Constructive Operations Research

*Bridging Logic, Mathematics, Computer Science, and Philosophy*

Klaus Mainzer
TUM Emeritus of Excellence
TUM Graduate School of Computer Science
Technische Universität München (TUM)
What is Operations Research?

Since the 2nd World War, operations research has been an interdisciplinary research field of mathematics, computer science, and management which should optimize algorithms with economic and societal relevance.
Why Constructive Operations Research (CORE)?

*Operations research aims at efficient and reliable algorithms to solve problems in economy and society.*

*Trust & security can only be guaranteed by constructive proofs.*

Therefore, besides mathematics, computer science and management, also logic and philosophy come in.
1. Basics of Computability
2. Computability in Higher Types
3. Proof Mining and Program Extraction
4. From Brouwer’s Creative Subject to Fan Theorem
5. Reverse Mathematics and Constructivity
6. Intuitionistic Type Theory and Proof Assistants
7. Univalent Foundation of Mathematics
9. Bridging Logic, Mathematics, Computer Science and Philosophy
1. Basics of Computability
C. Babbage: Computer and Operations Research

The British polymath Charles Babbage (1791-1871), mathematician, philosopher, inventor, and mechanical engineer, is well-known as „father of the computer“ who originated the concept of a digital programmable computer with his first mechanical models (e.g., difference machine, analytical engine).

Babbage also published *On the Economy of Machinery and Manufactures* (1832), on the organization of industrial production. It was an influential early work of operational research with great impact on political economy.
Turing’s Theory of Computability

A *Turing machine* is a formal procedure, consisting of
a) a *control box* in which a *finite program* is placed,
b) a potential *infinite tape*, divided lengthwise into squares,
c) a device for *scanning*, or *printing* on one square of the tape at a time, and for *moving* along the tape or *stopping*, all under the command of the *control box*.

Every *computable procedure* (algorithm) can be simulated by a *Turing machine* (*Church’s thesis*). Every *Turing program* can be simulated by a *universal Turing machine* (*general purpose computer*).
2. Computability in Higher Types
Approximations of Computable Functionals in Information Systems

In order to describe approximations of abstract objects like functionals by finite ones, we use an information system with a countable set $A$ of bits of data ("tokens") (Scott 1982, Schwichtenberg/Wainer 2012). Approximations need finite sets $U$ of data which are consistent with each other. An "entailment relation" expresses the fact that the information of a consistent set $U$ of data is sufficient to compute a bit of information ("token"):

An information system is a structure $(A, \text{Con}, \vdash)$ with a countable set $A$ ("tokens"), non-empty set $\text{Con}$ of finite ("consistent") subsets of $A$ and subset $\vdash$ of $\text{Con} \times A$ ("entailment relation") with

1. $U \subseteq V \in \text{Con} \Rightarrow U \in \text{Con}$
2. $\{a\} \in \text{Con}$
3. $U \vdash a \in \text{Con} \Rightarrow U \vdash a$
4. $a \in U \in \text{Con} \Rightarrow U \vdash a$
5. $U, V \in \text{Con} \Rightarrow \forall a \in V (U \vdash a) \Rightarrow (V \vdash b \Rightarrow U \vdash b)$

The ideals ("objects") of an information system $(A, \text{Con}, \vdash)$ are defined as subjects $x$ of $A$ with

1. $U \subseteq x \Rightarrow U \in \text{Con}$ (x is consistent)
2. $x \supseteq U \vdash a \Rightarrow a \in x$ (x is deductively closed)

Example: The deductive closure $\bar{U} := \{a \in A \mid U \vdash a\}$ of $U \in \text{Con}$ is an ideal.
Types are built from base types by the formation of function types $\rho \to \sigma$. For every type $\rho$, the information system $\mathcal{C}_\rho = (\mathcal{C}_\rho, \text{Con}_\rho, \vdash_\rho)$ can be defined. The ideals $x \in \mathcal{C}_\rho$ are the partial continuous functionals of type $\rho$. Since $\mathcal{C}_{\rho \to \sigma} = \mathcal{C}_\rho \to \mathcal{C}_\sigma$, the partial continuous functionals of type $\rho \to \sigma$ will correspond to the continuous functions from $\mathcal{C}_\rho$ to $\mathcal{C}_\sigma$ with respect to the Scott topology.

