Formal and Foundational Study of Programs and Programming Languages

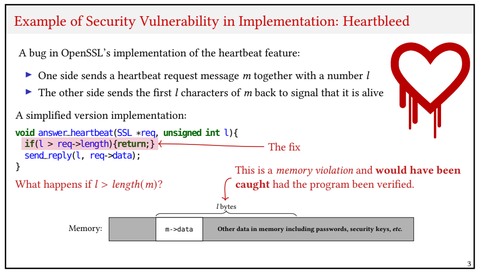
Amin Timany

Dec 6th, 2024,
Aarhus University, Aarhus, Denmark

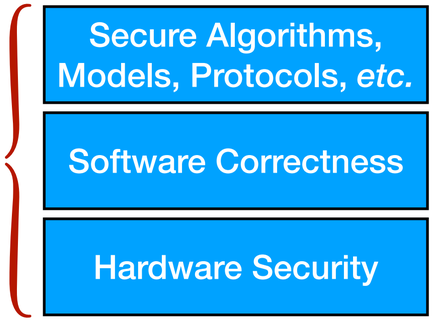
Why Study Programs?

Software correctness is essential for security

- Bugs, compromising security can occur in implementations, even if the high-level models and protocols are correct.
  - Example: the infamous Hearbleed bug
    - A memory safety bug
- The entire stack should be secure



Formal and foundational techniques apply to the entire stack

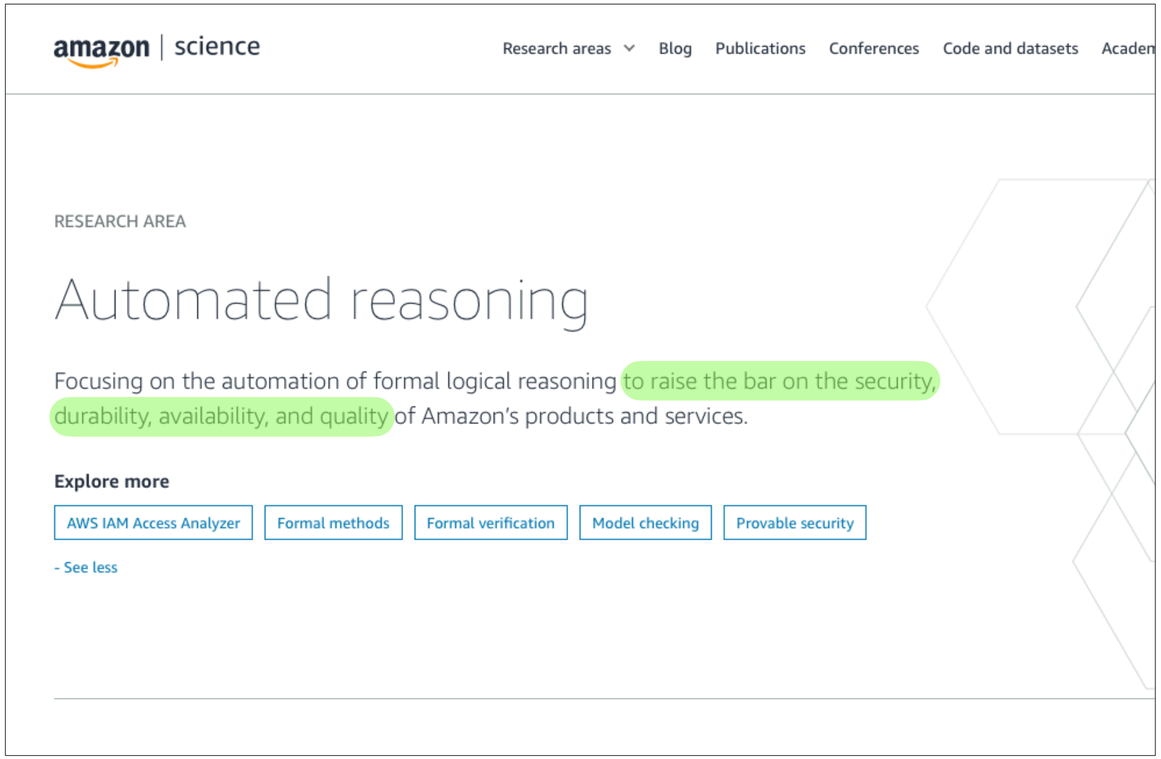


- February 2024: the White House issued a memorandum recommending memory safety and using formal methods

Why Study Programs?

Not just security: robustness of infrastructural software

- Some companies, notably AWS, have embraced formal reasoning
  - The formal guarantees give them a competitive edge in the market

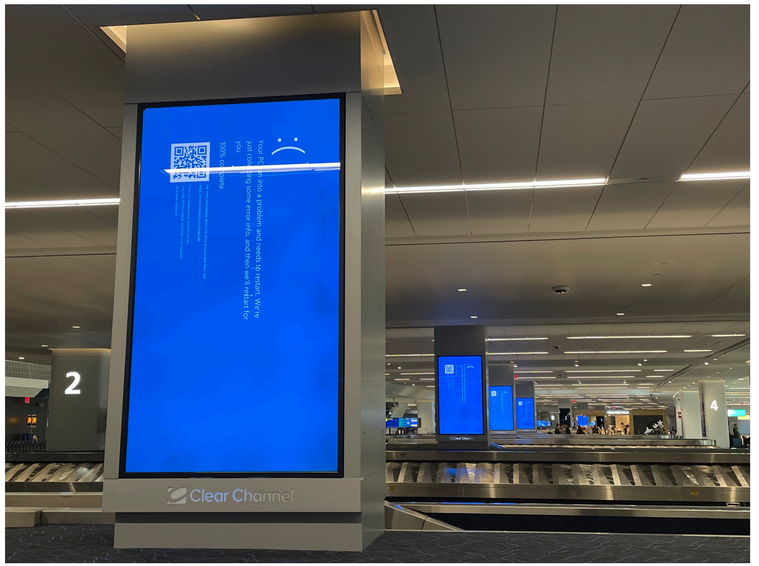


Why Study Programs?

Not just security: robustness of infrastructural software

- Some companies have not
  - Leading to data corruption, service unavailability, *etc.*
  - Can incur to a hefty cost
- Catastrophic example: the CrowdStrike outage
  - Affected the operation of airports, hospitals, *etc.*
  - Another memory safety bug

Blue screens of death at LGA airport from the CrowdStrike 2024 July outage



How Do We Prove a Program Correct?

This program should compute n!

```
// assume n > 0, i = 1, f = 1
while(i <= n){
    f = f * i;
    i = i + 1;
}
// f = n!
```

How Do We Prove a Program Correct?

This program should compute n!

Key Insight:

For each statement C,
if the condition above C holds,
then after running C the condition below it holds

```
// assume n > 0, i = 1, f = 1
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

```
// f = n!
```

How Do We Prove a Program Correct?

This program should compute n !

Key Insight:

For each statement C,
if the condition above C holds,
then after running C the condition below it holds

```
// assume n > 0, i = 1, f = 1
```

Generalized

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

```
// f = n!
```

How Do We Prove a Program Correct?

This program should compute n !

Key Insight:

For each statement C,
if the condition above C holds,
then after running C the condition below it holds

```
// assume n > 0, i = 1, f = 1
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

Loop Invariant

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

```
// f = n!
```

How Do We Prove a Program Correct?

This program should compute n!

```
// assume n > 0, i = 1, f = 1
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

```
// f = n!
```

Same as before the loop + the condition holds

Key Insight:

For each statement C,
if the condition above C holds,
then after running C the condition below it holds

How Do We Prove a Program Correct?

This program should compute n !

Key Insight:

For each statement C,
if the condition above C holds,
then after running C the condition below it holds

```
// assume n > 0, i = 1, f = 1
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

```
// f = n!
```

Same as before the loop + the condition doesn't hold

How Do We Prove a Program Correct?

This program should compute n !

