Tales from verification history.

You might be asking yourself why I have a large tree on the screen. There is an Indian author called R. K. Narayan who wrote in English well before the generation of authors like Salman Rushdie and Vikram Seth who would gain international recognition. Narayan wrote about life in small-town India and one collection of short stories is titled *Under the Banyan Tree*. It is about a man who observes the lives of people around him and then becomes a story teller. He sits under a banyan tree in the village tells the people stories. At some point during my PhD, I thought I could do the same. Look at the papers I had read and try to reverse engineer the stories behind them. I can't guarantee that it's true, but I hope it makes for a good story!

The story of program verification is much like a literary epic. If you think of the Greek Myths, they are spread over different ages. You have the Age of Gods, which in our case would begin with Aristotle who codified the laws of reasoning and strech to Kurt Goedel who blew a hole in the foundations of mathematics. Then, you have the Age of Heroes, which I view as the period in which the foundations of program verification as we know it today were laid. In the myths, the Age of Men is the third age, and it's an open question whether research has already reached that point or if there is still a hero out there. Like the literary epics, this story has many chapters and I can only give you a glimpse of a few chapters here.

Prologue. A Comedy of Errors.

Let's begin with a tangentially related event in the mist of computing history.

<Slide 3,4>

This is Rear Admiral Grace Murray Hopper, also known as 'Amazing Grace' to those around her. She earned a PhD in mathematics. She came from a family which had a strong tradition of serving in the armed forces and during the second world war, she took a leave of absence from her teaching position to join the United States Navy Reserves. I think she may have been the senior-most woman in the Navy when she finally retired. Her first project involved working with Howard Aiken on the Mark I computer in Havard University. In September 1947, one of the engineers involved with the project was puzzled because the machine had stalled. With some investigation, he discovered that there was literally a bug in the machine!

The term 'bug' was already in widespread use at the time to describe errors in an electro-mechanical device, which is why the log book says 'first case of bug being found.' The engineers were really kicked that they had found an actual bug in a computer. They put the word out that they were the first people to 'debug' a computer and I believe this is where the term debugging originated. This moth is preserved to this day in the Smithsonian National Museum of American History. Unfortunately, not all bugs are as easy to identify or eliminate.

<Slide 5,6>

Sometimes a bug is just embarrassing as with this ATM here. Other times, it can be a bit more

dramatic.

This is the USS Yorktown CG-48. It was commisioned on American Independence day in 1984. It is equipped with anti-ship, anti-submarine, anti-aircraft and and pretty much anti-everything missiles, not to mention a few torpedoes and even a canon. It has been involved in quite a few missions and conflicts. In 1996, the United States Navy decided that it needed to modernise the fleet and have some *Smart Ships*. So what did they do? They put a fibre optic LAN on some ships and installed 27 machines running Windows NT 4.0. One day, while they were undertaking maneuvers somewhere, a crew member entered a 0 *in the kitchen inventory software.* This caused a divsion-by-zero error, which eventually brought down all the machines on the network and paralyzed the ship's propulsion system. The ship was stuck in the water for about 3 hours. The reports differ on how they eventually recovered, but it's easy to imagine that many other things could have gone wrong with such a ship.

If you attended a conference in formal methods and verification between about 1998 and 2003, I'm guessing that one in three or four talks would have started with the example of the Ariane 5 rocket. The Ariane 5 is a European launch vehicle, and it's test flight was on June 4th 1996. You can watch it here:
http://www.youtube.com/watch?v=kYUrqdUyEpI

The Ariane 5 reused technology from its predecessor the Ariane 4, which had undertaken several flights. But Ariane 5 had a different flight trajectory. There was a conversion error from a 64-bit floating point number to a 16-bit integer. This triggered an exception which was not handled and ... boom!

This list could go on, but I'll just end with an example from Airbus since another Airbus flight has been in the news recently.
http://www.youtube.com/watch?v=SCwYAzqvcrQ

This is an Airbus A 320 machine flown by a computer. It was demoed in the Air Show in 1988 where the computer was supposed to land the plane.

