Proof theory and smart contracts

formal logic meets finance

Marcello Paris
marcello.paris@gmail.com

May 3, 2019
Quantitative Finance @ work
School of Economics
University of Roma, Tor Vergata

Disclaimer The author is currently employed at UniCredit, R&D dept. The content of this article is solely the responsibility of the author. The views expressed here are those of the author and strictly personal, and do not necessarily reflect the views of the UniCredit Group.

Agenda

- ❑ programmable agreements
- ❑ the need to get to formal verification
- ❑ constructivism in logic and type theory

Smartness

> "Allows self-executing computer code to take actions at specified times and/or based on reference to the occurrence or non-occurrence of an action or event (e.g., delivery of an asset, weather conditions, or change in a reference rate)"
> ["A primer on smart contracts" released by
> the U.S. Commodity Futures Trading Commission on 27 November 2018]

- ❑ concept of a smart contract is not new:
  the idea is to use some formal language to write the terms of an agreement
  and let the contract the formal language be processed fairly
- ❑ humans write software: they tell the machine what to do:
  agreements are starting to become software themselves,
  a crucial step in a real digitalization of finance

Agreements

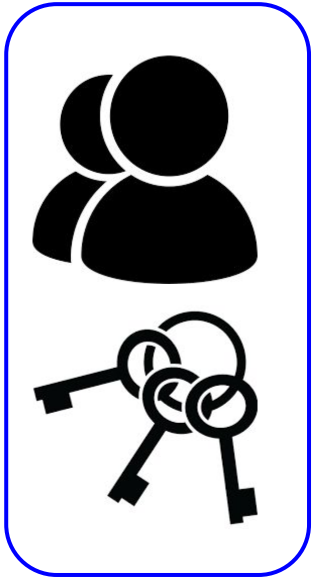
here, an ‘agreement’ is whatever: 2 (or more) actors agree to do some actions under a given set of rules [trade or service agreements, derivatives or claims contingent on other types of events, ...]

Actors A and B agree on:
B is selling a book to A for 20€

- payment in advance:
    - A pays 20€ to B
    - B verifies the payment and sends the book to A
- cash on delivery:
    - B sends the book to A
    - A verifies the shipment and pays 20€ to B
- mixture:
    - A pays 10€ to B
    - B verifies the payment and sends the book to A
    - A verifies the shipment and pays the remaining 10€ to B

Ethereum contract accounts

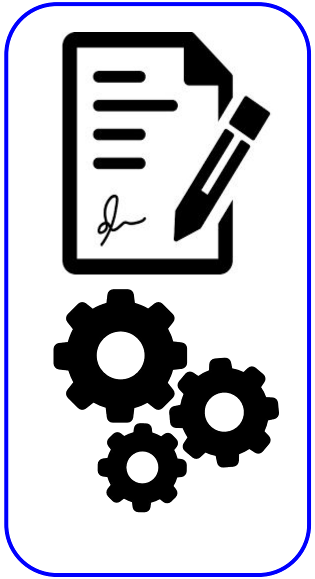
EXTERNALLY OWNED
ACCOUNT



Roughly, 2 types of accounts: externally owned and contracts.

- externally owned accounts have free will (which is represented by the absence of code and the presence of the keys) and they originate contracts and triggers
- contracts accounts have no free will, they are ruled by a code

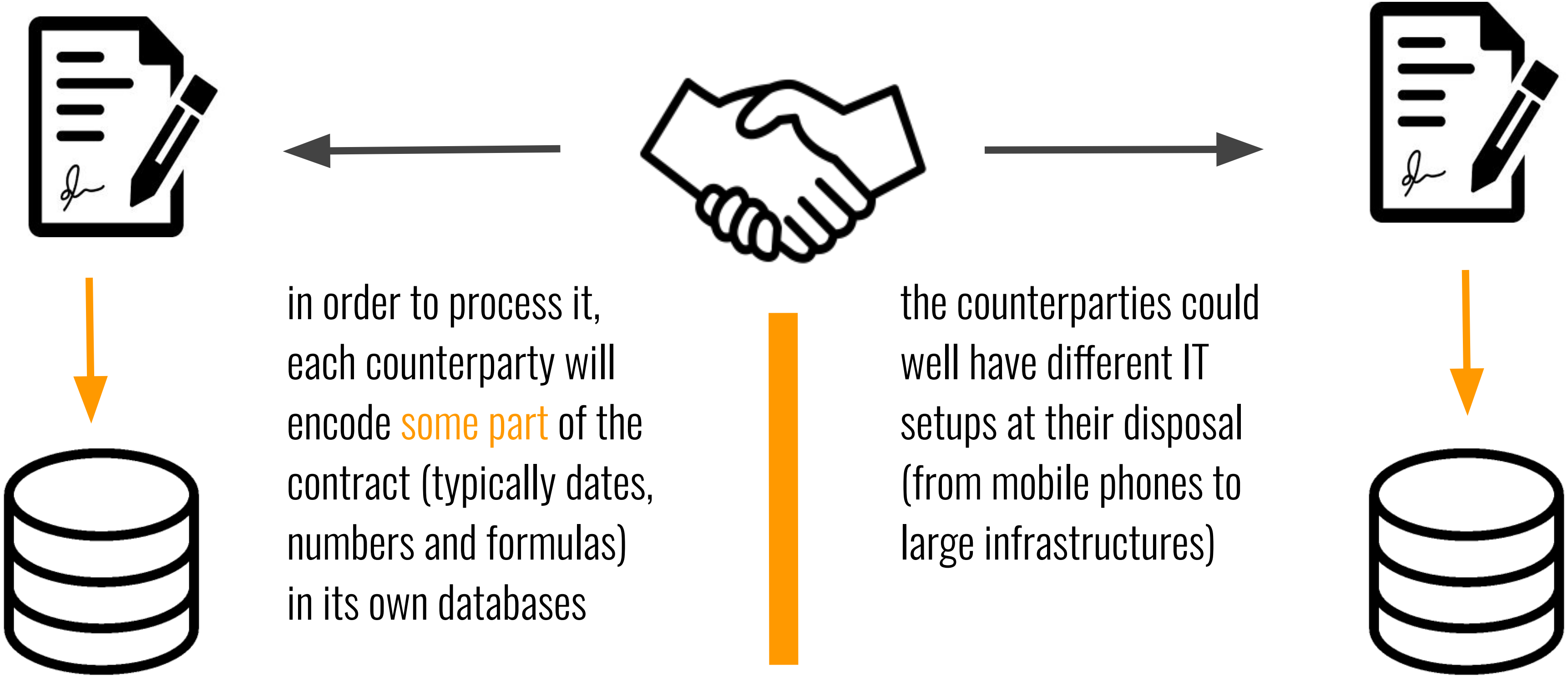
CONTRACT
ACCOUNT



Formalization and processing

legal language is commonly used to draft agreements, however
the actual processing is typically retained by the (say, 2) counterparties:
so, an agreement produces (at least) 2 signed copies of a contract,
to be processed separately and independently by the counterparties

Encoding agreements



in order to process it, each counterparty will encode some part of the contract (typically dates, numbers and formulas) in its own databases