Key Insight:

For each statement C,
if the condition above C holds,
then after running C the condition below it holds

```
// assume n > 0, i = 1, f = 1
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

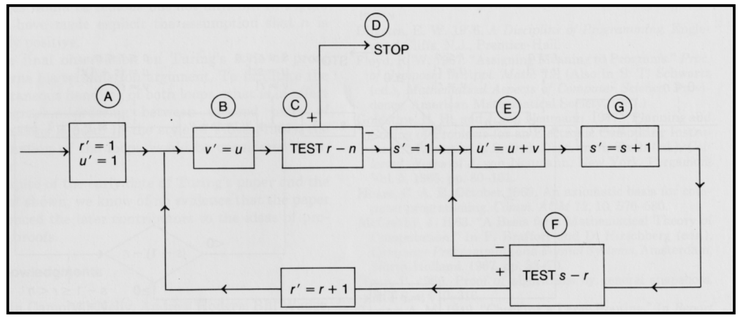
Implies

```
// f = n!
```

Factorial: The First Program Verified

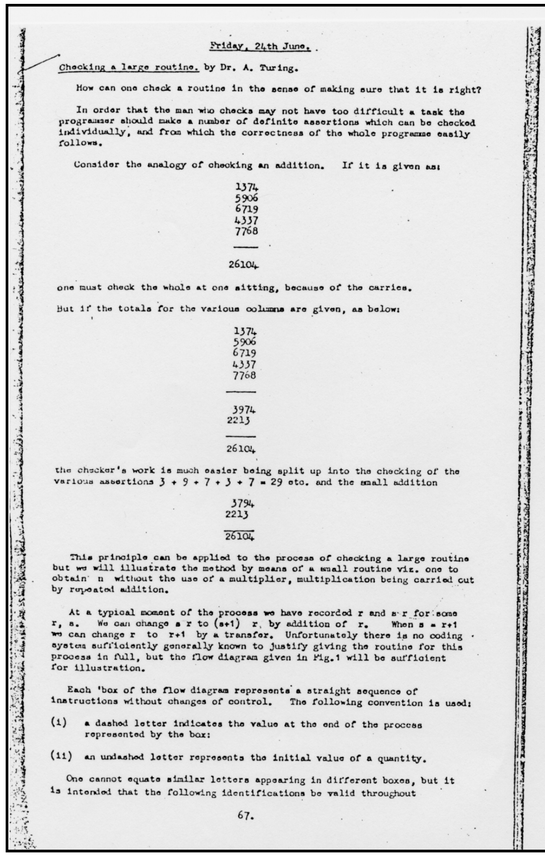
[Turing 1949]: Checking a Large Routine

- Verifies a factorial program with two nested loops
- Uses a “flow diagram” to write the program



Unfortunately there is no coding system sufficiently generally known to justify giving the routine for this process in full, but the flow diagram given in Fig. 1 will be sufficient for illustration.

- Turing’s work goes unnoticed for decades



Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Consider the analogy of checking an addition. If it is given as:

1374
5906
6719
4337
7768
——
26104

one must check the whole at one sitting, because of the carries.

But if the totals for the various columns are given, as below:

1374
5906
6719
4337
7768
——
3974
2213
——
26104

the checker's work is much easier being split up into the checking of the various assertions 3 + 9 + 7 + 3 + 7 = 29 etc. and the small addition

3794
2213
——
26104

This principle can be applied to the process of checking a large routine but we will illustrate the method by means of a small routine viz. one to obtain n without the use of a multiplier, multiplication being carried out by repeated addition.

At a typical moment of the process we have recorded r and s = r for some r, s. We can change s r to (s+1) r by addition of r. When s = r+1 we can change r to r+1 by a transfer. Unfortunately there is no coding system sufficiently generally known to justify giving the routine for this process in full, but the flow diagram given in Fig.1 will be sufficient for illustration.

Each 'box of the flow diagram represents a straight sequence of instructions without changes of control. The following convention is used:

(i) a dashed letter indicates the value at the end of the process represented by the box:

(ii) an undashed letter represents the initial value of a quantity.

One cannot equate similar letters appearing in different boxes, but it is intended that the following identifications be valid throughout

67.

Floyd Independently Rediscovers Turing's Ideas

[Floyd 1967]: Assigning Meaning to Programs

- Introduces verification conditions: $V_c(P, Q)$
- Describes how to massage verification conditions so that they match (can be laid on a flow chart)

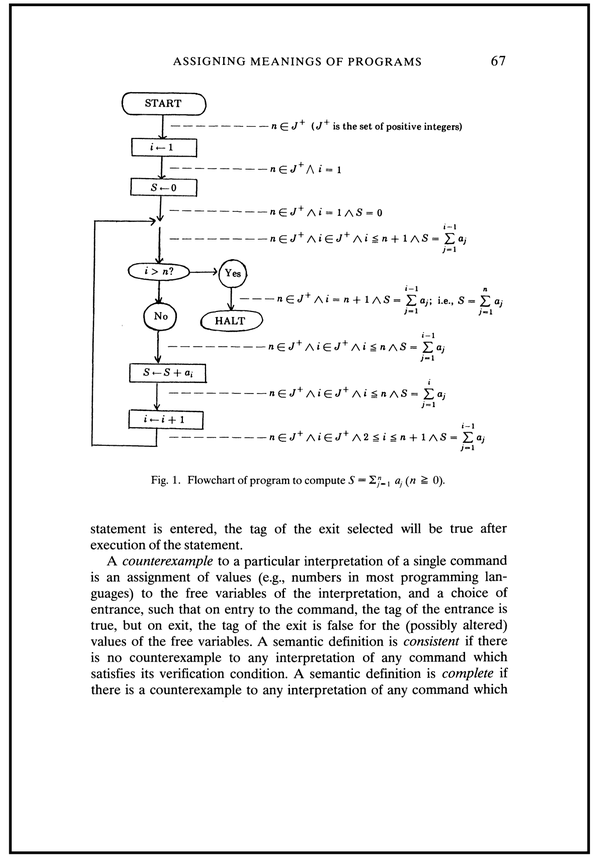


Fig. 1. Flowchart of program to compute $S = \Sigma_{j=1}^{n} a_j$ ($n \geq 0$).

statement is entered, the tag of the exit selected will be true after execution of the statement.

A *counterexample* to a particular interpretation of a single command is an assignment of values (e.g., numbers in most programming languages) to the free variables of the interpretation, and a choice of entrance, such that on entry to the command, the tag of the entrance is true, but on exit, the tag of the exit is false for the (possibly altered) values of the free variables. A semantic definition is *consistent* if there is no counterexample to any interpretation of any command which satisfies its verification condition. A semantic definition is *complete* if there is a counterexample to any interpretation of any command which

Hoare Builds on Floyd's work

[Hoare 1969]: An Axiomatic Basis for Computer Programming

- Abandons flow charts
- An axiomatic system based on "Hoare triples": $\{P\}e\{Q\}$

$$\frac{\{P\}e_1\{Q\} \qquad \{Q\}e_2\{R\}}{\{P\}e_1; e_2\{R\}} \qquad\qquad \frac{\{I \wedge B\}e\{I\}}{\{I\}\texttt{While}(B)\texttt{do } e\{I \wedge \neg B\}}$$

- Hoare emphasizes modularity
  - Decompose the problem into smaller, more manageable parts
  - Proof reuse

Separation Logic: Reasoning About Aliasing

[Reynolds 2002]: Separation Logic: A Logic for Shared Mutable Data Structures
[O'Hearn 2004]: Resources, Concurrency and Local Reasoning

- Our earlier argument was not very rigorous
- We are implicitly relying on non-aliasing
  - Memory storing n, f, and i are **disjoint**

```
// assume n > 0, i = 1, f = 1
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

```
// f = n!
```

Separation Logic: Reasoning About Aliasing

[Reynolds 2002]: Separation Logic: A Logic for Shared Mutable Data Structures
[O'Hearn 2004]: Resources, Concurrency and Local Reasoning

```
// assume n > 0, i = 1, f = 1
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
while(i <= n){
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n$$