The disturbing part about these anecdotes is that they are not rare. I could spend an entire day telling such stories and not be done. Even worse, there are errors not only in new systems, but in systems which have been operational for years and have undergone extensive testing and use. Why does this happen? Do the United States Navy, the Euroepan Space Agency and Airbus just employ terribly bad programmers? Could it be that all software developers just lazy or incompetent? Maybe the fault does not lie entirely with us. To understand this, it is worthwhile trying to get a feel for the complexity of the problem.

<Slide 7 -- 9>

This is a picture of the Prawn Nebula. I think everyone here would agree that the physical universe is a extremely complex entity. A simple back-of-the-envelope calculation will yield an estimate of 10^80 atoms in the universe. Now say we have 10MB of computer memory. How many values can it hold? This number is several orders of magnitude greater than the number of atoms in the universe. The behaviour of a computer system is essentially the dynamic interaction between and evolution of values in memory. Although only one value exists in memory at any instant, the interaction of these values and how they change gives rise to a phenomenally complex system. I

hope this simple argument convinces you that the complexity of our software is on a cosmological scale, comparable to that of large biological or physical systems. When we deal with large mechanical systems, a theory like Newtonian mechanics allows us to abstract away from atomic details. Software does not always afford us this luxury because, as we have seen, individual values of bits may actually make the difference between a successful flight and an explosion.

Thomas Henzinger, a senior researcher in our area, put it quite well in his inaugral Lecture in Lausanne in 2006:

"Software truly is the most complex artefact we build routinely.
It is not surprising we rarely get it right."

So where does that leave us? Do we accept that software will be buggy, rockets will crash and blue screens shall inherit the earth? Software systems are not the only complex systems we have built to date. How did we manage with suspension bridges, railway networks, and in some cases, space shuttles? Architects of such systems extensively use applied mathematics, engineeing physics, and modelling and simulation tools.

These are exactly the kinds of tools we need as well and a most research in programming languages and software verification is about making such technology a reality. Software verification is the *quest* to develop the mathematics, the analysis methods and the tools to design and deploy among the most complex systems we have engineered to date. If I had to sum up the question we try to answer, I could put it quite dramatically in the words of the American Physicist, Heinz Pagels.

"Could the God that plays dice, trigger a nuclear holocaust by a random error in a military computer?"

To put it less dramatically, we ask:

Can we build a machine that can reason about programs and prove mathematical facts about them?

This is the part where you groan inside because the fun is over and we are now getting to the serious part. We should ask if this question even makes sense.
What does it mean to prove mathematical facts about a program? How can a machine reason about anything, let alone programs? I hope I can answer these questions in the rest of the talk.

Chapter 1. Love's Labour Lost.

The first efforts to reason about programs were actually lost and had to be rediscovered, hence the title. The story begins with no less a personality than John von Neumann.

<Slides 11 -- 14>

John von Neumann was among the foremost polymaths of the last century. By the age of eight, he was fluent in classical greek. At the age of six, he could *divide,* 8 digit numbers in his head. Not add, subtract or multiply, but *divide!* When his parents has guests visiting, as a game, the guests would be allowed to pick a random page of a phone book and show it to von Neumann. He would then

memorize it and answer any question about who lives on which street and what is the number of Mr. Darvas etc. His precocity would develop well beyond such mental gymnastics. In his early twenties, he was already a celebrity in the mathematical community. One anecdote goes that researchers would say:

"Most mathematicians prove what they can, von Neumann proves what he wants."

He would go on to establish the mathematical foundations of quantum mechanics, game theory and make fundamental contributions to set theory, measure theory and several areas of mathematics, physics, economics and computer science.

In the early 1930's he moved to the United States because he felt he had better prospects there. During the World War II, he was a consultant to the United States government and military. In particular, he was involved in the Manhattan Project, then a top secret effort to develop the first atomic weapon. Here you can see him with Stanislaw Ulam and Richard Feynman who were also involved in the Manhattan project. This project involved several detailed calculations which could take a long time to complete. One story goes that the engineers were trying to evaluate a complex formula without any success. They ask von Neumann if he could help them find an approximate solution. He seemed quite puzzled and could not understand why they wanted to approximate the solution. They explained that they had all tried and failed to calculate the value. He replied that he did not know the approximate value and had no idea how to calculate an approximation, but if they wanted he could tell them the exact solution. And he did all of that in his head!