A partial continuous functional $x \in \mathcal{C}_\rho$ is computable iff it is recursive enumerable as set of tokens.

Partial continuous functionals of type $\rho$ can be used as semantics of a formal functional programming language: Every closed term of type $\rho$ in the programming language denotes a computable partial continuous functional of type $\rho$, i.e. a recursive enumerable consistent and deductively closed set of tokens.

Another approach uses recursive equations to define computable functionals (Berger, Eberl, Schwichtenberg 2003).
3. Proof Mining and Program Extraction
Proofs as Verification of Truth

**Theorem:** There are infinitely many prime numbers.

The predicate \( P(x) \equiv 'x \text{ is a prime number}' \) can be expressed in a quantifier-free way as primitive recursive predicate.


Assume there are finitely many prime numbers \( p \leq x \)

construct \( a := 1 + \prod_{p \leq x} p \) \( p \text{ prime} \)

\( \Rightarrow a \text{ cannot be prime} \) (because \( a > p \) for all \( p \leq x \))

\( \Rightarrow a \text{ contains a prime factor} \) (by the decomposition of every number into prime factors)

\( q \leq a \) with \( q > x \) (otherwise \( q \) is a prime factor \( q \leq x \))

\( \Rightarrow \text{contradiction to assumption!} \)
Proofs are more than Verification!

„What more do we know if we have proved a theorem by restricted means than if we merely know that it is true?“

G. Kreisel: Unwinding of Proofs

Consider an existential theorem $A \equiv \exists x B(x)$ (closed):

A weaker requirement is to construct a list of terms $t_1, \ldots, t_n$ which are candidates for $A$, such that $B(t_1) \lor \cdots \lor B(t_n)$ holds. More general:

If $A \equiv \forall x \exists y B(x, y)$, then one can ask for an algorithm $p$ such that

$\forall x B(x, p(x))$ holds

or – weaker – for a bounding function $b$ that $\forall x \exists y \leq b(x) B(x, y)$. 
Automatic Program Extraction with MINLOG

MINLOG is an interactive proof system which is equipped with tools to extract functional programs directly from proof terms. The system is supported by automatic proof search and normalization by evaluation as an efficient term rewriting device.

Example: Existence proof for “list reversal”

Write $vw$ for the result $v \ast w$ of appending the list $w$ to the list $v$,
Write $vx$ for the result $v \ast x$ of appending one element list $x$ to the list $v$,
Write $xv$ for the result $x :: v$ of writing one element $x$ in front of a list $v$
Assume Init Rev: $R(\text{nil}, \text{nil})$
GenRev: $\forall v, w, x (Rvw \rightarrow R(vx, xw))$

Proposition: $\forall v \in T \exists w \in T Rvw$ („Existence of list $w$ with reverse order of given list $v“$)
Proof: Induction on the length of $v$
Goal of Program Extraction in Computer Science: Metatheorems of Software

Customers want software which solves a problem. Thus, they require a proof that it works. Suppliers answer with a proof of the existence of a solution to the specification of the problem. The proof has been automatically extracted from the formal specification of the problem by a proof mining software (e.g., MINLOG).