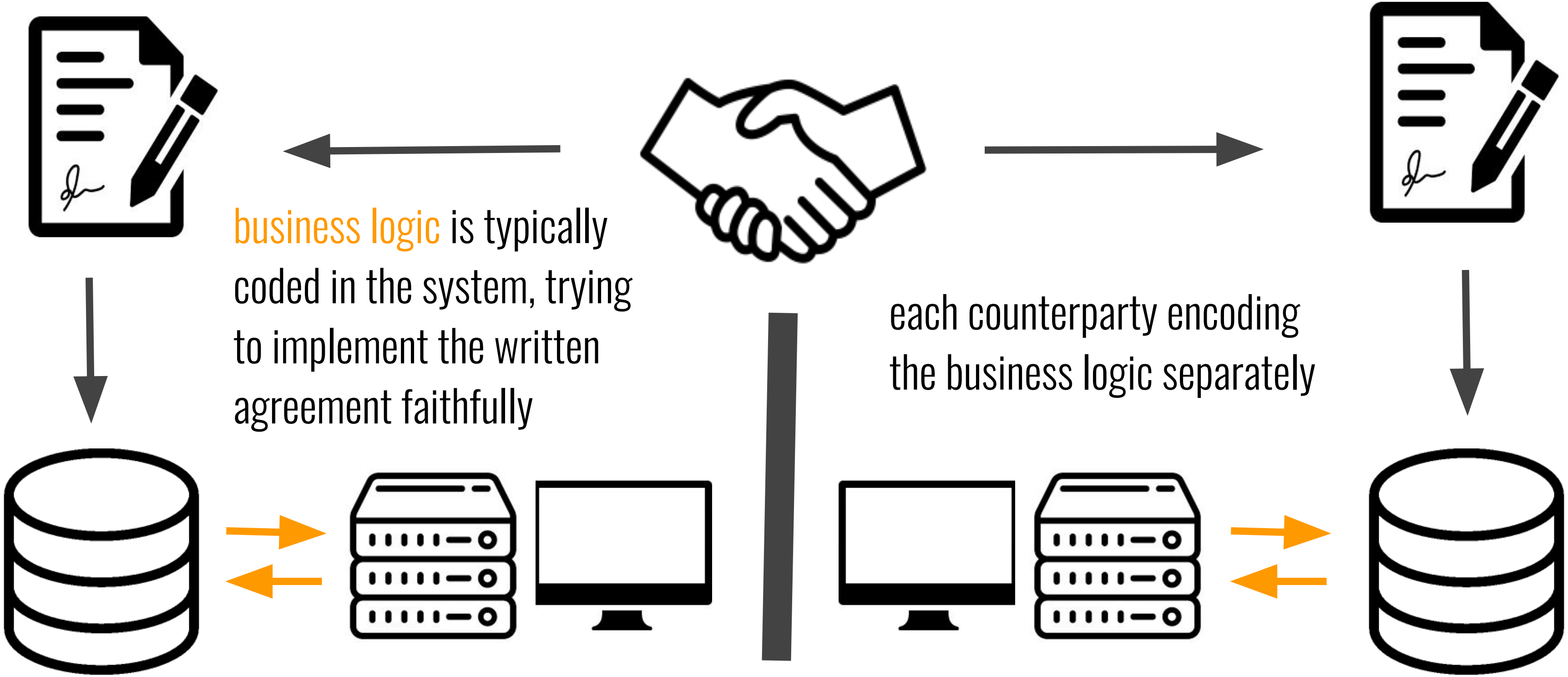
the counterparties could well have different IT setups at their disposal (from mobile phones to large infrastructures)

Encoding agreements

- 2 banks on a derivative contract
- a buyer from some large retailer
- a buyer of a service from a small business

- 2 individuals agreeing on something
- 2 large companies
- an entire community

Implementation of a written agreement



business logic is typically coded in the system, trying to implement the written agreement faithfully

each counterparty encoding the business logic separately

Why 2 different processing of a some contract ?

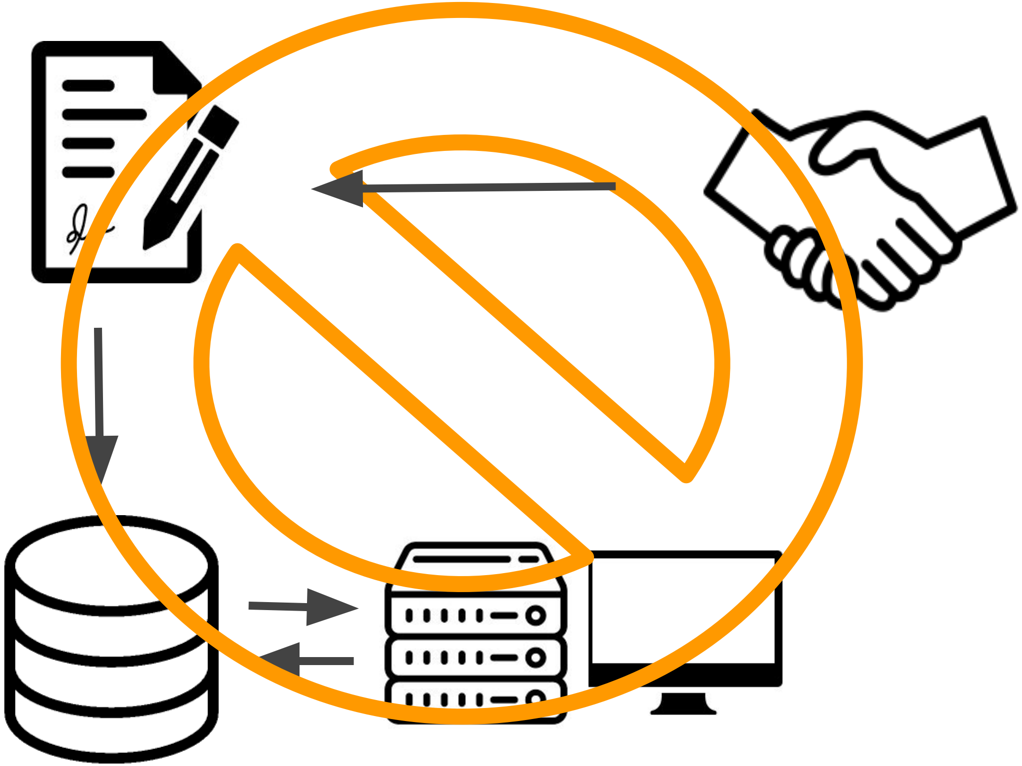
fulfillment of contract obligations are completely up to the actors

there may possibly be different interpretations of the contract

each actor could misbehave (at will or unwillingly while doing so)

the contract will typically take into account the discretionary nature of each actor's actions while processing it (by penalties, clauses or fines)

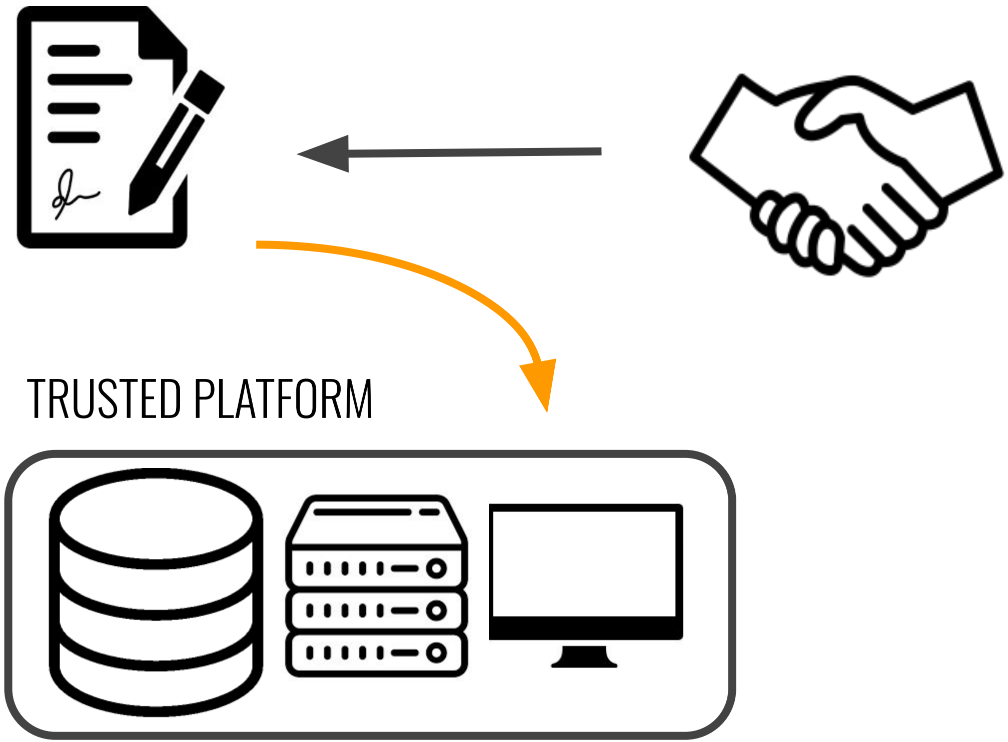
Trusting a single processing of the contract



it could end up with 1 actor trusting the processing of the other
(say: a buyer from a huge retailer will use its systems and mobile apps)