One day, he had gone to visit the ballistics research lab in Aberdeen Maryland, when he met a man called Herman Goldstine on the railway platform. Herman Goldstine was a mathematician. His PhD supervisor worked as a ballistics expert during World War I and got him to join the Ballistics Research Laboratory during World War II. The main task of this lab was to produce firing tables. These tables contained information about how to direct guns to precisely hit distant targets. The 'calculator' they used was around one hundred women in Aberdeen who worked these numbers out by hand. Goldstine was convinced that this was not a good mode of operation. He managed to get funding for building an electro-mechanical calculator called the ENIAC. It was shortly thereafter that he met von Neumann on a railway platform. He began talking about this amazing machine his team had built which could do 300 multiplications a second. And then, as he said in his biography,

"*... the whole atmosphere of our conversation changed from one of relaxed good humour to one more like the oral examination for the doctor's degree in mathematics*"

von Neumann collaborated with Goldstine's team on new project to build another computer. Von Neumann and Goldstine typed a hundred page report on this, which Goldstine then circulated. The report listed von Neumann as the only author and the kind of system it proposed is today known as the 'von Neumann architecture.' None of the other members of the project were mentioned, which caused a huge rift in the group leading to the resignation of Eckert and Mauchly, two primary members of the project. After the war von Neumann and Goldstine joined the Institute for Advanced Study in Princeton and wrote a few reports about designing and programming computers.

<*Slides 15* -- 18>

The reports were titled "Planning and Coding of Problems for an Electronic Computing Instrument." I was completely amazed when I looked at these reports. They covered everything from the architecture of the machine to programming methodology and considerations such as bit-vector

arithmetic. Here's is what they thought of programming:

"Since coding is not a static process of translation, but rather the technique of providing a dynamic background to control the automatic evolution of a meaning, it has to be viewed as a logical problem and one that represents a new branch of formal logics. We propose to show in the course of this report how this task is mastered."

It was clear to them, way back in 1947 that programming is intimately connected to the activity of logical reasoning. For curiosity value, here are some program outlines they sketched. They point out:

"It is clear that this notation is incomplete and unsatisfactory. Figures 7.1 d-f fail to indicate how control gets into these loops, how many times it circles each loop, and how it leaves it."

They called such a structure a flow diagram. A flow diagram is then augmented with three kinds of boxes: operation boxes in which you modify a value in memory, substitution boxes, which we can think of as an assignment to a variable, and assertion boxes which are like assertions today. Imagine that, they didn't have compilers, or programming languages as we know them today, but already, in 1947, they thought of assertions! Here is a program with all the three.

What is an assertion in a flow diagram? According to them,

"An assertion box contains one or more relations. ... An assertion box never requires that any specific calculations be made, it indicates only that certain relations are automatically fulfilled whenever [the code] gets to the region which it occupies."

Here is a more detailed description that I will not read out, but what it in effect says is that if the truth of an assertion is not blatantly clear, one must provide a mathematical proof.
There are two observations to make here: First, to von Neumann and Goldstine, who were essentially discovering the fundamentals of programming, there was no distinction between writing a program and proving that it was correct. This is not surprising if we consider their background. They designed computers to calculate missile trajectories and bomb payloads and computers those days were huge and slow and took time to program. If you had an error the program had to be rewritten and rerun. It is not at all surprising that they would not consider writing a program unless it was, in their own words, *"completely obvious"* that the program did what it was supposed to do. The second observation is that they believed writing assertions in a flow diagram was the right way to achieve correctness. This idea would occur to several other people over the next 20 years.

One such person was Alan Turing.

<Slide 19 -- 21>

Alan Turing's story has been sung by many bards; founding father of theoretical computer science, hero of the second world war, martyr for homosexuals, a symbol of repressive government policies and visionary for artificial intelligence. I will skip over his career to 1949, when he presented a 3-page paper titled "Checking a Large Routine" at a conference in Cambridge. I think this paper very nicely sums up the mathematical intuition behind methods in software verification.