But, the question arises whether the extraction mechanism of the proof is itself, in general, correct. The metatheorem of soundness guarantees that every formal proof can be realized by a normalized extracted term.
4. From Brouwer‘s Creative Subject to FanTheorem
Intuitionistic Philosophy of Creative Subject

According to Brouwer, mathematical truth is founded by construction of a creative subject. Following Kant, mathematical construction can only be realized in a finite process, step by step in time like counting in arithmetic. Thus, for Brouwer, mathematical truth depends on finite stages of realization in time by a creative subject (in a definition of Kripke and Kreisel 1967):

The creative subject has a proof of proposition $A$ at stage $m$ ($\Sigma \vdash_m A$) iff

(CS1) For any proposition $A$, $\Sigma \vdash_m A$ is a decidable function of $A$, i.e. $\forall x \in \mathbb{N} (\Sigma \vdash x A \lor \neg \Sigma \vdash x A)$

(CS2) $\forall x, y \in \mathbb{N} (\Sigma \vdash x A \rightarrow (\Sigma \vdash x+y A)$

(CS3) $\exists x \in \mathbb{N} (\Sigma \vdash x A) \leftrightarrow A$

A weaker version of CS3 is G. Kreisel’s “Axiom of Christian Charity” (1967)

(CC) $\neg \exists x \in \mathbb{N} (\Sigma \vdash x A) \rightarrow \neg A$. 
Fan Principle and Fan Theorem

The *fan principle* states that for every fan $T$ in which every *branch* at some point satisfies a property $A$, there is a *uniform bound* on the depth at which this property is met. Such a property is called a *bar* of $T$.

**Fan Principle:**

\[
\forall \alpha \in T \quad \exists m \quad \forall \alpha \in T \quad \exists n \leq m \quad A(\alpha(n))
\]

with $\alpha$ *choice sequences* and $\alpha(n)$ the *initial segment* of $\alpha$ with the first $m$ elements.

**Fan Theorem:**

Every *continuous real function* on a *closed interval* is *uniformly continuous*.

**Proof:** *Fan Principle*
Brouwer‘s Bar Principle for the Universal Spread

The bar principle provides intuitionistic mathematics with an induction principle for trees. It expresses a well-foundedness principle for spreads with respect to decidable properties:

\[
\forall \alpha \forall n A(\alpha(n)) \lor \neg A(\alpha(n)) \land \forall \alpha \exists n A(\alpha(n)) \land \forall \alpha \forall n A(\alpha(n)) \rightarrow B(\alpha(n)) \land \\
\forall \alpha \forall n (\forall m B(\alpha(n) \cdot m) \rightarrow B(\alpha(n))) \rightarrow B(\varepsilon)
\]

with \( \varepsilon \) empty sequence.
Intuitionism and Functional Interpretation

In functional interpretation of proofs, the intuitionistically unexplained notion of construction is defined by computable functionals of finite type with sets of finite approximations which are (primitive) recursively enumerable.

Thus, the intuitionistically unexplained notion of a “constructive proof at a finite stage” is explained by finite approximation of some computable functional at a finite stage.

As computable functionals of finite type are (mathematical) ideals of information systems, constructions do not depend on “mental actions of human creative subjects” (i.e. psychology and epistemology), but on finite processes of (mathematically definable) information systems. Humans (Brouwer’s “creative subject”) and computers (“machines”) are only epistemical resp. technical examples of information systems:

Constructivity and Computability are founded in Information Systems!
5. Reverse Mathematics and Constructivity
Reverse Mathematics in Antiquity

Since Euclid (Mid-4th century – Mid 3rd century BC), axiomatic mathematics has started with axioms to deduce a theorem. But the “forward” procedure from axioms to theorems is not always obvious. How can we find appropriate axioms for a proof starting with a given theorem in a „backward“ (reverse) procedure?

Pappos of Alexandria (290-350 AC) called the “forward” procedure as “synthesis” with respect to Euclid’s logical deductions from axioms of geometry and geometric constructions (Greek: “synthesis”) of corresponding figures. The reverse search procedure of axioms for a given theorem was called “analysis” with respect to decomposing a theorem in its necessary and sufficient conditions and the decomposition of the corresponding figure in its building blocks.
Classical Reverse Mathematics

Reverse mathematics is a modern research program to determine the minimal axiomatic system required to prove theorems. In general, it is not possible to start from a theorem $\tau$ to prove a whole axiomatic subsystem $T_1$. A weak base theory $T_2$ is required to supplement $\tau$:

If $T_2 + \tau$ can prove $T_1$, this proof is called a reversal.
If $T_1$ proves $\tau$ and $T_2 + \tau$ is a reversal, then $T_1$ and $\tau$ are said to be equivalent over $T_2$.