1 of the 2 actors is likely to be sure enough that the business logic implemented in the IT systems of the other is exactly implementing the written agreement

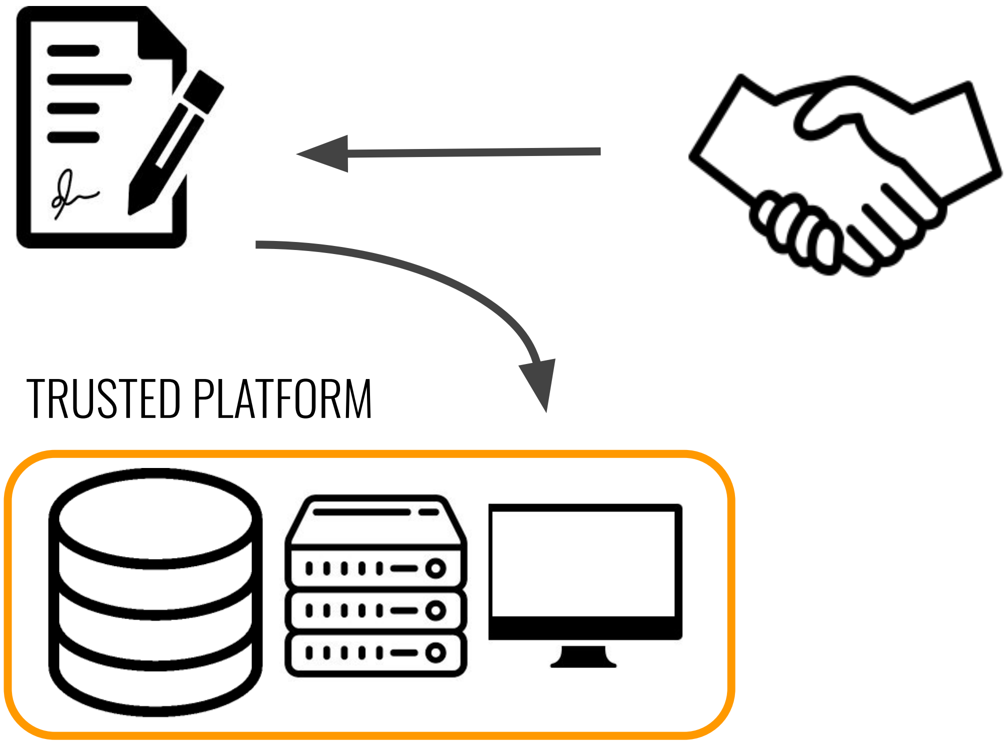
Encoding on a smart contract platform



an attractive idea is to use some platform (external to the actors in the contract) that each actor trusts in order to process their agreement fairly

clearly enough, the actors should tell the platform the agreement that have reached, so the encoding could not just be number and formulas, but the entire contract (not in legal language, but in some programming language)

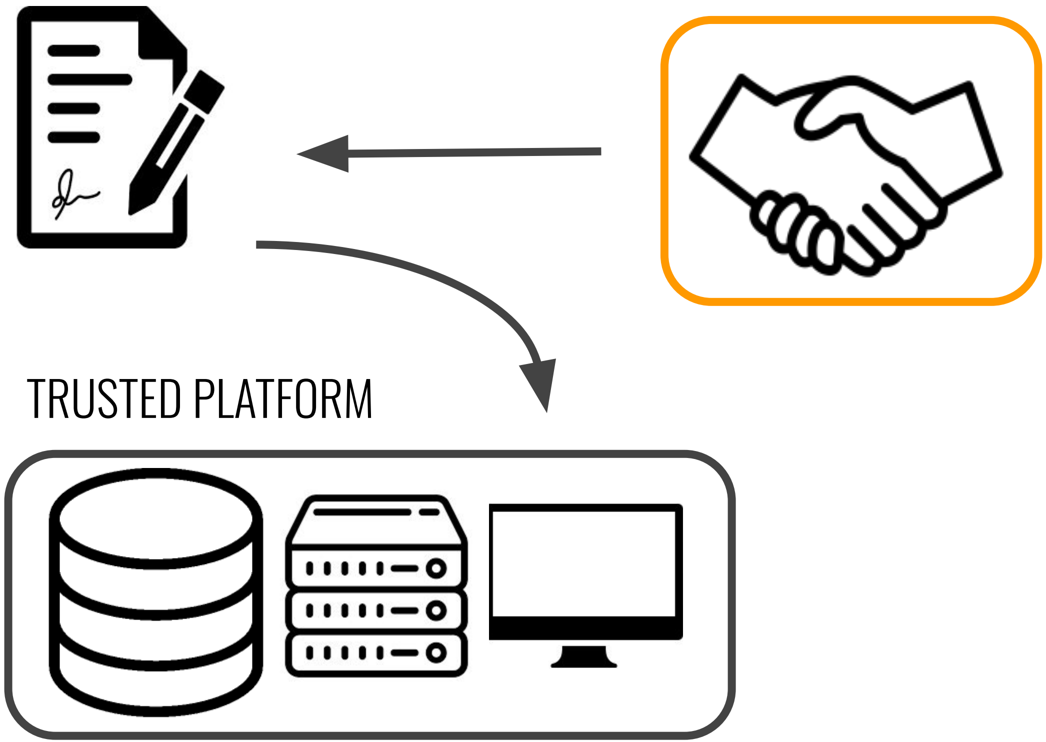
A trusted platform



the last 10y has shown very remarkable advances in the design of platforms that could offer enough robustness and reliability to be trusted (many of them are community-based)

here, I will not discuss any of those, I'd rather go further in analyzing the many level of trust that the actors need to pass through in order to outsource their agreement to some external platform

Programmability: the benefits



TRUSTED PLATFORM

Actors A and B agree on:
B is selling a book to A for 20€

- cash on delivery
    - the platform verifies that A has 20€ available to pay B
    - the platform could reserve the amount for the purchase, so B can stay safe (if the book is shipped within the terms and the shipment is ok)
    - B sends the book to A
    - A verifies the shipment and confirms that the shipment is ok
    - the platform pays 20€ from A to B

Formalize an agreements

pseudocode for the agreement of the right (think of your favourite programming language):