Suppose I show you some numbers and you tell me their sum. The only way in which I can check if your answer is correct is to add up the numbers myself and see if our answers match. But, if you

show me your calculations with all the subtotals, all I have to do is check that the totals and carries in each column are correct. I can even do the checks in any order, so my task will actually be simpler than adding the numbers again. The genius of Turing is that he realised that this principle applies to *any computation*. If you multiply numbers, you can work out intermediate products, if you do long division, you have a sequence of calculations. But even if you sort a list or words, or search for a node in a graph there exists an equivalent of a subtotal. To check that a computation is correct, you only need to check these generalised subtotals.

Here is Turing's proof of correctness of a program computing factorials. Like von Neumann and Goldstine, he has annotated the program with assertions. When the program starts, we have that n is a positive integer and when the program exits, the output is n-factorial. Another great insight Turing had was that we can check these assertions independently and in any order. He also realised that one mush show a program terminates.

<Slide 22>

Sadly, these ideas did not receive much attention and slowly faded from the collective scientific consciousness. In the 20 years that followed, hardware technology evolved significantly and while the notions of machine code, assembler and programming languages were being developed, correctness was forgotten. Donald Knuth recalls that period for us in his tribute to Bob Floyd:

"People would write code and make test runs, then find bugs and make patches, then find more bugs and make more patches, and so on until not being able to discover any further errors, yet always living in dread for fear that a new case would turn up on the next day and lead to a new type of failure. We never realized that there might be a way to construct a rigorous proof of validity; at least, I'm sure that such thoughts never crossed my own mind when I was writing programs, even though I was doing nothing but proofs when I was in a classroom. I considered programming to be an entirely different category of human activity. The early treatises of Goldstine and von Neumann, which provided a glimpse of mathematical program development, had long been forgotten."

Chapter 2. Taming of the Shrew

Though program verification had been forgotten and programming had become routine activity by then, a new revolution was brewing which should reignite work on program correctness. It was called Artificial Intelligence.

<Slide 24, 25>

In the summer of 1955, four young mathematicians authored a proposal titled:

"A proposal for the Dartmouth summer research project on Artificial Intelligence."

The authors were John McCarthy, and Marvin Minsky whom you can see in this picture, Nathaniel Rochester from IBM and Claude Shannon in from Bell Labs. Their proposals gave a lot of momentum to the new area of artificial intelligence. It was a time of heady optimism that anything was possible. The people involved in this research had written programs which could play games like chess, hold a conversations and prove mathematical identities -- all accomplishments that completely amazed outsiders to the area. John McCarthy as you can see from this statement was convinced that logic

would be important in this effort.

"It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance."

-- A Basis for a Mathematical Theory for Computation, 1963

He also wrote a paper that clearly restated the problem of verification and generated interest in the problem.

"Primarily, we would like to be able to prove that given procedures solve given problems …Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program. For this to be possible, formal systems are required in which it is easy to write proofs."

-- Towards a Mathematical Science of Computation, Proc. of Information Processing 1962

<Slide 26 -- 28>

This paper caught the attention of a young man named Robert Floyd. Floyd received a bachelors degree in liberal arts at the age of 17 and then got another degree in physics. He was self-taught in computer science, and learnt programming to support himself while studying physics. By the age of 27, even though he had no PhD, he had written quite a few important papers in computer science and was appointed an assistant professor at Carnegie Mellon University. He had a few quirks too. He loved his BMW, which he nicknamed Tarzan, and wanted another one so that both cars could have bumper stickers saying '*My other car is a BMW.*'

Floyd and Donald Knuth established an active scientific correspondence. They would eventually become friends and Floyd painstakingly reviewed the first volumes of The Art of Computer Programming. When Knuth applied to Stanford, he wrote a letter to Floyd saying:

"Bob, I have the feeling that this is going to be a somewhat extraordinary letter. During the last year or so I have been getting lots of offers of employment from other colleges. I think I told you that I plan (and have planned for a long time) to decide on a permanent home where I will want to live the rest of my life... Due to the present supply and demand in computer science, I am fortunate enough to be able to pick just about any place I want to go; ... I believe the four places that are now uppermost in my mind are Stanford, Cornell, Harvard, and Caltech (in that order)... It occurs to me that I would very much like to be located at the same place you are, if this is feasible;"

Floyd replied saying *"I'd say if you want to make the move, I don't have any plans that would conflict, and I will be very tempted to go to Stanford myself; I probably will go, in fact."*

Floyd's experience with programming convinced him that one needed a method more fool-proof than debuugging. His work on this topic led to the paper 'Assigning Meaning to Programs,' which is usually cited as the origin of program correctness work today. Let me mention another piece of trivia. Bob Floyd added "W" to his name as a middle name. In the paper he wrote his name as Robert W. Floyd. When he was asked why he wrote it with a dot, he replied that the correct abbreviation of 'W' is 'W.'.

