We start with the basics of computability theory, proof theory, and information theory. In a second step, we introduce new concepts of information and computing systems, in order to overcome the gap between the digital world of logical programming and the analog world of real computing in mathematics and science. The book also considers consequences for digital and analog physics, computational neuroscience, financial mathematics, and the Internet of Things (IoT).
World Scientific
www.worldscientific.com
10583 hc
ISBN 978-981-3225-48-0
Mainzer
The Digital and the Real World
The Digital and the Real World
Computational Foundations of Mathematics, Science, Technology, and Philosophy
Klaus Mainzer
World Scientific


}essentials{
Klaus Mainzer
Wie berechenbar ist unsere Welt
Herausforderungen für Mathematik, Informatik und Philosophie im Zeitalter der Digitalisierung
Springer VS