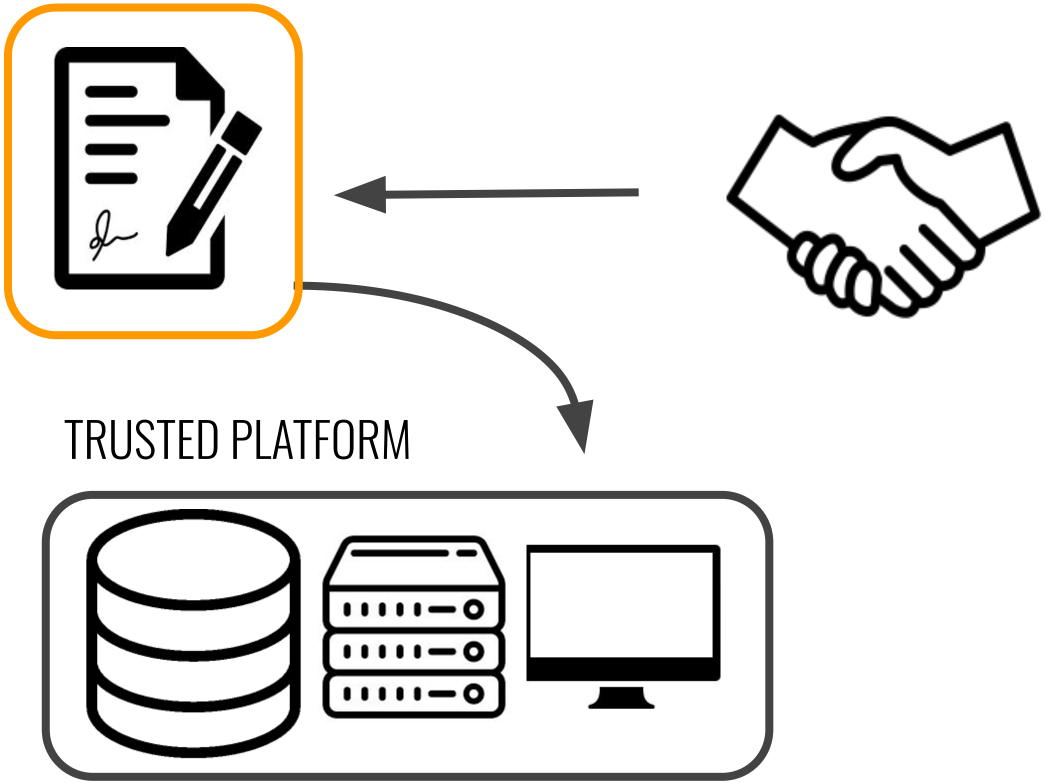
- contract starts today
- verify(A, 20€)
- reserve 20€ from A for this agreement
- wait for B to confirm shipment (within, say, 1 week)
- wait for A to confirm shipment is ok
- pay 20€ from A to B
- contract terminates

encoding intentions into pseudocode

Actors A and B agree on:
B is selling a book to A for 20€

- cash on delivery
    - the platform verifies that A has 20€ available to pay B
    - the platform could reserve the amount for the purchase, so B can stay safe (if the book is shipped within the terms and the shipment is ok)
    - B sends the book to A
    - A verifies the shipment and confirms that the shipment is ok
    - the platform pays 20€ from A to B

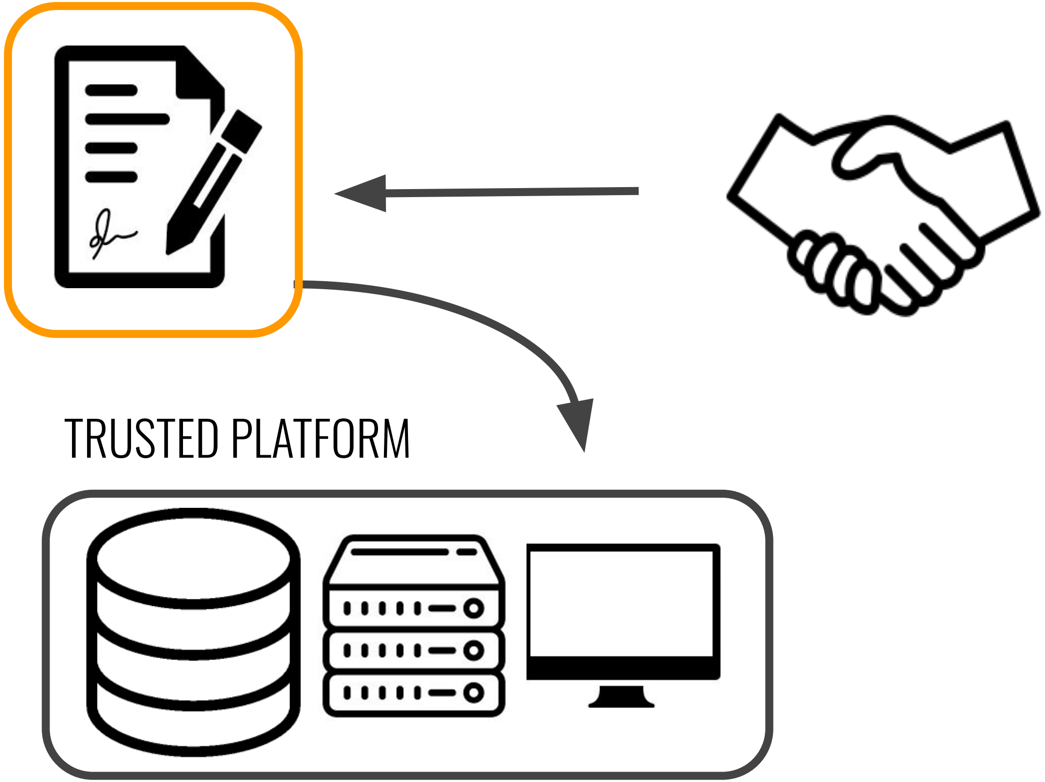
So many points to trust



trust your code is exactly
encoding actors’ intentions

- ☐ code should be readable by each actor
- ☐ is code more or less readable than a legal agreement ?
- ☐ software can help the actors in reading their code, it could help providing some verification of it

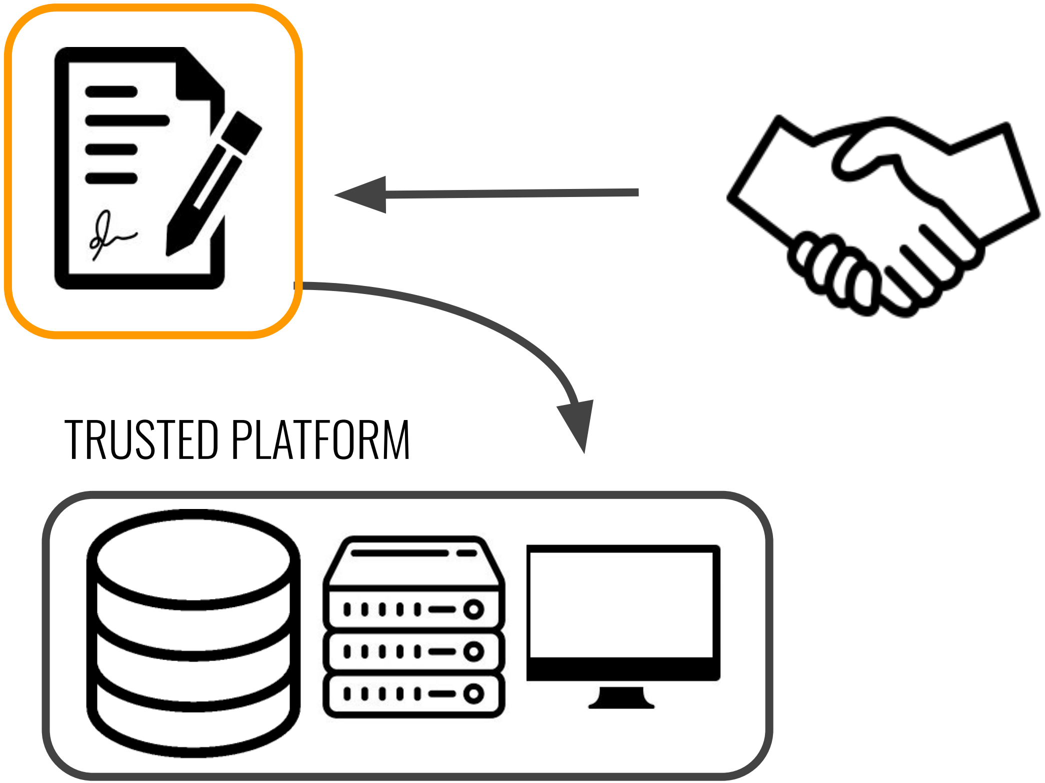
So many points to trust



trust actors' intentions are consistent and no corner case is left behind during the life of the contract

- ❑ in the example above, if B ships the book 3y later, A is bound to pay for it ? (if B is a bookstore, the contract should have expired, but, maybe, B is a famous writer)

So many points to trust



trust your code
will not be misunderstood
or misinterpreted

- the trusted platform should be aligned with your interpretation of the code or there should be one 1 interpretation

Agreements in code

Bitcoin messages are written in terms of its scripting language.