```
    f = f * i;
```

$$n > 0 \wedge f = i! \wedge i \leq n$$

```
    i = i + 1;
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1$$

```
}
```

$$n > 0 \wedge f = (i-1)! \wedge i \leq n+1 \wedge i > n$$

```
// f = n!
```

Separation Logic: Reasoning About Aliasing

[Reynolds 2002]: Separation Logic: A Logic for Shared Mutable Data Structures
[O'Hearn 2004]: Resources, Concurrency and Local Reasoning

Key Points:

Distinguish between address and value: $\mapsto$

Separating conjunction: $\star$

```
// assume n > 0, i = 1, f = 1
```

$$n \mapsto \mathbb{K} \star \mathbb{K} > 0 \star f \mapsto (\mathbb{L}-1)! \star i \mapsto \mathbb{L} \star \mathbb{L} \leq \mathbb{K}+1$$

```
while(i <= n){
```

$$n \mapsto \mathbb{K} \star \mathbb{K} > 0 \star f \mapsto (\mathbb{L}-1)! \star i \mapsto \mathbb{L} \star \mathbb{L} \leq \mathbb{K}$$

```
    f = f * i;
```

$$n \mapsto \mathbb{K} \star \mathbb{K} > 0 \star f \mapsto \mathbb{L}! \star i \mapsto \mathbb{L} \star \mathbb{L} \leq \mathbb{K}$$

```
    i = i + 1;
```

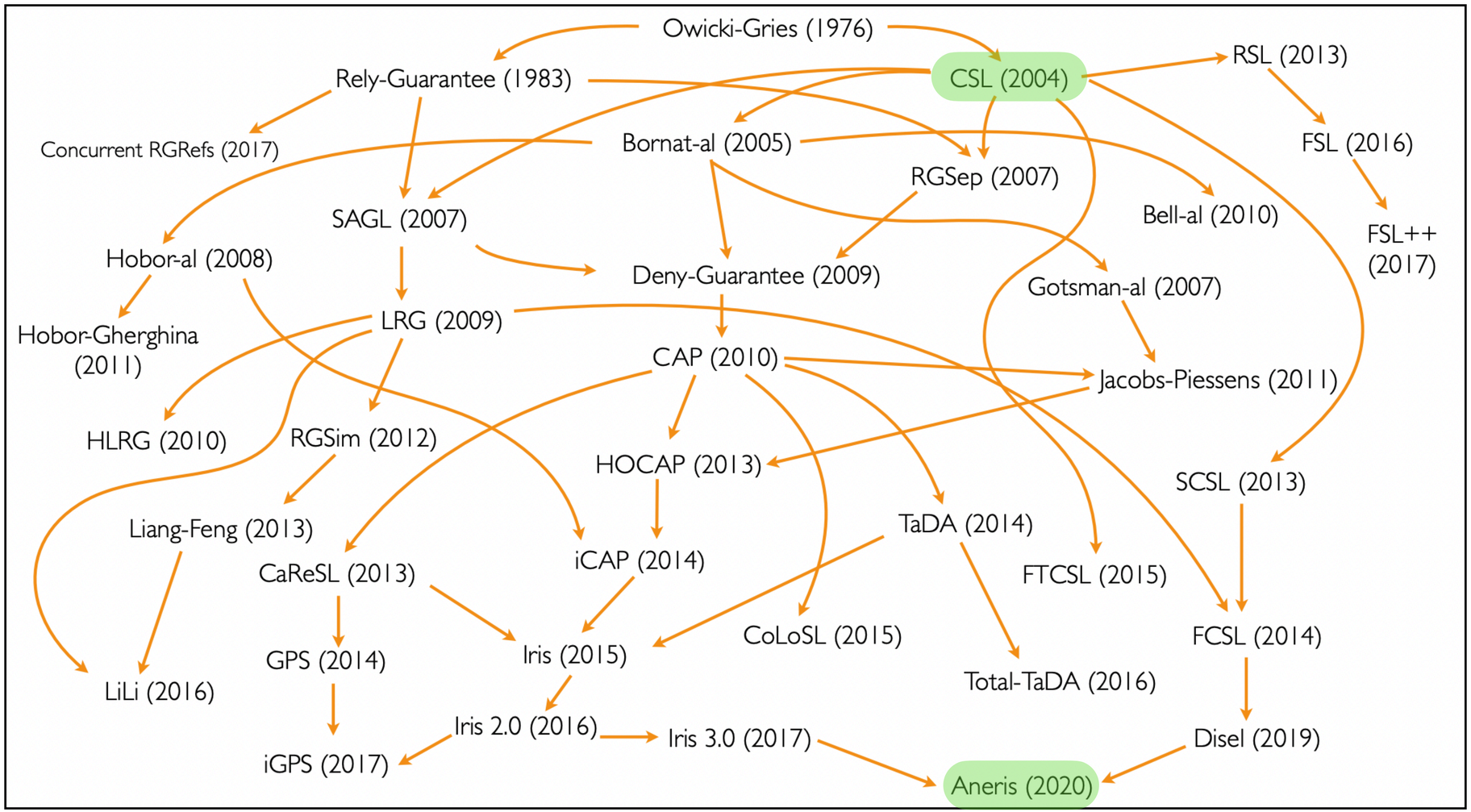
$$n \mapsto \mathbb{K} \star \mathbb{K} > 0 \star f \mapsto \mathbb{L}! \star i \mapsto \mathbb{L}+1 \star \mathbb{L} \leq \mathbb{K}$$

```
}
```

$$n \mapsto \mathbb{K} \star \mathbb{K} > 0 \star f \mapsto (\mathbb{L}-1)! \star i \mapsto \mathbb{L} \star \mathbb{L} \leq \mathbb{K}+1 \star \mathbb{L} > \mathbb{K}$$

```
// f = n!
```

Verification of Concurrent and Distributed Programs



Taken from wikimedia.org, by Ilya Sergey
Description by the author: This image depicts the "flow of ideas" that have been implemented in various logical frameworks for proving correctness of concurrent and distributed programs.

My Work

Investigating and developing the logical foundations of program verification

Verification of Distributed Systems (the Aneris project)

- Aneris: a framework for reasoning about distributed systems [ESOP'2020]
  - A verified load balancer
  - A verified two-phase commit protocol
- Verified causally-consistent distributed key-value store [POPL'21]
- Verified conflict-free replicated data types (CRDTs) [OOPSLA'22; ECOOP'23]
- Reliable communication on top of an unreliable channel [ICFP'23]

Current/future work

- Verified transactional databases
- Verified transport protocols (QUIC)
- Liveness properties of distributed systems (using Trillium)

Language-level properties and security

- Reasoning about non-interference (programs don't leak secrets) [POPL'21]
- Using hardware capabilities to enforce security of control flow [POPL'21]
- Using hardware capabilities to secure DMA devices [CSF'22]
- Purity in the presence of encapsulated memory access [OOPSLA'22]
- Using capabilities to confine untrusted (attacker) code [JACM 2023]
- Robust safety of core Hafnium functionalities (Google's hypervisor) [PLDI'23]
- Logical characterization of call stacks (well-bracketedness of control) [POPL'24]
- Denotational semantics suitable higher-order language interactions [POPL'24]
- A logical approach to type soundness [JACM 2024]

Current/future work

- Extending denotational semantics to other effects
- Correctness and security of compilers

Refinement-Based Reasoning (the Trillium project)

- Trillium: a framework for establishing refinement [POPL'24]
  - Verification of a consensus algorithm against TLA+ specs
  - Proof of termination of concurrent programs

Current/future work

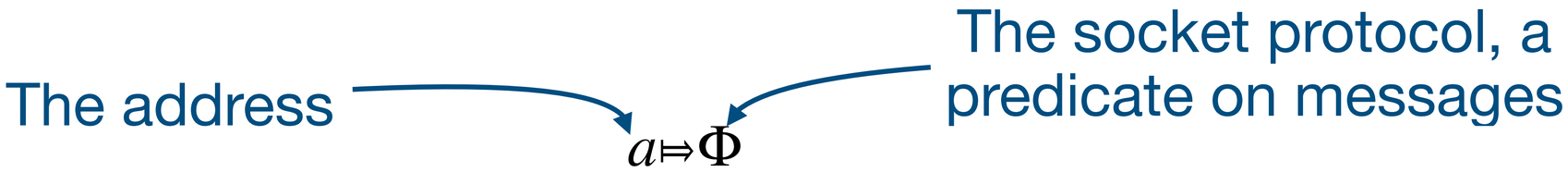
- Modular verification of liveness properties of concurrent and distributed systems

Verification of Distributed Systems

The Aneris Program Logic

[Krogh-Jespersen, Timany, Ohlenbusch, Gregersen, and Birkedal 2020]: Aneris: *A Mechanised Logic for Modular Reasoning about Distributed Systems*.