Reverse mathematics allows to determine the proof-theoretic strength resp. complexity of theorems by classifying them with respect to equivalent theorems and proofs. Many theorems of classical mathematics can be classified by subsystems of second-order arithmetic $\mathbb{Z}_2$ with variables of natural numbers and variables of sets of natural numbers.
\( \beth_2 \)-Subsystems and Philosophical Research Programs

The five most commonly used \( \beth_2 \)-subsystems in reverse mathematics correspond to philosophical programs in foundations of mathematics with increasing proof-theoretic power starting with the weakest \( \text{RCA}_0 \)-subsystem.

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Philosophical Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{RCA}_0 )</td>
<td>Turing's computability</td>
</tr>
<tr>
<td>( \text{WKL}_0 )</td>
<td>Hilbert's finitistic reductionism</td>
</tr>
<tr>
<td>( \text{ACA}_0 )</td>
<td>Weyl's &amp; Lorenzen's predicativity (with classical logic)</td>
</tr>
<tr>
<td>( \text{ATR}_0 )</td>
<td>Friedman's &amp; Simpson's predicative reductionism</td>
</tr>
<tr>
<td>( \Pi^1_1 - \text{CA}_0 )</td>
<td>Impredicativity</td>
</tr>
</tbody>
</table>

\( \Delta^1_1 - \text{CA}_0 \) yields systems of hyperarithmetic analysis (Feferman et al.) with \( \Delta^1_1 \)-predicativism:

\( T \) is a theory of hyperarithmetic analysis iff

i. its \( \omega \)-models are closed under joins and hyperarithmetic reducibility

ii. it holds in \( \text{HYP}(x) \) for all \( x \)
Constructive Reverse Mathematics

Classical reverse mathematics (Friedmann/Simpson) uses classical logic and classification of proof-theoretic strength with $\text{RCA}_0$ ($\Delta^0_1$-recursive comprehension) as weakest subsystem.

Constructive reverse mathematics (Ishihara et al.) uses intuitionistic logic and Bishop’s constructive mathematics (BISH) as weakest subsystem of a constructive classification (Bishop/Bridges/Vita/Richman).

- **BISH = $\mathsf{Z}_2 + \text{Intuitionistic Logic} + \text{Axioms of Countable, Dependent and Unique Choice}**

- Intuitionistic Mathematics (Brouwer, Heyting et al.):
  - **INT = BISH + Axiom of Continuous Choice + Fan Theorem**

- Constructive Recursive Mathematics (Markov et al.):
  - **RUSS = BISH + Markov’s Principle + Church’s Thesis**

- Classical Mathematics (Hilbert et al.):
  - **CLASS = BISH + Principle of Excluded Middle + Full Axiom of Choice**
6. Intuitionistic Type Theory and Proof Assistants
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 $a, b, c, \ldots$; propositions are represented by $A, B, C, \ldots$.

Examples:

\[
\begin{align*}
\lambda a(\lambda b. a) &: [A] \\
B \rightarrow A &: [A] \\
(\rightarrow I) &: A \rightarrow (B \rightarrow A)
\end{align*}
\]

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 $\text{HA}^0$ and Gödel's system $\text{T}$ of primitive recursive functions of higher type) with primitive identity types, well founded tree types, universe hierarchies and general notions of inductive and inductive–recursive definitions.