Something similar happens in ticket restaurants, they are:

- ❑ strictly personal
- ❑ spend only the entire face value
- ❑ locking script: requesting a signature
- ❑ unlocking script: to sign
- ❑ spending it only once

Bitcoin scripting language is a stack-based bytecode with a limited list of basic primitives.

the basic ‘pay-to-pubkey-hash’ is:

```
OP_DUP
OP_HASH160
<pubhash>
OP_EQUALVERIFY
OP_CHECKSIG
```

(also Bitcoins come forged with the name of the owner; spending them means melting them down to do other coins)

Anatomy of a Bitcoin transaction

- the output scripts are the locking scripts (the scriptPubKey); they code what it must be verified to unlock the coins and spend them
- the stack-based language is asking to: DUPlicate the top of the stack, calculate an HASH160, PUSHDATA (20 bytes literal) on the stack, check that the top 2 elements are EQUAL and VERIFY this (i.e., exit if false), then, at last, CHECKSIGnature

- the input scripts are the unlocking scripts (the scriptSig); their code is meant to satisfy the locking scripts
- the stack-based language is now just PUSHing on top of the stack a signature (71 bytes) and the public key of the owner (both are needed by CHECKSIG and that's why the locking script starts with a DUP)

Output Scripts 756607ca95f708ddebc79efe9467b13972f8dcd033cfbe0de620f73a3d426d58

```
HASH160 PUSHDATA(20)[9bb7c24f2617321953a48d68c755abd13effde1a] EQUAL
DUP HASH160 PUSHDATA(20)[56eaf88e75a7e0086844fdef5da1cab7d9ea99f6] EQUALVERIFY CHECKSIG
```

Input Scripts 8942a7e2e7f1d83f40c6722a40373e31bb9c21b603966468581e3e234fd90663

```
ScriptSig: PUSHDATA(71)
[304402205c87cb59846bef7bebdb5759bbb6091744b121cbedbaed66a0b5a446fd58535602201188f8b1a23ecb8129a13a00a5348fbe3bafa8684173ad89b089aea645d11cfd01]
PUSHDATA(33)[0338433cda8f204c09520493a84566caa916d5881f1c714a6ed127e945963623cc]
```

The need for a proof

proofs are peculiar to mathematical reasoning,
only mathematicians care about proofs:
why should anyone care about them ?
what is it that triggers the need for a proof ?

- humans could identify best practices, design testing pipelines and install verification procedures or complete programming risk management processes, but if the software is really critical, we may not feel that risks are completely wiped out
- if agreements are starting to become software themselves, then this is surely to be accounted as critical (on the same floor as medical applications or transportation and automotive industry)

Intentions

humans write software:
they write some high-level code, design
the algorithms and tell the machine what to do.
There are many non-trivial gaps:

- between what the programmers had in mind and the actual codebase
- between a formal set of specifications for the task (if any) and what the programmers had in mind

This is an horrible code fragment (say, C language) summing up integers up to 99 (included).
Gauss formula tells us the answer is 4950.
We write the code and see that
the answer is 86 ! what’s wrong ?
In which sense, the answer 86 is wrong ?

```
unsigned char k, s;

s = 0

for(k = 0; k < 100; ++k)

    { s += k; }
```

Speak to the machine

There are many other gaps:

- ❏ between the high-level codebase and the actual target output from a compiler (intermediate representation or similar)
- ❏ between the output from a compiler and the actual physical machine instructions
- ❏ between the actual machine instructions and the reliability of their execution

Let's ask our compiler for
an intermediate representation of our code
(clang -Os -S -emit-llvm).
where did the for-loop go ?
is that the same program we wrote, or, rather,
a program that returns an equivalent value ?
(moreover, a value we didn't mean to)

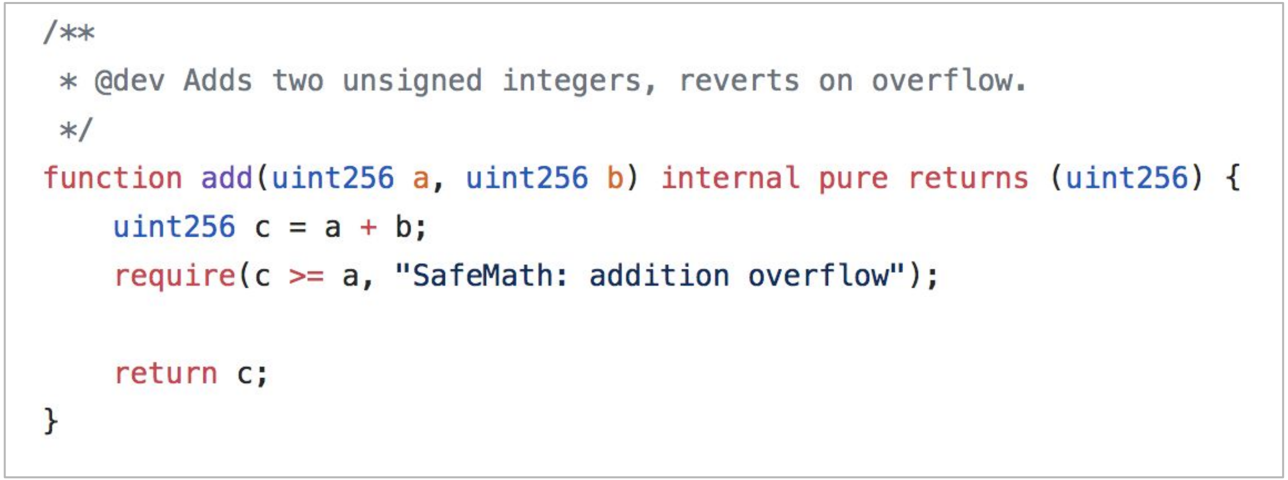
```
define i32 @main()
local_unnamed_addr #0 {

  ret i32 86

}
```

Specifications

Solidity (an high-level language for Ethereum) allow data types
such as uint8 and this kind of things could (and will) happen
[on the right an actual example from the national vulnerability database]

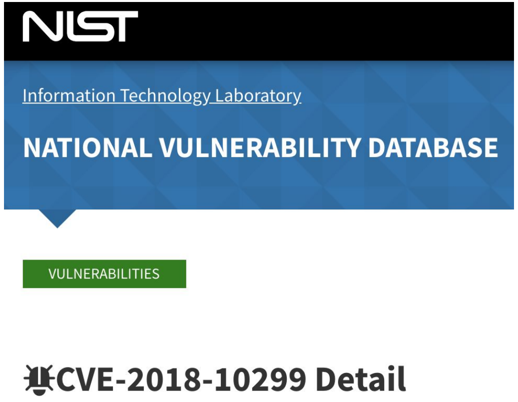


```
/**
 * @dev Adds two unsigned integers, reverts on overflow.
 */
function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    require(c >= a, "SafeMath: addition overflow");

    return c;
}
```

In the example before,
the programmer (maybe) may had in mind to verify Gauss formula.