- The key to Aneris's modularity (in addition to Hoare triples, separation logic, *etc.*):
  - so-called "socket protocols"

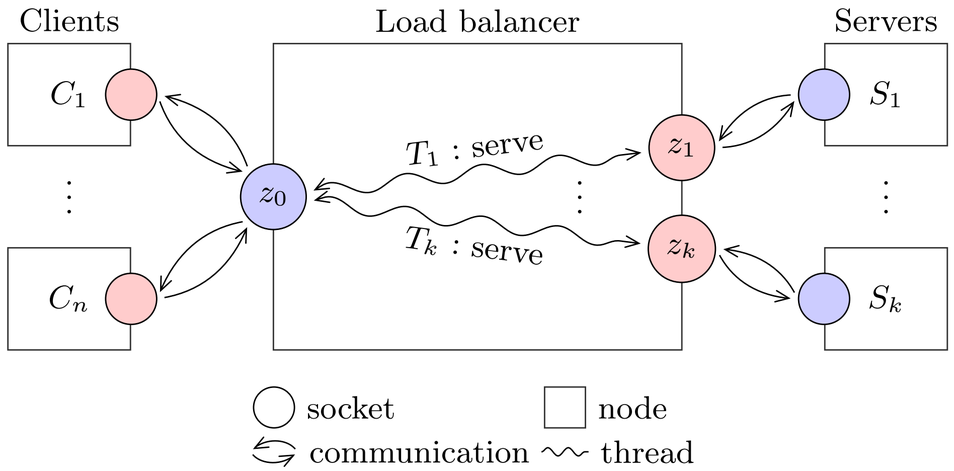


  - Sending message m to address a: we should prove $\Phi(m)$
  - Receiving message m on a socket bound to a: we know $\Phi(m)$

The Aneris Program Logic

[Krogh-Jespersen, Timany, Ohlenbusch, Gregersen, and Birkedal 2020]: Aneris: *A Mechanised Logic for Modular Reasoning about Distributed Systems*.

- A verified load balancer



- What is a good specification for a general-purpose load balancer?
  - Should act exactly as the server does

The Aneris Program Logic

[Krogh-Jespersen, Timany, Ohlenbusch, Gregersen, and Birkedal 2020]: Aneris: *A Mechanised Logic for Modular Reasoning about Distributed Systems*.

How do we state this formally?

Accepts any message m accepted by the server behind the load balancer

As long as the sender's socket protocol Ψ accepts server's response

$$\Phi_{LB}(m) = \Phi_{SRV}(m) \star \exists \Psi\,.\ \mathsf{sender}(m) \Rightarrow\!\!\!\triangleright \Psi \star \forall m'\,.\ \mathsf{server\_response}(m, m') \implies \Psi(m')$$

Higher-order: quantifies over protocols to define a protocol

Necessary for modularity: no need to specify upfront who can contact a node

Refinement-Based Reasoning to Strengthen Program Logics

Limitation of Higher-Order Program Logics

[Timany, Gregersen, Stefanesco, Hinrichsen, Gondelman, Nieto, and Birkedal 2024]: Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement.

- Safety versus liveness properties
  - Safety properties: nothing “bad” ever happens
    - Example: the program does not crash
  - Liveness properties: something “good” will eventually happen
    - Example: the server will eventually respond to all requests
- **Trillium**: a higher-order program logic capable of liveness reasoning
  - This was thought to be impossible prior to Trillium

Limitation of Higher-Order Program Logics

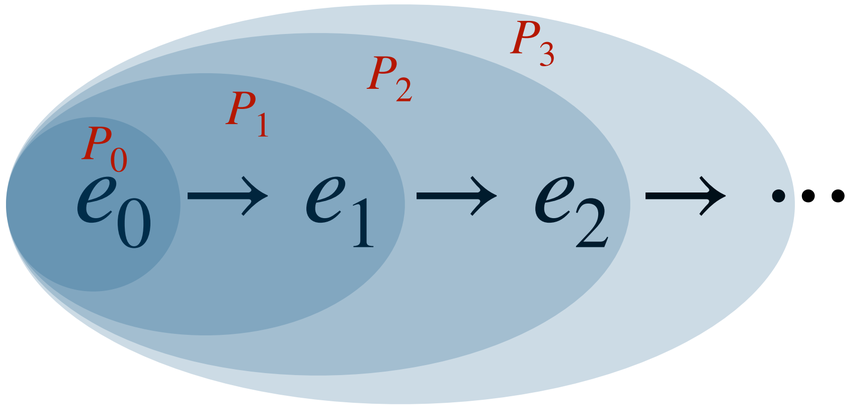
[Timany, Gregersen, Stefanesco, Hinrichsen, Gondelman, Nieto, and Birkedal 2024]: Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement.

- Recall the inherent circularity in high-order specs:

$$\Phi_{LB}(m) = \Phi_{SRV}(m) \star \exists \Psi\,.\ \mathsf{sender}(m) \Rrightarrow \Psi \star \forall m'\,.\ \mathsf{server\_response}(m, m') \implies \Psi(m')$$

- State-of-the-art program logics use the so-called “step-indexing” technique:
  - Break the cycle by stratifying the program logic along program execution

When proving P, we prove $\forall n\,.\ P_n$:



This works for safety properties but not liveness.

Limitation of Higher-Order Program Logics

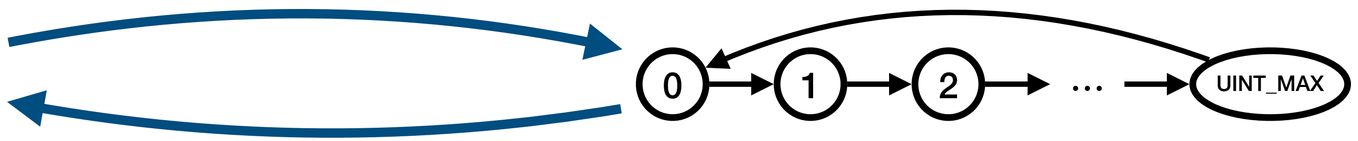
[Timany, Gregersen, Stefanesco, Hinrichsen, Gondelman, Nieto, and Birkedal 2024]: Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement.

Trillium: a novel higher-order (based on step-indexing) program logic

- To establish refinements between programs and a high-level models (LTSs)
- Properties (including liveness) satisfied by the LTS are also satisfied by the program
- Supports both safety and liveness properties

```
void incr_loop(unsigned int *p){
    while(1)
        (*p)++;
}
```

Refinement



Visits every number arbitrarily many times without skipping

Formal and Foundational Proofs

Formal and Foundational Proofs

All proofs we develop are mechanized in the proof assistant Coq

- Why?
  - Both the programs we study, and the program logics we use to study them are complex and not always intuitive!