His extension increases the **proof-theoretic strength** of the theory and its application to **programming** as well as to **formalization of mathematics**.
Calculus of Constructions (CoC)

CoC is a type theory of Thierry Coquand which can serve as typed programming language as well as constructive foundation of mathematics. With inductive types, the calculus of inductive constructions (CiC) removes impredicativity (cf. Weyl, Lorenzen). It extends the Curry-Howard isomorphism to proofs in the full intuitionistic predicate calculus. CoC has very few basic operators (e.g., \(\lambda\forall\)):

**Logical operators:**

\[
\begin{align*}
A \Rightarrow B & \equiv \forall x: A. B \ (x \notin B) \\
A \land B & \equiv \forall C: P. (A \Rightarrow B \Rightarrow C) \Rightarrow C \\
A \lor B & \equiv \forall C: P. (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C \\
\neg A & \equiv \forall C: P. (A \Rightarrow C) \\
\exists x: A. B & \equiv \forall C: P. (\forall x: A(B \Rightarrow C)) \Rightarrow C
\end{align*}
\]

**Data types:**

- **Booleans:** \(\forall A: P. A \Rightarrow A \Rightarrow A\)
- **Naturals:** \(\forall A: P. (A \Rightarrow A) \Rightarrow (A \Rightarrow A)\)
- **Product** \(A \times B\): \(A \land B\)
- **Disjoint union** \(A + B\): \(A \lor B\)
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 formalization of mathematics as well as the development of programs.
The Language of Proof Assistant Coq

The heart of Coq is a type-checking algorithm in the language of CiC:

Coq objects are sorted into the Prop sort and the Type sort:
- Prop is the sort of propositions (with type Prop):
  * e.g., \( \forall A, B: \text{Prop}, A \rightarrow B \rightarrow A \lor B \)
  
  New predicates can be defined either inductively or by abstracting over existing propositions:
  * e.g., Definition divide \( (x, y:N) := \exists z, x \times z = y \)

- Type is the sort for datatypes and mathematical structures (with type Type):
  Types can be inductive structures or types for tuples or a form for subsets (\( \Sigma \)-types):
  * e.g., the type of even natural numbers \( \{ n:N | \text{even} n \} \)

Coq implements a functional programming language supporting these types:
* e.g., the pairing function of type \( z \rightarrow z \times z \) is written \( \text{fun } x \Rightarrow (x, x) \)
7. Univalent Foundations of Mathematics
From Leibniz’ Analysis Situs to Algebraic Topology

In 1679, Leibniz wrote a letter to Huygens which described his „analysis situs“:

Examples of Leibniz’s characteristica geometrica propria:

\[ A \smallfrown B \] (coincidence)
\[ A \sim B \] (similarity)
\[ A \equiv B \] (congruence)

I am still not satisfied with algebra, because it does not give the shortest methods or the most beautiful constructions in geometry. This is why I believe that, so far as geometry is concerned, we need still another analysis which is distinctly geometrical or linear and which will express situation [situs] directly as algebra expresses magnitude directly. And I believe that I have found the way and that we can represent figures and even machines and movements by characters, as algebra represents numbers or magnitudes. I am sending you an essay which seems to me to be important. (1; 382)

Euler's Seven Bridges of Königsberg Problem and Polyhedron Formula are the field's first theorems.

In modern mathematics, algebraic topology uses tools of abstract algebra to study topological spaces. The basic goal is to find algebraic invariants that classify topological spaces up to homeomorphism, though usually most classify up to homotopy equivalence.
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*. 
## Intuitionistic Type Theory and Homotopy Theory

<table>
<thead>
<tr>
<th>Intuitionistic Type Theory</th>
<th>Homotopy Theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>types $A$</td>
<td>spaces $A$</td>
</tr>
<tr>
<td>terms $a$</td>
<td>points $a$</td>
</tr>
<tr>
<td>$a: A$</td>
<td>$a \in A$</td>
</tr>
<tr>
<td>dependent type $x: A \vdash B(x)$</td>
<td>fibration $B \to A$</td>
</tr>
<tr>
<td>identity type $\text{Id}_A(a, b)$</td>
<td>space of paths from $a$ to $b$</td>
</tr>
<tr>
<td>$p: \text{Id}_A(a, b)$</td>
<td>path $p: a \leftrightarrow b$</td>
</tr>
<tr>
<td>$\alpha: \text{Id}_{\text{Id}_A(a, b)}(p, q)$</td>
<td>homotopy $\alpha: p \Rightarrow q$</td>
</tr>
</tbody>
</table>