Was that true ?
How to write down exactly
the specifications ?



NIST

Information Technology Laboratory

NATIONAL VULNERABILITY DATABASE

VULNERABILITIES

CVE-2018-10299 Detail

Solidity: vulnerabilities

Please remark that smart contracts are meant to be immutable.

Reentrant behaviors

this high-level language also allow for reentrant behaviors in the flow:
a contract may call external functions that could re-enter the contract

This could result in a vulnerability if, using this feature, the transactionality if not properly handled. This line of reasoning was used in the DAO attack.

Delegate calls

“a contract can dynamically load code from a different address at runtime. Storage, current address and balance still refer to the calling contract, only the code is taken from the called address.” [from std docs]

This fundamental feature could expose a vulnerability if not properly used.
Parity Multisig Wallet has been hacked explotying a delegate call in the fallback function.

Community and verification

- ❑ you wrote a wonderful contract
  doing amazing things,
  maybe it's not just you:
  a very serious and skilled team,
  everybody checked it (at least) twice
- ❑ how could you ever be sure that
  you wrote what you meant ?,
  how could you ever be sure that
  what you meant is consistent ?
- ❑ your code is very clean, your coding style is flawless,
  your design, develop and test processes are professional,
  a huge amount of tests where passed successfully,
  many skilled people verified the codebase,
  code has been in use for a lot of time,
  code has been challenged by a crowded community
- ❑ ... and so what ?
  you are more or less in the same situation of
  mathematics

Mathematics

what is a theorem in mathematics ?
most people could think of it as
a sort of apodictic truth

they are likely to be disappointed:
mathematics is (still) totally driven
by humans for humans and there have been
examples of (even, famous) theorems
that were later proved to be false as stated or
that were requiring adjustments in their proofs

we could rather say that a theorem is a
statement that some non-trivial community
of mathematicians finds valid and useful
(and those humans need proofs
to convince themselves
about the validity of their claims)

proofs need to convince:
a prover must convince a verifier

V.Voevodsky (1966 - 2017)

The IAS Institute Letter, Summer 2014, hosts an article by V.Voevodsky (subtitle: “Professor Voevodskys Personal Mission to Develop Computer Proof Verification to Avoid Mathematical Mistakes”)

“In 1999 - 2000, again at the IAS, I was giving a series of lectures, and Pierre Deligne was taking notes and checking every step of my arguments. Only then did I discover that the proof of a key lemma in my paper contained a mistake and that the lemma, as stated, could not be salvaged.”

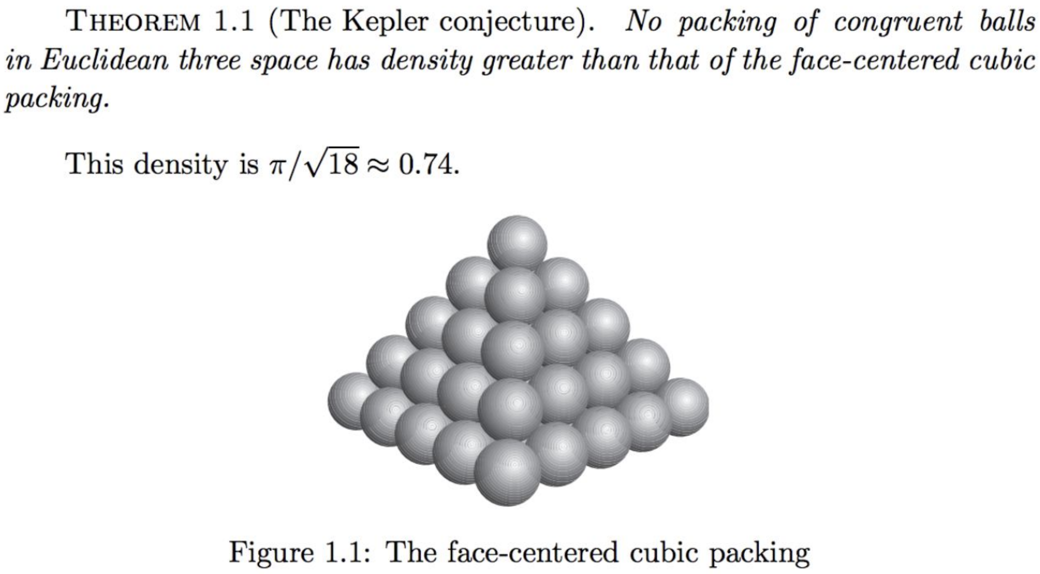
“This story got me scared. Starting from 1993, multiple groups of mathematicians studied my paper at seminars and used it in their work and none of them noticed the mistake.”
“Mathematical research currently relies on a complex system of mutual trust based on reputations.”

“When I first started to explore the possibility, computer proof verification was almost a forbidden subject among mathematicians.” “the foundations of mathematics were unprepared for the requirements of the task”

An example

“Despite the unusual nature of the proof, the editors of the Annals of Mathematics agreed to publish it, provided it was accepted by a panel of twelve referees.

In 2003, after four years of work, the hea of the referee’s panel, G ́abor Fejes T ́oth, reported that the panel were “99% certain” of the correctness of the proof, but they could not certify the correctness of all of the computer calculations.”



THEOREM 1.1 (The Kepler conjecture). *No packing of congruent balls in Euclidean three space has density greater than that of the face-centered cubic packing.*

This density is $\pi/\sqrt{18} \approx 0.74$.

Figure 1.1: The face-centered cubic packing

Then, ”the Kepler conjecture was accepted as a theorem.
In 2014, the Flyspeck project team, headed by Hales, announced the completion of a formal proof of the Kepler conjecture using a combination of the Isabelle and HOL Light proof assistants.”

(source: Wikipedia)

Back to contracts: testing vs proving

pseudocode for the agreement of the right (think of your favourite programming language):

```
-   contract starts today
-   verify(A, 20€)
-   reserve 20€ from A for this agreement
-   wait for B to confirm shipment
    (within, say, 1 week)
-   wait for A to confirm shipment is ok
-   pay 20€ from A to B
-   contract terminates
```

having formalized our intentions,
we could test out implementation.
There could be tests and simulations running,
but none of these could prove anything
(even if it is of the utmost importance).

our formalization should allow us
to ask questions to it, proofs for statements
or claims that certain statements are false.

Claims on contracts

pseudocode for the agreement of the right (think of your favourite programming language):

```
- contract starts today
- verify(A, 20€)
- reserve 20€ from A for this agreement
- wait for B to confirm shipment
  (within, say, 1 week)
- wait for A to confirm shipment is ok
- pay 20€ from A to B
- contract terminates
```

possible claims:

- ☐ there is no possibility for this contract to allow a payment without a signature of the buyer
- ☐ the contract cannot last more than 1y
- ☐ there is no possibility the amount reserved is not available at payment time
- ☐ the case when A disappear is taken into account

Claims as verification rules

can we prove (any of) these claims ?

we are not merely asking that the code is correct (maybe this does not make any sense at all),
we are asking about a proof for these claims, given the contract

the collection of those claims be the our set of verification rules that we deem important to be answered