We've seen the ideas of Turing and Goldstine and von Neumann on this subject. There were a few other people having similar ideas at the time, like Peter Naur. They all essentially concluded that you needed to write conditions that looked like assertions in your code if you wanted to do a proof. What they did not tell you was what these assertions should look like. Could one plug arbitrary logical formulae in the assertions? If I told you what a correct program was supposed to do, could you identify which assertions to write in the code? Floyd's paper addressed these issues as he himself states:

"To prevent an interpretation from being taken arbitrarily, a condition is imposed on each command of the program."

Remember the analogy to sub-totals from Turing? Turing told us that we needed to find the sub-totals for an arbitrary calculation but gave us no idea how to do that. What Floyd did was characterise what the equivalent of 'subtotals' for arbitrary programs looked like. Meaning, if you found some assertions that satisfied these conditions, you were guaranteed to have a proof.

Let's take an example from Floyd himself. This will be the most technical slide I have. This is a program to compute the sum of n numbers. What Floyd suggested is that you propagate information along a control flow graph as logical formulae. In the beginning we only know that n is an integer. After the assignment to i, we know nothing more about n except that it is an integer but we also know that i is 1. But what happens when we enter a loop? If we just label a loaction with a fact, then we have to scratch it out and rewrite the labelling every time we visit the location in a loop. Floyd realised that you need something called a *loop invariant* that is true *every time* you go through the loop. So, if I have gone through the loop k times, I have the sum of the first k numbers. This guarantees that if I go through the loop n times, I will obtain the sum of the n numbers.

Another appealing feature of Floyd's method was that some assertions could be obtained mechanically by looking at the program. The only assertions that have to be invented are loop invariants. Remember that our final goal is to build a machine that can do such reasoning for us. If some part of it can be automated, it will make our task easier. In fact, most papers on verification written in the 4 decades since Floyd's paper have been about methods for discovering loop invariants.

Before Floyd's paper, it was completely unclear how to go about proving correctness of a program. After Floyd, it was hard to imagine any other way. It was the first point in the correctness quest where people looked at a method and said: Of course! Why didn't I see that before! One of these people was the British computer scientist and pioneer Tony Hoare.

<Slide 29 -- 32>

Tony Hoare was born in Colombo where his father was in the Civil Services and his mother was the daughter of a plantation owner. He grew up in Ceylon till 1945 and then returned to England. He studied Classics in Oxford learning ancient Greek, Latin, literature and philosophy among other things. His service in the navy followed, where he had to study military Russian. After his service, he studied language translation and did an exchange year at Moscow State University. He was paid to write a translation system between English and Russian. The electronic dictionaries were stored in sorted form on a long magnetic tape. It was efficient to look up the words in a sentence in alphabetical order. In pondering on how he could do this, he developed the Quicksort algorithm that most of you are familiar with. Afterwards, he joined a company called Eliott Brothers and was part of a project to implement an Algol compiler. The language definition for Algol had been made very

precise by the Danish engineer Peter Naur, using a notation we now call Backus-Naur Form. Hoare felt that the semantics of the language were quite unclear. He decided that this was the problem he had to solve.

How can one describe what a program is supposed to do without specifying compiler specific and architectural details? Tony Hoare believed that mathematical logic, which he had studied in philosophy, was the right medium. His objectives were clear but a method was missing. In 1968 he began working as a professor at the Queen's University in Belfast to pursue this question when Peter Lucas gave him an preprint of Floyd's paper. There, Hoare found the conditions he needed for his logical setting. To his credit, Hoare made a lot of changes to the logical presentation of the material. His excellent exposition skills led to this becoming one of the most cited papers in computer science. Here is the first page from his beautifully handwritten manuscript:

"In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs."