In intuitionistic type theory, a term $a: A$ can be understood as an element of the type $A$ or a proof of the proposition $A$ or, in homotopy type theory, as a point of the space $A$. Proofs $p$ of identity between two elements $a, b$ of a type $A$ are geometrically illustrated as paths connecting the corresponding points.

In intuitionistic type theory, all proofs of an identity are not forced to be equal (Hofmann/Streicher 1998): This was shown by a model where each type is interpreted as a groupoid (i.e. a category, in which each morphism is invertible resp. an isomorphism).
Provability, Constructivity, and Computability 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:

\[ \text{type checking } \Rightarrow \text{proof checking in higher categories} \]

(„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).
Foundations of Mathematics

- Intuitionistic Type Theory
- Homotopy Type Theory
- Constructive Proof Theory
  
  (Computable functionals in information systems, constructive reverse mathematics)

- Proof Assistant (Coq, Agda, Minlog)
- Constructive Set Theory
From Financial Crisis to Crisis of Trust in Banks

How far were shortcomings of the policy frameworks of the major central banks responsible for the financial crisis 2008?

“The root problem with conventional currency is all the trust that is required to make it work. The central bank must be trusted not to debase the currency, but the history of fiat currencies is full of breaches of that trust. Banks must be trusted to hold our money and transfer it electronically, but they lend it out in waves of credit bubbles with barely a fraction in reserve … With e-currency based on cryptographic proof, without the need to trust a third party middleman, money can be secure and transactions effortless.”

Satoshi Nakamoto 2009
Satoshi Nakamoto is the name used by the unknown person or people who designed bitcoin (2008) and created its original reference implementation (2009). As part of the implementation, they also devised the first blockchain database. In the process they were the first to solve the double-spending problem for digital currency.

Nakamoto claimed to be a man living in Japan, born on 5 April 1975. Speculation about the true identity of Nakamoto has mostly focused on a number of cryptography and computer science experts. Thus, it reminds us of a well-known question in mathematics (but in this case, the mathematicians are well known):

Who is Niklas Bourbaki?
Bitcoin as Decentral Currency

The bitcoin-network is founded on a decentral database which is administrated by all users with a bitcoin software which registers all transactions of currency.

The network is peer-to-peer and transactions take place between users directly, without an intermediary (e.g. central bank). The proof-of-work algorithm is computationally expensive and practically secure.

A wallet stores the information necessary to transact bitcoins. The exchange rate of bitcoin and other currencies depend on supply and demand (speculative bubbles !!!)
Blockchain is an expandable list of records (blocks) which are linked with cryptographic codes.

Each block contains a cryptographic hash pointer to a previous block, a timestamp and transaction data. New blocks are generated by a consensus procedure (e.g. proof-of-work algorithm).

A blockchain is a decentralized and distributed digital ledger that is used to record transactions of digital values across the nodes (computers) resp. users of a network (Internet of Values). The record cannot be altered retroactively without the alteration of all subsequent blocks and the collusion of the network.
Security of Hash Functions

It is easy to compute hash value $y = h(x)$ from $x$ but it's very hard to find $x$ given only $y$.

A full hash inversion has a known computationally infeasible brute-force running time, being $O(2^k)$ where $k$ is the hash size $k=256$ in SHA256.

If a pre-image was found anyone could very efficiently verify it by computing one hash.

There is a asymmetry in full pre-image mining (computationally infeasible) vs verification (a single hash invocation).