Formal and Foundational Proofs

All proofs we develop are mechanized in the proof assistant Coq

- Why?
  - Both the programs we study, and the program logics we use to study them are complex and not always intuitive!
- Advantage of using proof assistants:
  - You **do not need to rely on** intuition

Formal and Foundational Proofs

All proofs we develop are mechanized in the proof assistant Coq

- Why?
  - Both the programs we study, and the program logics we use to study them are complex and not always intuitive!
- Advantage of using proof assistants:
  - You **do not need to rely on** intuition
- Disadvantage of using proof assistants:
  - You **cannot rely on** intuition

```
Theorem add_com n m : n + m = m + n.
Proof.
  revert m.
  induction n as [|n IHn].
  - intros m.
    simpl.
    induction m as [|m IHm].
    + simpl. trivial.
    + simpl. rewrite <- IHm. reflexivity.
  - intros m.
    induction m as [|m IHm].
    + simpl.
      rewrite IHn. simpl. reflexivity.
    + simpl.
      rewrite IHn.
      simpl.
      rewrite <- IHm.
      simpl.
      rewrite IHn.
      reflexivity.
Qed.
```

Rigor,
the Foundations of Mathematics,
and Computer Science

Conception of Set Theory

[Cantor 1874]: Ueber eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen

- Proves that there are different magnitudes of infinity
  - $|Alg| = |\mathbb{N}|$
  - $|Alg| < |\mathbb{R}|$
- This is the beginning of set theory

Georg Cantor

Taken from https://commons.wikimedia.org/wiki/File:Georg_Cantor3.jpg

Ueber eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen.

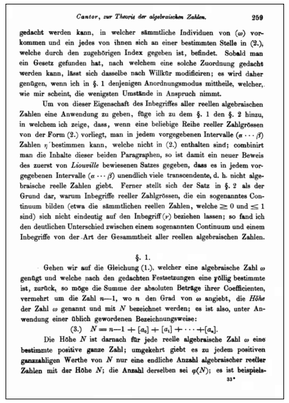
(Von Herrn *Cantor* in Halle a. S.)

Unter einer reellen algebraischen Zahl wird allgemein eine reelle Zahlgrösse ω verstanden, welche einer nicht identischen Gleichung von der Form genügt:

(1.) $\quad a_n \omega^n + a_1 \omega^{n-1} + \cdots + a_n = 0,$

wo $n, a_0, a_1, \cdots a_n$ ganze Zahlen sind; wir können uns hierbei die Zahlen n und a_0 positiv, die Coefficienten $a_0, a_1 \cdots a_n$ ohne gemeinschaftlichen Theiler und die Gleichung (1.) irreductibel denken; mit diesen Festsetzungen wird erreicht, dass nach den bekannten Grundsätzen der Arithmetik und Algebra die Gleichung (1.), welcher eine reelle algebraische Zahl genügt, eine völlig bestimmte ist; umgekehrt gehören bekanntlich zu einer Gleichung von der Form (1.) höchstens soviel reelle algebraische Zahlen ω, welche ihr genügen, als ihr Grad n angiebt. Die reellen algebraischen Zahlen bilden in ihrer Gesammtheit einen Inbegriff von Zahlgrössen, welcher mit (ω) bezeichnet werde; es hat derselbe, wie aus einfachen Betrachtungen hervorgeht, eine solche Beschaffenheit, dass in jeder Nähe irgend einer gedachten Zahl ω unendlich viele Zahlen aus (ω) liegen; um so auffallender dürfte daher für den ersten Anblick die Bemerkung sein, dass man den Inbegriff (ω) dem Inbegriffe aller ganzen positiven Zahlen ν, welcher durch das Zeichen (ν) angedeutet werde, eindeutig zuordnen kann, so dass zu jeder algebraischen Zahl ω eine bestimmte ganze positive Zahl ν und umgekehrt zu jeder positiven ganzen Zahl ν eine völlig bestimmte reelle algebraische Zahl ω gehört, dass also, um mit anderen Worten dasselbe zu bezeichnen, der Inbegriff (ω) in der Form einer unendlichen gesetzmässigen Reihe:

(2.) $\quad \omega_1, \omega_2 \cdots \omega_\nu, \cdots$



Cantor, zur Theorie der algebraischen Zahlen. 259

gedacht werden kann, in welcher sämmtliche Individuen von (ω) vorkommen und ein jedes von ihnen sich an einer bestimmten Stelle in (2.), welche durch den zugehörigen Index gegeben ist, befindet. Sobald man ein Gesetz gefunden hat, nach welchem eine solche Zuordnung gedacht werden kann, lässt sich dasselbe nach Willkür modificiren; es wird daher genügen, wenn ich in §. 1 denjenigen Anordnungsmodus mittheile, welcher, wie mir scheint, die wenigsten Umstände in Anspruch nimmt.

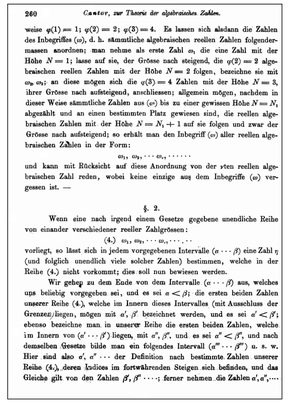
Um von dieser Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen eine Anwendung zu geben, füge ich zu dem §. 1 den §. 2 hinzu, in welchem ich zeige, dass, wenn eine beliebige Reihe reeller Zahlgrössen von der Form (2.) vorliegt, man in jedem vorgegebenen Intervalle $(\alpha \cdots \beta)$ Zahlen η bestimmen kann, welche nicht in (2.) enthalten sind; combinirt man die Inhalte dieser beiden Paragraphen, so ist damit ein neuer Beweis des zuerst von *Liouville* bewiesenen Satzes gegeben, dass es in jedem vorgegebenen Intervalle $(\alpha \cdots \beta)$ unendlich viele transcendente, d. h. nicht algebraische reelle Zahlen giebt. Ferner stellt sich der Satz in §. 2 als der Grund dar, warum Inbegriffe reeller Zahlgrössen, die ein sogenanntes Continuum bilden (etwa die sämmtlichen reellen Zahlen, welche $\geqq 0$ und $\leqq 1$ sind) sich nicht eindeutig auf den Inbegriff (ν) beziehen lassen; so fand ich den deutlichen Unterschied zwischen einem sogenannten Continuum und einem Inbegriffe von der Art der Gesammtheit aller reellen algebraischen Zahlen.

§. 1.

Gehen wir auf die Gleichung (1.), welcher eine algebraische Zahl ω genügt und welche nach den gedachten Festsetzungen eine völlig bestimmte ist, zurück, so möge die Summe der absoluten Beträge ihrer Coefficienten, vermehrt um die Zahl $n-1$, wo n den Grad von ω angiebt, die *Höhe* der Zahl ω genannt und mit N bezeichnet werden; es ist also, unter Anwendung einer üblich gewordenen Bezeichnungsweise:

(3.) $\quad N = n - 1 + |a_0| + |a_1| + \cdots + |a_n|.$

Die Höhe N ist darnach für jede reelle algebraische Zahl ω eine bestimmte positive ganze Zahl; umgekehrt giebt es zu jedem positiven ganzzahligen Werthe von N nur eine endliche Anzahl algebraischer reeller Zahlen mit der Höhe N; die Anzahl derselben sei $\varphi(N)$; es ist beispiels-



260 *Cantor, zur Theorie der algebraischen Zahlen.*

weise $\varphi(1) = 1$; $\varphi(2) = 2$; $\varphi(3) = 4$. Es lassen sich alsdann die Zahlen des Inbegriffes (ω), d. h. sämmtliche algebraischen reellen Zahlen folgendermassen anordnen: man nehme als erste Zahl ω_1 die eine Zahl mit der Höhe $N = 1$; lasse auf sie, der Grösse nach steigend, die $\varphi(2) = 2$ algebraischen reellen Zahlen mit der Höhe $N = 2$ folgen, bezeichne sie mit ω_2, ω_3; an diese mögen sich die $\varphi(3) = 4$ Zahlen mit der Höhe $N = 3$, ihrer Grösse nach aufsteigend, anschliessen; allgemein mögen, nachdem in dieser Weise sämmtliche Zahlen aus (ω) bis zu einer gewissen Höhe $N = N_1$ abgezählt und an einen bestimmten Platz gewiesen sind, die reellen algebraischen Zahlen mit der Höhe $N = N_1 + 1$ auf sie folgen und zwar der Grösse nach aufsteigend; so erhält man den Inbegriff (ω) aller reellen algebraischen Zahlen in der Form:

$$\omega_1, \omega_2, \cdots \omega_\nu, \cdots,$$

und kann mit Rücksicht auf diese Anordnung von der νten reellen algebraischen Zahl reden, wobei keine einzige aus dem Inbegriffe (ω) vergessen ist. —

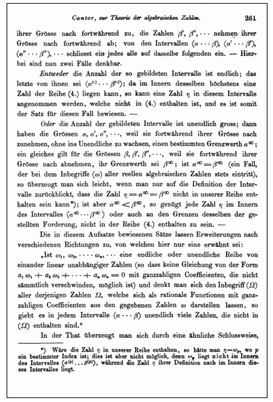
§. 2.