One drawback in Floyd's method is that it was monolithic. You had to start from the entry point of a program and walk through it, using loop invariants to jump through the loops. Hoare instead tried to treat reasoning about programs as applying laws of inference. A common rule of inference from classical logic is *modus ponens*. If a statement A is true, and A implies B, we can conclude that B is true. Hoare's paper showed us that one could reason about programs in the same way.

Unlike Floyd's method, what Hoare logic does is treat a program like a jigsaw puzzle. You can prove something about each individual statement in the program, like assignments and values of Boolean expressions etc. Then one can use the rules for structuring constructs to put these pieces together in if-then-else constructs and while-loops. Putting everything together gives you a proof of correctness. Amazingly, this connects back to Turing's observation that one should be able to check each assertion separately. You can see some typical rules summarised here in Prof. Patrick Cousot's slides.

Where does that leave us? We started out to build a machine that could reason about programs. With Hoare's paper, we finally had a way for human beings to reason about programs. To put it flippantly, all we have to do is find assertions satisfying some conditions. But there is a catch. Who is going to come up with the assertions? There were several other questions that were asked and answered in the next few chapters in this history. I will skip them and jump ahead to a separate thread in this story.

Chapter 5. Measure for Measure.

<Slide 33 -- 35>

The researchers who blazed the trail in the story so far used logic as a tool, but were largely driven to this work by their experience building systems. What about the logicians? They were not far behind. Infact, the founding figures in several areas of computer science started out with a background in mathematical logic. Around the time that John McCarthy and Marvin Minsky were organising their AI conference, Alfred Tarski, Alonzo Church and several other eminent logicians in

the United States got together in Cornell University in the summer of 1957 for the Summer Institute on Symbolic Logic.

I apologise for the name-dropping I am about to do, but this is meant to be a geek-out for the logicians. This meeting was attended by 87 logicians and Anil Nerode remarked that no such event has taken place in logic ever since. There were two large families of logicians there. The students of Alonzo Church from the East Coast and the Tarski school from the West coast. Church talked about electronic circuit synthesis from logic, an idea that had been developed by Claude Shannon in his Masters thesis. Abraham Robinson, who would a few years later change the face of automated deduction with his paper on resolution talked about automated theorem proving. Michael Rabin and Dana Scott presented finite automata, for which they would later receive the Turing Award. Martin Davis talked about his experience with implementing Presburger arithmetic. He would later play a very important role, along with Hilary Putnam, Julia Robinson and Yuri Matiyasevich in showing that Hilbert's 10th problem was undecidable. It completely blows my mind to imagine that they were all together, in the same location, talking about all this amazing work! Apart from the intellectual explosion, there was a lot of competition and personal conflict on display, not to mention scandal.

The most highly regarded logician of the time was Goedel, who was a recluse and declined to attend. Alfred Tarski, who was quite egocentric, was happy enough to fill that role. He even remarked on ocassion that he was the greatest sane logician alive, taking a dig at Goedel.

Alfred Tarski was born in Warsaw in modern day Poland, which was then a part of the Russian empire. Warsaw underwent a lot of political turmoil while he was growing up. It was annexed by Germany and Austria-Hungary during the first World War and in 1916 Poland declared its independence. For various reasons including spiritual, a need for a more Polish than a Jewish identity and the growing anti-semitism in the region, Alfred and his brother became Roman Catholic and changed their names to Tarksi. One story is that he was presenting a paper at a conference when someone in the audience interjected *"Your result has already been proved by Teitelbaum."* Tarski glowered, *"I am Teitelbaum!"*

In August 1939, Tarski was attending a conference in Havard University when Hitler ordered the invasion of Poland. It was clear to Tarski that despite a change of name and religion, he would not survive very long if he returned to Poland. He applied for asylum in the United Status and after a long period of instability, he finally obtained a position in the University of California Berkeley. He was initially disappointed because it was quite far from the East coast where most of the European intelligentsia had agglomerated, especially the stimulating environment for studying logic in Princeton. Tarski would eventually build a very strong school of logic in Berkeley.