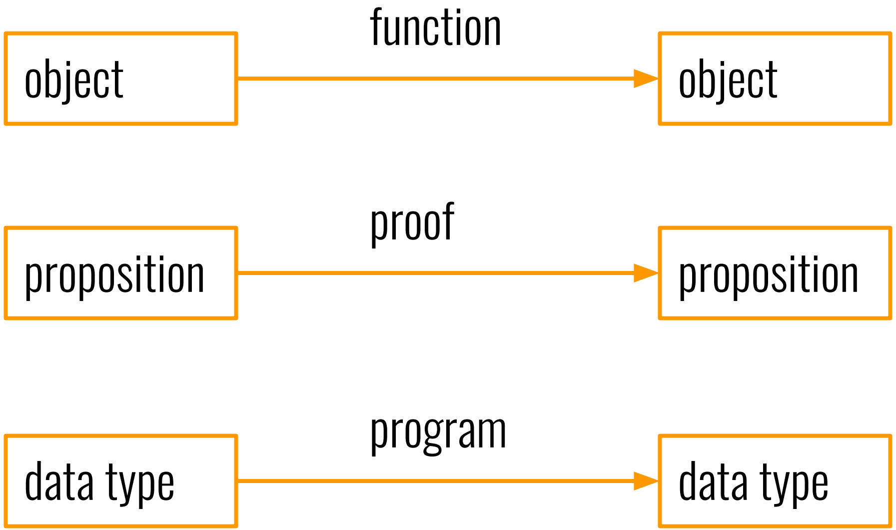
suppose we are given a swap contract:
a claim could be a risk assessment:
- “a given leg will never make a payment exceeding a given amount”

or claims could be contingent:
- “if a leg defaults, the contract will still allow some recovery”

please remark that we are not encoding uncertainty in our context.

Proof as functions

next part is just to give a clue about how we could approach the problem of giving a proof for a statement



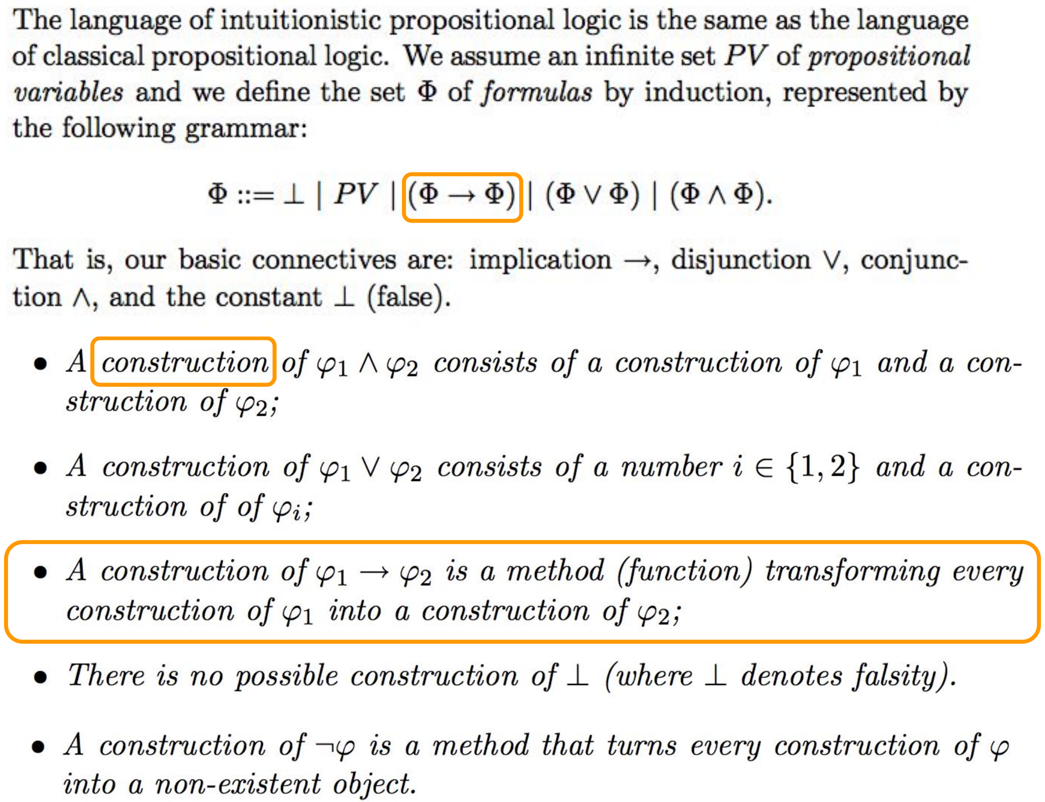
a function (morphism is category theory) is an arrow from an object to another.

a program is something of arrow type, transforming a data type into another. Similarly, a proof is something of arrow type, showing that a proposition follows from another.

Classical vs constructive logic

classical logic is based on the notion of truth	constructive approach to logic is based on the notion of proof
the truth of a statement is independent of any observer actually ‘understanding’ the statement being true of false	an observer can prove a statement true or prove that the statement imply a contradiction (i.e., it is false)
a priori, any statement is either true or false *tertium non datur*	a priori, any statement is neither true nor false (unless an observer can prove any of them)

Informal semantics for intuitionistic logic



The language of intuitionistic propositional logic is the same as the language of classical propositional logic. We assume an infinite set PV of *propositional variables* and we define the set Φ of *formulas* by induction, represented by the following grammar:

$$\Phi ::= \bot \mid PV \mid \boxed{(\Phi \to \Phi)} \mid (\Phi \vee \Phi) \mid (\Phi \wedge \Phi).$$

That is, our basic connectives are: implication $\to$, disjunction $\vee$, conjunction $\wedge$, and the constant $\bot$ (false).

- *A construction of $\varphi_1 \wedge \varphi_2$ consists of a construction of φ_1 and a construction of φ_2;*
- *A construction of $\varphi_1 \vee \varphi_2$ consists of a number $i \in \{1, 2\}$ and a construction of of φ_i;*
- *A construction of $\varphi_1 \to \varphi_2$ is a method (function) transforming every construction of φ_1 into a construction of φ_2;*
- *There is no possible construction of $\bot$ (where $\bot$ denotes falsity).*
- *A construction of $\neg\varphi$ is a method that turns every construction of φ into a non-existent object.*

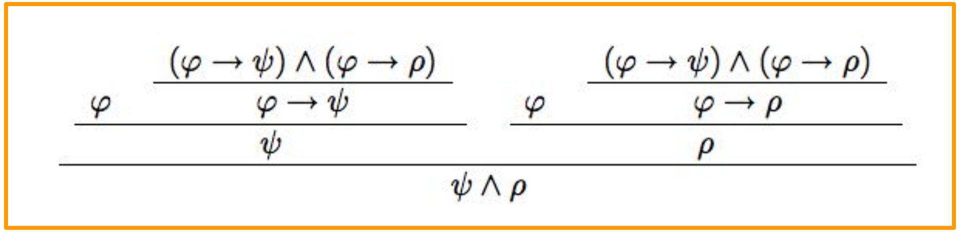
propositional calculus includes variables and formulas (please remark, the implication $\longrightarrow$ being among the basic connectives).