Each block contains the hash of the preceding block, thus each block has a chain of blocks that together contain a large amount of work. Changing a block requires regenerating all successors and redoing the work they contain. This protects the block chain from tampering.
Mining and Proof-of-Work

Generating a new valid block (mining) means solving a cryptographic problem: The proof-of-work requires miners to find a number called a nonce, such that when the block content is hashed along with the nonce, the result is numerically smaller than the network's difficulty target. This proof is easy for any node in the network to verify, but extremely time-consuming to generate, as for a secure cryptographic hash, miners must try many different nonce values (e.g. 0, 1, 2,…).

Proof-of-Work : (threshold inversely proportional to mining-difficulty)
1. initialize block, compute root-hash from transactions
2. compute hash: $h = \text{SHA256(SHA256(block header))}$
3. if $h \geq$ threshold, change nonce (block header) and return to step 2
4. otherwise ($h <$ threshold): valid block found, stop computing and publish block.
Nonce in Blockchain

The nonce is a 32-bit field whose value is set so that the hash of the block will contain a run of leading zeros*:

- Since it is believed infeasible to predict which combination of bits will result in the right hash, many different nonce values are tried: The hash is recomputed for each value until a hash containing the required number of zero bits is found.

- The number of zero bits required is set by the mining difficulty.

- The resulting hash has to be a value less than the current mining difficulty.

- As this iterative calculation requires time and resources, the presentation of the block with the correct nonce value constitutes proof of work.

* The rest of the fields (with defined meaning) may not be changed.
Mining difficulty is a measure of how difficult it is to find a hash below a given target. The Bitcoin network has a global block difficulty:

\[
\text{difficulty} = \frac{\text{difficulty}_\text{1 target}}{\text{current target}}
\]

(target is a 256 bit number.)

The difficulty is adjusted every 2016 blocks based on the time it took to find them.

At the desired rate of one block each 10 minutes, 2016 blocks would take exactly two weeks to find.

\[
\text{time (2016 blocks)} > 2 \text{ weeks} \Rightarrow \text{difficulty reduced}
\]

\[
\text{time (2016 blocks)} < 2 \text{ weeks} \Rightarrow \text{difficulty increased}
\]
Merkle Tree and Secure Verification of Large Data Structures

Hash trees can be used to verify any kind of data stored, handled and transferred in and between computers.

They can help ensure that data blocks received from other peers in a peer-to-peer network are received undamaged and unaltered, and even to check that the other peers do not lie and send fake blocks (trusted computing).

In some proof-of-work systems, users provide hash collisions as proof that they have performed a certain amount of computation to find them.
Trust & Security by Computer Power Only?

With growing blockchain (one block per ten minutes), the difficulty of the proof-of-work increases in proportion to increasing computer power:

Pros:
• *Increasing difficulty of proof-of-work* diminish the probability of tampering and improves *security & trust* in blockchain technology.
• *Security & trust* is completely based on *constructive algorithms* of proof-of-work with *finitely growing binary hash trees* („collision resistant“).

Cons:
• *Exponentially increasing computer power* means *growing power consumption* and (with that) *economic* and *environmental costs*: e.g., Bitcoin network Nov 2017 30,14 TWh p.a. > power consumption of Ireland.
• The *initially democratic idea* of a decentralized cryptocurrency without central control (and failures) of banks and *equal chances of clients* cannot be realized: Influence of members depends on computer power and energy power (*monopolies* of states/concerns).
9. Bridging Logic, Mathematics, Computer Science, and Philosophy
Proof Theory:
- intuitionistic/minimal logic
- functional interpretation
- information system

Mathematics:
- numerical analysis
- functional analysis
- financial mathematics

CORE:
„Trust & Security by Constructive Methods in Mathematics and Operations Research!“

Computer Science:
- functional programing
- scientific computing
- program extraction

Philosophy:
- intuitionism
- constructivism
- ethical evaluation
The Universe as Automaton
From Simplicity and Symmetry to Complexity

Klaus Mainzer

Die Berechnung der Welt
Von der Weltformel zu Big Data

C.H. Beck
Künstliche Intelligenz – Wann übernehmen die Maschinen?