In his long and productive career, Tarski had several important results such as the Banach-Tarski paradox and the decidability of the theory of real arithmetic with addition and multiplication. Among logicians, he is regarded by some as one of the greatest four. There was Aristotle whose principles of reasoning stood unquestioned for nearly 2000 years, Gottlob Frege who gave us the formalisation of first order logic as we know it today, Kurt Goedel who showed that powerful arithmetic systems contain statements whose truth we cannot prove and finally Tarski, who defined the notion of truth in formal systems and showed that the true statements in a formal system cannot be defined within that system.

<Slide 36, 37>

The result of Tarski that is most used in program verification is his fixed point theorem. To discuss

this, we have to first understand what a fixed point is. Fixed points occur in several areas of mathematics and have numerous applications. Mathematically speaking, a fixed point of a function f is a point x where $f(x) = x$. There is a simple puzzle to illustrate this.

This is a region of central Switzerland. Imagine a pilgrim starts from the bottom of the hill at 9AM and walks up reaching the top at 5PM. The pilgrim camps overnight and begins the descent *along the same path* at 9AM the next day, reaching the bottom at 5PM. Was the pilgrim ever at the same place at the same time on both days?

This puzzle can be modelled as a function from a set to itself. Each element of the set is a time-space coordinate and the function maps the position on the hill at a time t on the first day to the position at the same time on the next day. There is a theorem called Brouwer's fixed point theorem which states that this function has a fixed point. In other words, there is a position on the hill where we were at the same time on both days. If you want an intuitive argument for the solution, imagine you superimposed the paths of the two days starting the pilgrim off at the same time from the top and the bottom. The two pilgrims would meet at some point. This is your fixed point.

Not all fixed point theorems have such intuitive physical realizations but the basic content is often similar. When mathematicians present a fixed point theorem, they usually take a class of functions that operate on some structure and find conditions under which those functions have fixed points. Tarski studied a structure called a complete lattice. He gave a sufficient condition for a function on a complete lattice to have a fixed point. The proof is very elegant, but I will not go into it here.

Fixed points can be quite dizzying to study, as this little gem from the logician Augustus De Morgan illustrates:

"Great fleas have little fleas
upon their backs to bite 'em,
And little fleas have lesser fleas,
and so ad infinitum.

And the great fleas themselves, in turn,
have greater fleas to go on,
While these again have greater still,
and greater still, and so on."

One person who had an excellent grasp of such concepts was Dana Scott.

<Slide 38, 39>

Scott was an undergrad in Berkeley and worked in the library to earn some money. He was a very special kind of undergrad, who decided to randomly browse through the pages of the Journal of Symbolic Logic. There he came across a paper by one of his professors and though *"I should attend his course"*. That professor in turn mentioned Tarski which led Scott to attend Tarski's course. Tarski had very high standards and conducted clear but intense lectures. During his set theory course, the story goes that he would write a statement on the board and then ask *"Now how shall we prove this?"* Richard Montague, who was his PhD student and clamoured for his approval and praise would flail his arms to show he had a solution before Tarski had finished the question. Scott then asked calmly from the back of the class, *"Do you want it with or without the Axiom of Choice?"* meaning he already had *two* proofs in his head! Tarski himself was not to be outdone and once

retorted *"I first proved this when I was thirteen."*

It was clear to everyone that Scott was a prodigy and he was soon included in the group of graduate students and eventually began his PhD with Tarski. There is a slight side story here. During his time in Poland, Tarski had published quite a bit in German. There was an English Biologist called Joseph Woodger, who it it seem was a bit of a logic groupie! I didn't know such things existed. He tried to translate Tarski's work from German to English. But his German was not very good, and his mathematical training was not very good, so as you can imagine, the translation was not very good. This annoyed Tarski and he wanted to have good translations done. He asked Montague and Scott to do it, but half-way through the project, Montague graduated and Scott had to complete the translations alone. Now Scott wanted to go on with his work and had not signed up for a PhD to do German-English translation. But Tarski had a strong overbearing personality and did not take no for an answer, so Scott did not know how to communicate his situation. Eventually Tarski got fed up and fired Scott, behind his back, hiring a new PhD student. This hurt Scott who left Berkeley and went to Princeton where he was welcomed with open arms by Alonzo Church. Tarski and Scott eventually reconciled and Tarski liked to call Scott *"my student".*

Princeton, and Church afforded a significantly different environment from Berkeley and Tarski. For one, unlike the tempermental and exacting Tarski, Church was a very methodical person, as this reminiscence from Rosser and Kleene shows us.