Fundamental keyword is: construction

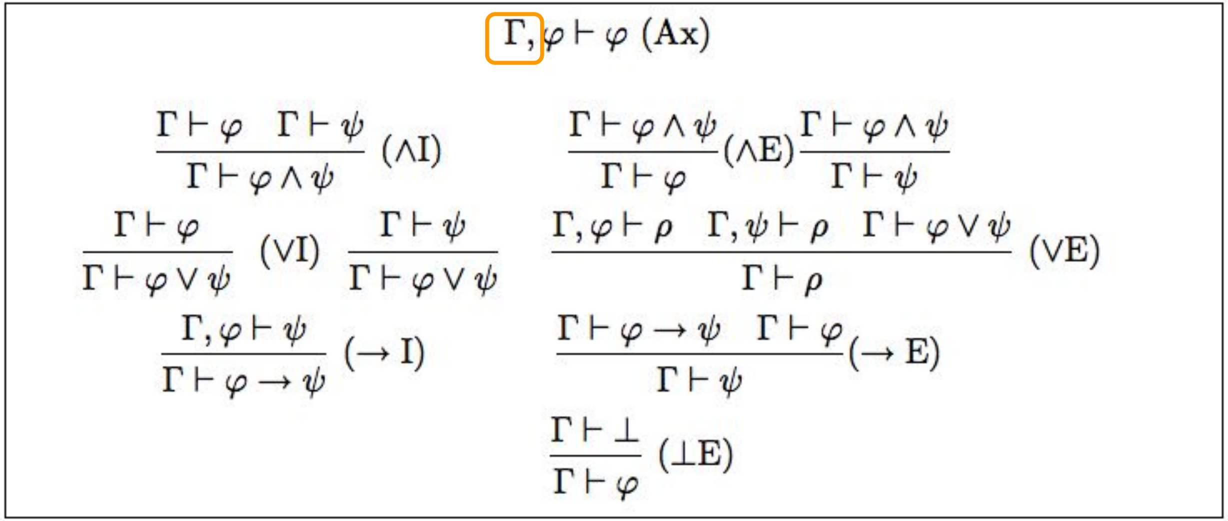
true as inhabitated

Natural deduction

a language for expressing proofs.



$$\dfrac{\dfrac{\varphi \qquad \dfrac{(\varphi\to\psi)\wedge(\varphi\to\rho)}{\varphi\to\psi}}{\psi} \qquad \dfrac{\varphi \qquad \dfrac{(\varphi\to\psi)\wedge(\varphi\to\rho)}{\varphi\to\rho}}{\rho}}{\psi\wedge\rho}$$



$$\boxed{\Gamma,}\varphi \vdash \varphi \ (\mathrm{Ax})$$

$$\dfrac{\Gamma\vdash\varphi \quad \Gamma\vdash\psi}{\Gamma\vdash\varphi\wedge\psi}\,(\wedge\mathrm{I}) \qquad \dfrac{\Gamma\vdash\varphi\wedge\psi}{\Gamma\vdash\varphi}\,(\wedge\mathrm{E})\,\dfrac{\Gamma\vdash\varphi\wedge\psi}{\Gamma\vdash\psi}$$

$$\dfrac{\Gamma\vdash\varphi}{\Gamma\vdash\varphi\vee\psi}\;(\vee\mathrm{I})\;\dfrac{\Gamma\vdash\psi}{\Gamma\vdash\varphi\vee\psi} \qquad \dfrac{\Gamma,\varphi\vdash\rho \quad \Gamma,\psi\vdash\rho \quad \Gamma\vdash\varphi\vee\psi}{\Gamma\vdash\rho}\;(\vee\mathrm{E})$$

$$\dfrac{\Gamma,\varphi\vdash\psi}{\Gamma\vdash\varphi\to\psi}\;(\to\mathrm{I}) \qquad \dfrac{\Gamma\vdash\varphi\to\psi \quad \Gamma\vdash\varphi}{\Gamma\vdash\psi}(\to\mathrm{E})$$

$$\dfrac{\Gamma\vdash\bot}{\Gamma\vdash\varphi}\;(\bot\mathrm{E})$$

Figure 2.1: INTUITIONISTIC PROPOSITIONAL CALCULUS

the proof system consists of an axiom scheme (Ax) and rules (introduction ‘I’ and elimination ‘E’ rules).

Functions: λ-calculus (untyped)

terms	syntax (M and N denote terms)
variable	x
application	$M\ N$
abstraction	$\lambda x.M$

function (mathematics)	abstraction (λ-calculus)
$f(x) := x^2$	$\lambda x.x^2$

evaluation (mathematics)	β-reduction (λ-calculus)
$f(z+1) := (z+1)^2$	$(\lambda x.x^2)\ (z+1) \longrightarrow (z+1)^2$

“λ-calculus is a fundamental topic originating from A.Church in the 1930s, which may be regarded as the calculus underlying the behavior of functions, including variable binding and substitution - essential concepts in mathematics and computer science.”

Types, arrow types and typed λ-calculus

the concept of a type is fundamental in mathematical logic and computer science.

a very basic type theory typically includes arbitrary type variables (denoted α, β, … meaning, say, the primitive types 'int' or 'string') and arrow types (denoted $\boldsymbol{\sigma}, \boldsymbol{\tau}$, … meaning, say, the type of a function from int to int, $\boldsymbol{\sigma} : \alpha \rightarrow \alpha$)

❏ an arrow type is the type of a function, its notation ⟶ (the arrow) reminds that of an implication (between propositions in logic) [this similarity will be made precise in the Curry-Howard isomorphism]

❏ a (λ-calculus) term M could be of type $\boldsymbol{\sigma}$, which is denoted by $M : \boldsymbol{\sigma}$ (so that, for example, $x : \alpha$, then $\lambda x.x^2 : \alpha \rightarrow \alpha$)

The Curry-Howard isomorphism

propositions-as-types: linking logic to computation:
to each proposition a (given) logic there is a corresponding type in a (given) programming language.
It is an amazing correspondence between

- ❏ λ-calculus (a formalism for expressing functions)
- ❏ natural deduction for intuitionistic logic (a formalism for expressing proofs)

In particular, in this formal analogy (isomorphism),
if we take the set of propositional variables equal to the set of type variables,
then the set of propositional formulas and the set of simple types are identical,
i.e., arrows for function types in λ-calculus corresponds to implications in propositional formulas
[as presented, this is true only on the implicational fragment of intuitionistic propositional logic]

Proof assistants in mathematics

Coq	Home About Coq Get Coq Documen The Coq Proof Assistant
Isabelle	Isabelle
Lean	LEAN

```
Theorem sqrt2_not_rational :
 forall p q : nat, q <> 0 -> p * p = 2 * (q * q) -> False.

intros p q;
generalize p; clear p;
elim q using (well_founded_ind lt_wf). clear q;
intros q Hrec p Hneq; generalize (neq_O_lt _ (sym_not_equal Hneq));
intros Hlt_O_q Heq.
apply (Hrec (3 * q - 2 * p) (comparison4 _ _ Hlt_O_q Heq) (3 * p - 4 *
q)).
apply sym_not_equal; apply lt_neq; apply plus_lt_reg_l with (2 * p);
rewrite <- plus_n_O; rewrite <- le_plus_minus; auto with *.
apply new_equality; auto.
Qed.
```

Proof assistants in mathematics

Coq	Home About Coq Get Coq Documen The Coq Proof Assistant
Isabelle	Isabelle
Lean	LEAN

```
Theorem minus_minus : forall a b c : nat, a - b - c = a - (b + c).
intros a; elim a; auto.
intros n' Hrec b; case b; auto.
Qed.

Remark expand_mult2 : forall x : nat, 2 * x = x + x.
intros x; ring.
Qed.

Theorem lt_neq : forall x y : nat, x < y -> x <> y.
unfold not in |- *; intros x y H H1; elim (lt_irrefl x);
 pattern x at 2 in |- *; rewrite H1; auto.
Qed.
```

Conclusions

- agreements in finance and economics (as many other things) are becoming (critical) software
- the many levels of trust implied by an agreement are directly linked to the quality of the software produced
- the way proof assistants currently work in mathematics could be much useful in providing a formal verification framework for agreements drafted in formal languages

References

- Lectures on the Curry-Howard isomorphism [notes]
- “Type theory and formal proof” by Rob Nederpelt and Herman Geuvers [book]
- Programming and proving with dependent types [notes]
- ERC20 contracts: overflow bug example
- Some Solidity vulnerabilities
- Securing smart contracts
- Cardano

Thank you !

Disclaimer The author is currently employed at UniCredit, R&D dept. The content of this article is solely the responsibility of the author. The views expressed here are those of the author and strictly personal, and do not necessarily reflect the views of the UniCredit Group.