Wenn eine nach irgend einem Gesetze gegebene unendliche Reihe von einander verschiedener reeller Zahlgrössen:

(4.) $\quad \omega_1, \omega_2 \cdots \omega_\nu, \cdots$

vorliegt, so lässt sich in jedem vorgegebenen Intervalle $(\alpha \cdots \beta)$ eine Zahl η (und folglich unendlich viele solcher Zahlen) bestimmen, welche in der Reihe (4.) nicht vorkommt; dies soll nun bewiesen werden.

Wir gehen zu dem Ende von dem Intervalle $(\alpha \cdots \beta)$ aus, welches uns beliebig vorgegeben sei, und es sei $\alpha < \beta$; die ersten beiden Zahlen unserer Reihe (4.), welche im Innern dieses Intervalles (mit Ausschluss der Grenzen) liegen, mögen mit α', β' bezeichnet werden, und es sei $\alpha' < \beta'$; ebenso bezeichne man in unserer Reihe die ersten beiden Zahlen, welche im Innern von $(\alpha' \cdots \beta')$ liegen, mit α'', β'', und es sei $\alpha'' < \beta''$, und nach demselben Gesetze bilde man ein folgendes Intervall $(\alpha''' \cdots \beta''')$ u. s. w. Hier sind also α', $\alpha'' \cdots$ der Definition nach bestimmte Zahlen unserer Reihe (4.), deren Indices im fortwährenden Steigen sich befinden, und das Gleiche gilt von den Zahlen β', $\beta'' \cdots$; ferner nehmen die Zahlen $\alpha', \alpha'', \ldots$



Cantor, zur Theorie der algebraischen Zahlen. 261

ihrer Grösse nach fortwährend zu, die Zahlen $\beta', \beta'', \cdots$ nehmen ihrer Grösse nach fortwährend ab; von den Intervallen $(\alpha \cdots \beta)$, $(\alpha' \cdots \beta')$, $(\alpha'' \cdots \beta'')$, $\cdots$ schliesst ein jedes alle auf dasselbe folgenden ein. — Hierbei sind nun zwei Fälle denkbar.

Entweder die Anzahl der so gebildeten Intervalle ist endlich; das letzte von ihnen sei $(\alpha^{(\nu)} \cdots \beta^{(\nu)})$; da im Innern desselben höchstens eine Zahl der Reihe (4.) liegen kann, so kann eine Zahl η in diesem Intervalle angenommen werden, welche nicht in (4.) enthalten ist, und es ist somit der Satz für diesen Fall bewiesen. —

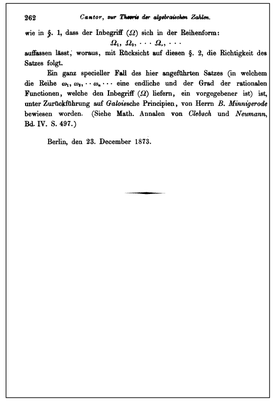
Oder die Anzahl der gebildeten Intervalle ist unendlich gross; dann haben die Grössen $\alpha, \alpha', \alpha'', \cdots$, weil sie fortwährend ihrer Grösse nach zunehmen, ohne ins Unendliche zu wachsen, einen bestimmten Grenzwerth α^∞; ein gleiches gilt für die Grössen $\beta, \beta', \beta'', \cdots$, weil sie fortwährend ihrer Grösse nach abnehmen, ihr Grenzwerth sei β^∞; ist $\alpha^\infty = \beta^\infty$ (ein Fall, der bei dem Inbegriffe (ω) aller reellen algebraischen Zahlen stets eintritt), so überzeugt man sich leicht, wenn man nur auf die Definition der Intervalle zurückblickt, dass die Zahl $\eta = \alpha^\infty = \beta^\infty$ nicht in unserer Reihe enthalten sein kann*); ist aber $\alpha^\infty < \beta^\infty$, so genügt jede Zahl η im Innern des Intervalles $(\alpha^\infty \cdots \beta^\infty)$ oder auch an den Grenzen desselben der gestellten Forderung, nicht in der Reihe (4.) enthalten zu sein. —

Die in diesem Aufsatze bewiesenen Sätze lassen Erweiterungen nach verschiedenen Richtungen zu, von welchen hier nur eine erwähnt sei:

„Ist $\omega_1, \omega_2 \cdots \omega_\nu \cdots$ eine endliche oder unendliche Reihe von einander linear unabhängiger Zahlen (so dass keine Gleichung von der Form $a_1\omega_1 + a_2\omega_2 + \cdots + a_\nu\omega_\nu = 0$ mit ganzzahligen Coefficienten, die nicht sämmtlich verschwinden, möglich ist) und denkt man sich den Inbegriff (Ω) aller derjenigen Zahlen Ω, welche sich als rationale Functionen mit ganzzahligen Coefficienten aus den gegebenen Zahlen ω darstellen lassen, so giebt es in jedem Intervalle $(\alpha \cdots \beta)$ unendlich viele Zahlen, die nicht in (Ω) enthalten sind."

In der That überzeugt man sich durch eine ähnliche Schlussweise,

*) Wäre die Zahl η in unserer Reihe enthalten, so hätte man $\omega_\nu = \omega_\mu$, wo μ ein bestimmter Index ist; dies ist aber nicht möglich, denn ω_μ liegt nicht im Innern des Intervalles $(\alpha^{(\nu)} \ldots \beta^{(\nu)})$, während die Zahl η ihrer Definition nach im Innern dieses Intervalles liegt.



262 *Cantor, zur Theorie der algebraischen Zahlen.*

wie in §. 1, dass der Inbegriff (Ω) sich in der Reihenform:

$$\Omega_1, \Omega_2, \cdots \Omega_\nu, \cdots$$

auffassen lässt, woraus, mit Rücksicht auf diesen §. 2, die Richtigkeit des Satzes folgt.

Ein ganz specieller Fall des hier angeführten Satzes (in welchem die Reihe $\omega_1, \omega_2, \cdots \omega_\nu \cdots$ eine endliche und der Grad der rationalen Functionen, welche den Inbegriff (Ω) liefern, ein vorgegebener ist) ist, unter Zurückführung auf *Galois*sche Principien, von Herrn *B. Minnigerode* bewiesen worden. (Siehe Math. Annalen von *Clebsch* und *Neumann*, Bd. IV. S. 497.)

Berlin, den 23. December 1873.

“From the paradise that Cantor created for us no-one shall be able to expel us.”

David Hilbert (1926), about set theory

David Hilbert

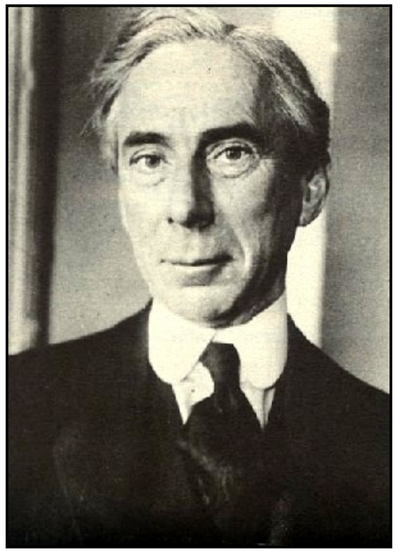
Taken from https://commons.wikimedia.org/wiki/File:Hilbert.jpg

Frege Logicizes Arithmetic

[Frege 1884]: Die Grundlagen der Arithmetik
[Frege 1893, 1902]: Grundgesetze der Arithmetik

- Bases all of arithmetic on set theory and logic
- Allows unrestricted comprehension:
  - $\{x \mid P(x)\}$ where P can be any logical formula

Bertrand Russel



Taken from https://commons.wikimedia.org/wiki/File:Bertrand_Russell_in_1924.jpg

Gottlob Frege

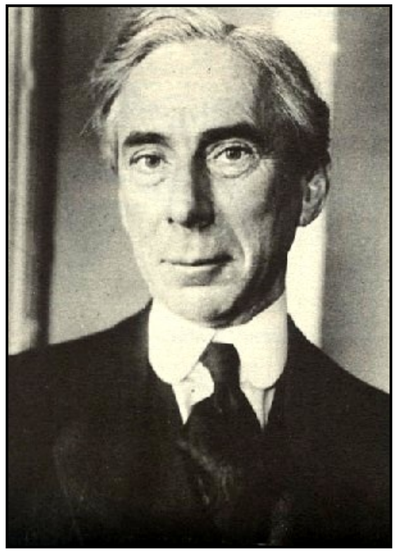
Taken from https://commons.wikimedia.org/wiki/File:Young_frege.jpg

Frege Logicizes Arithmetic

[Frege 1884]: Die Grundlagen der Arithmetik
[Frege 1893, 1902]: Grundgesetze der Arithmetik

- Bases all of arithmetic on set theory and logic
- Allows unrestricted comprehension:
  - $\{x \mid P(x)\}$ where P can be any logical formula
- In 1902, Russel writes to Frege about a paradox
  - Russel's paradox: define a set S as
    - $S := \{x \mid x \notin x\}$
- Today, set theory, *e.g.*, ZF(C), restricts comprehension

Bertrand Russel



Taken from https://commons.wikimedia.org/wiki/File:Bertrand_Russell_in_1924.jpg

Gottlob Frege

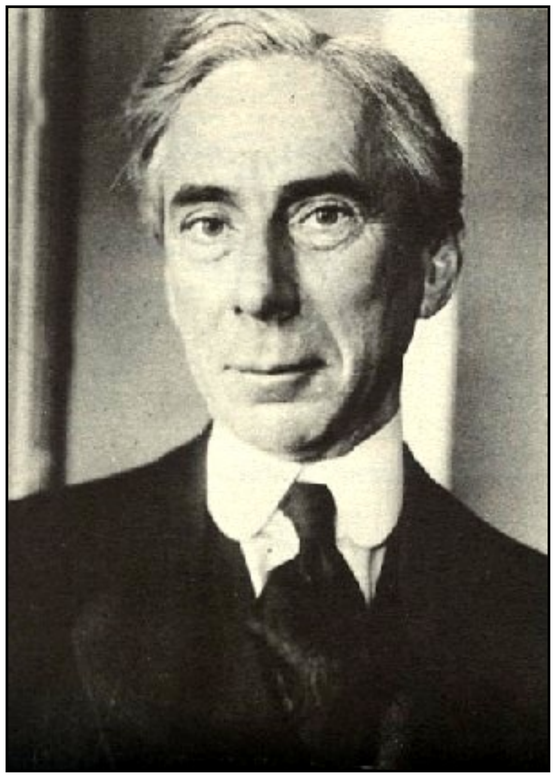
Taken from https://commons.wikimedia.org/wiki/File:Young_frege.jpg

Types to Avoid Paradoxes

[Whitehead and Russel 1910–1913] Principia Mathematica

- Introduce an infinite hierarchy of types: $tp_0, tp_1, tp_2, \ldots$
  - Individual elements at tp_0
  - Allows quantifying over all objects at tp_i to define a collection which then is at tp_{i+1}
- This is the first explicit, formal use of types in mathematics

Bertrand Russel



Taken from https://commons.wikimedia.org/wiki/File:Bertrand_Russell_in_1924.jpg

The Entscheidungsproblem (Decision Problem)

[Hilbert and Ackermann 1928]: Grundzüge der Theoretischen Logik

- Propose that there should be a system for deciding problems
  - **Axiomatic version**:
    an axiomatic system that for any P can prove P, or can prove $\neg P$
  - **Computational version** :
    a "program" that for any P, can determine truth of P, *e.g.*, a program that returns 1 if P holds, returns 0 if $\neg P$ holds

Wilhelm Ackermann

Taken from https://commons.wikimedia.org/wiki/File:Ackermann_Wilhelm.jpg

David Hilbert

Taken from https://commons.wikimedia.org/wiki/File:Hilbert.jpg

Gödel's Incompleteness Theorems

[Gödel 1931]: Über Formal Unentscheidbare Sätze der "Pincipia Mathematica" und Verwandter Systeme I

- A negative answer to the axiomatic version of the Entscheidungsproblem
- Proves that no axiomatic system, expressive enough to include arithmetic, can decide all propositions

Kurt Gödel

Taken from https://commons.wikimedia.org/wiki/File:Kurt_gödel.jpg

Church Proposes a System

[Church 1932]: A Set of Postulates for the Foundation of Logic

- Axioms (postulates) were intended as a sound, but not complete system
  - λ terms are included for succinct expression of formulas

Alonzo Church

Photograph by Paul Halmos, taken from https://mathshistory.st-andrews.ac.uk/Biographies/Church/pictdisplay/

Church Proposes a System

[Church 1932]: A Set of Postulates for the Foundation of Logic

- Axioms (postulates) were intended as a sound, but not complete system
  - λ terms are included for succinct expression of formulas
- In 1935, Kleene and Rosser found an inconsistency
  - In 1935–1936, Church removes the logical parts
  - Uses λ terms as a model of computation to refute Entscheidungsproblem

Stephen C. Kleene

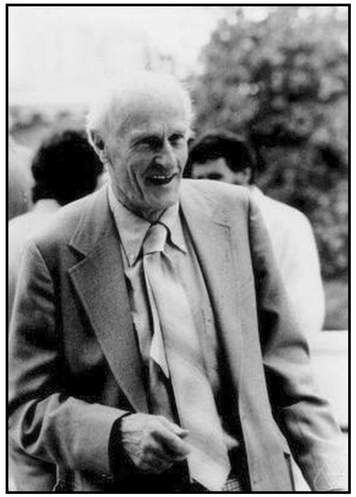


Photo by Konrad Jacobs, Erlangen, taken from https://commons.wikimedia.org/wiki/File:Georg_Cantor3.jpg

Alonzo Church

Photograph by Paul Halmos, taken from https://mathshistory.st-andrews.ac.uk/Biographies/Church/pictdisplay/

Church Proposes a System

[Church 1932]: A Set of Postulates for the Foundation of Logic

- Axioms (postulates) were intended as a sound, but not complete system
  - λ terms are included for succinct expression of formulas
- In 1935, Kleene and Rosser found an inconsistency
  - In 1935–1936, Church removes the logical parts
  - Uses λ terms as a model of computation to refute Entscheidungsproblem
- In 1940, Church introduces typed lambda calculus
  - As a logical system based on “type theory”

Stephen C. Kleene

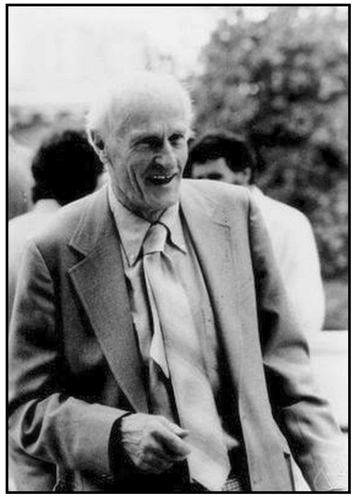


Photo by Konrad Jacobs, Erlangen, taken from https://commons.wikimedia.org/wiki/File:Georg_Cantor3.jpg

Alonzo Church

Photograph by Paul Halmos, taken from https://mathshistory.st-andrews.ac.uk/Biographies/Church/pictdisplay/

Models of Computation and the Entscheidungsproblem

[Church 1936]: An Unsolvable Problem of Elementary Number Theory
[Turing 1936]: On Computable Numbers, with an Application to the Entscheidungsproblem

- Gödel (1935): Herbrand–Gödel general recursive functions
- Church (1935–1936): λ-calculus
  - The first negative answer to computation Entscheidungsproblem
- Turing (1936): Turing machines
  - The second negative answer to computation Entscheidungsproblem
  - Turing immediately recognizes that his machines are equivalent to λ-calculus
- Kleene (1936): general recursive functions are equivalent to λ-calculus

Curry-Howard Correspondence

[Curry and Howard, 1934–1969]

- Observed that typed λ-calculus does not need logical principles added on top of it
  - **Propositions are types**
  - **Proofs are programs**
- Observed for propositional logic at first
  - Has been extended to a myriad of other systems

Haskell Brooks Curry

Taken from https://mathshistory.st-andrews.ac.uk/Biographies/Curry/

William Alvin Howard

Photo by Andrej Bauer, taken from https://commons.wikimedia.org/wiki/File:William_Alvin_Howard_May_2004.jpg

Calculus of Constructions (CoC)

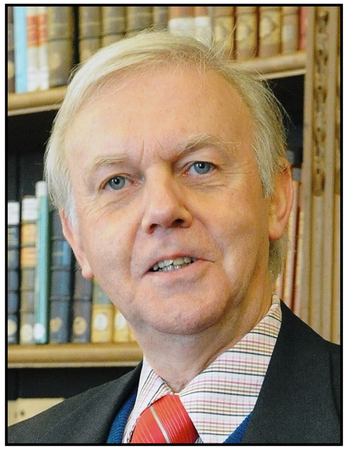
[Coquand, Huet 1986]: The Calculus of Constructions

- A typed λ-calculus with a very expressive type system (**the core of the Coq proof assistant**)
- CoC features a single universes (types of types): $*$
  - Terms (programs) have types
    - $0,1,2,\ldots : nat; add : nat \to nat \to nat; \ldots$
  - Types are also terms, the universe is the type of types
    - $nat : * ; nat \to nat \to nat : *$
    - $True : * ; False : * ;$
    - $\left(\forall (n : nat) \,.\, Even(n) \to Odd(n+1)\right) : *$
  - CoC is a very strong system for proofs but it has limitations for programming

Thierry Coquand

Picture by Andrej Bauer, taken from https://commons.wikimedia.org/wiki/File:Thierry_Coquand_(cropped).jpg

Gérard Huet

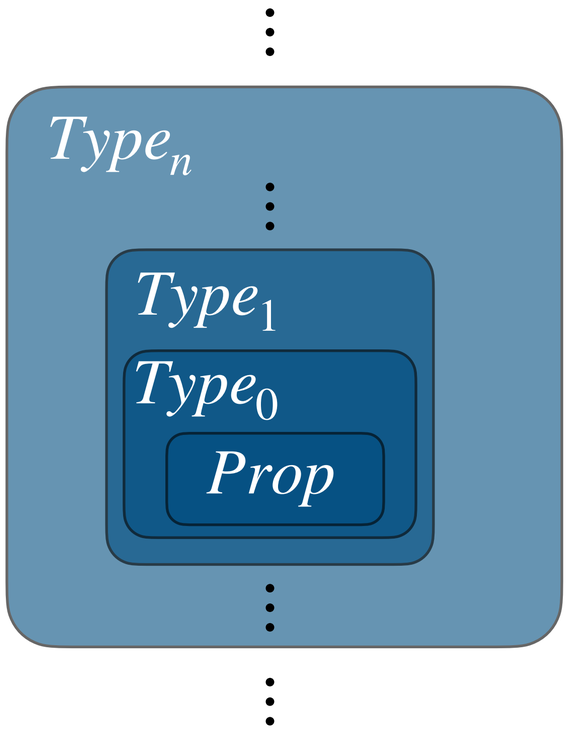


Taken from https://awards.acm.org/award-recipients/huet_1246701

Extended Calculus of Constructions (ECC)

[Luo, 1990]: An Extended Calculus of Construction

- Keeps the sort $*$, renamed to $Prop$ only for propositions
- Features a hierarchy of universes
  - $Type_0 : Type_1 : Type_2 : \ldots$
- The hierarchy is *predicative* (like in Principia)
  - $(\forall x : Type_i \,.\, A) : Type_{i+1}$
    if A itself is a valid type in $Type_{i+1}$
- The hierarchy is *cumulative*
  - if $A : Type_i$ and $i \leq j$ then $A : Type_j$



Zhaohui Luo

Taken from https://www.cs.rhul.ac.uk/home/zhaohui/

Predicative Calculus of Inductive Constructions (pCIC)

[Pfenning and Paulin-Moring 1990]: Inductively defined types in the Calculus of Constructions
[Paulin-Moring 1996]: Définitions Inductives en Théorie des Types d'Ordre Supérieur

- Add inductive types to Coq (CoC; ECC)
  - $list\ A : Type_i$ whenever $A : Type_i$
- First, only for the impredicative system (CoC)
  - Using Church encoding
- Later, added as primitives in all universes

Frank Pfenning

Photo by Andrej Bauer, taken from https://commons.wikimedia.org/wiki/File:Frank_Pfenning.jpg

Christine Paulin-Mohring

Taken from https://www.lri.fr/~paulin/

Predicative Calculus of Cumulative Inductive Constructions (pCuIC)

[Timany and Jacobs 2015]: First Steps Towards Cumulative Inductive Types in CIC
[Timany and Sozeau 2017-2018]: Cumulative Inductive Types In Coq

- Recall cumulativity: if $A : Type_i$ and $i \leq j$ then $A : Type_j$

Bart Jacobs

Taken from https://distrinet.cs.kuleuven.be/people/BartJacobs

Matthieu Sozeau

Taken from https://sozeau.gitlabpages.inria.fr/www/

Predicative Calculus of Cumulative Inductive Constructions (pCuIC)

[Timany and Jacobs 2015]: First Steps Towards Cumulative Inductive Types in CIC
[Timany and Sozeau 2017-2018]: Cumulative Inductive Types In Coq

- Recall cumulativity: if $A : Type_i$ and $i \leq j$ then $A : Type_j$
- Two versions of $list\ A$, one in $Type_i$ and one in $Type_j$
  - What is the relationship between them?

Bart Jacobs

Taken from https://distrinet.cs.kuleuven.be/people/BartJacobs

Matthieu Sozeau

Taken from https://sozeau.gitlabpages.inria.fr/www/

Predicative Calculus of Cumulative Inductive Constructions (pCuIC)

[Timany and Jacobs 2015]: First Steps Towards Cumulative Inductive Types in CIC
[Timany and Sozeau 2017-2018]: Cumulative Inductive Types In Coq

- Recall cumulativity: if $A : Type_i$ and $i \leq j$ then $A : Type_j$
- Two versions of $list\ A$, one in $Type_i$ and one in $Type_j$
  - What is the relationship between them?
- More interesting, we can use inductive types to define the type $Grp_i : Type_i$ of groups whose carrier set (type) is in $Type_i$
  - For $i \leq j$, what is the relationship between Grp_i and Grp_j?

Bart Jacobs

Taken from https://distrinet.cs.kuleuven.be/people/BartJacobs

Matthieu Sozeau

Taken from https://sozeau.gitlabpages.inria.fr/www/

Predicative Calculus of Cumulative Inductive Constructions (pCuIC)

[Timany and Jacobs 2015]: First Steps Towards Cumulative Inductive Types in CIC
[Timany and Sozeau 2017-2018]: Cumulative Inductive Types In Coq

- Recall cumulativity: if $A : Type_i$ and $i \leq j$ then $A : Type_j$
- Two versions of $list\ A$, one in $Type_i$ and one in $Type_j$
  - What is the relationship between them?

They are equal!

- More interesting, we can use inductive types to define the type $Grp_i : Type_i$ of groups whose carrier set (type) is in $Type_i$
  - For $i \leq j$, what is the relationship between Grp_i and Grp_j?

Grp_i is included in Grp_j!

Bart Jacobs

Taken from https://distrinet.cs.kuleuven.be/people/BartJacobs

Matthieu Sozeau

Taken from https://sozeau.gitlabpages.inria.fr/www/

Thanks!