"There was a story that went the rounds. If Church said it's obvious, then everybody saw it a half hour ago. If Weyl says it's obvious, von Neumann can prove it. If Lefschetz says it's obvious, it's false."

-- Barkley Rosser and Stephen Cole Kleene

However, even if Church's style may not have been what Scott was used to, the environment was fertile and he met several logicians there, including Michael Rabin with whom he would do seminal work on finite automata. When he was done with his PhD, Scott he moved back to Berkeley as an assistant professor. Once at a conference, Scott met a British computer scientist called Christoph Strachey.

<Slide 40 -- 42>

Christopher Strachey had a similar background to Bob Floyd and Tony Hoare. He had worked on several programming projects including the design of CPL, a language called the Combined Programming Language, developed in Cambridge. CPL was not very successful and its failure is said to have bothered him a lot. He often had strong opinions and in the design of the language believed that the conditional construct should be called *if-then-otherwise* because *if-then-else* was grammatically incorrect. Thankfully the other language designers won. CPL was believed to be too complicated and would be simplified to BCPL. A minimal subset of BCPL was implemented by Ken Thompson in Bell Labs and called B. With further changes by Thompson and Dennis Ritchie, we ended up with C.

A few years ago, some students in MIT wrote an automatic paper generator. Half a century earlier, during his programming job, Strachey was often bored and wrote a program to automatically generate Victorian love letters and even managed to publish one in a magazine.

Later in his career, Strachey, like Hoare, was bothered by the problem of giving semantics to programming languages. Unlike Hoare, he believed that mathematical functions were sufficient. For

example, the plus sign in a programming language could be formalised by addition. The *and* and *or* operators could be replaced by logical conjunction and disjunction. With some work, you could also formalise branching constructs. More complicated control flow was challenging. The impenetrable problem seemed to be how one could model a loop and recursive procedures.
Strachey got Scott interested in the problem and Scott, with his rigorous mathematical and logical training, realised that while loops and recursion could be described using fixed points of functions. Essentially, a loop represents some behaviour that is repeated until the result does not change anymore. That's just like a fixed point! The resulting theory differed significantly from that of Floyd and Hoare and is one of the main achievements of theoretical computer science at that time. We call it the Denotational Semantics of programs.

The interesting part about fixed points is that one can calculate them. There is a result from Church's student, Stephen Kleene, saying that if you have a monotone function on a complete lattice, then the least fixed point can be obtained by by an iterative procedure. That's very nice. But what does this have to do with reasoning about programs? Remember that Hoare logic gave *humans* a ways to reason about programs but it required inventing loop invariants and such conditions. Denotational semantics is in my opinion less user-friendly because functions and fixed points are more intimidating. The connection between the two was provided by a PhD student from Cornell university called Edmund Clarke.

<Slide 43,44>

The difficulty with automating the Hoare-style approach was that one had to invent invariants. The approach of Scott and Strachey was interesting because it used functions and could be viewed as calculations. The title of Ed Clarke's paper will explain everything.

He showed that invariants of programs could be described as fixed points of functions on complete lattices! The second sentence of his abstract explains it all:

"We give a characterization of program invariants as fixed points of functionals which may be obtained in a natural manner from the text of a program."

This was great because it reduced the problem of computing invariants to that of computing fixed points. Thanks to the Kleene theorem, obtaining a fixed point amounts to a straightforward calculation. Machines are not good at inventing formulae and guessing invariants for a proof but they are definitely good at calculation. Clarke's result essentially told us: don't worry about inventing; just calculate.

It would seem at this point that we had achieved our goal, at least in theory. We needed a way to reason about programs, Floyd and Hoare had done that. We needed a way to calculate properties about programs, Scott and Strachey had done that. Finally, to build a machine, Clarke provided the crucial bridge by showing that the process of reasoning could be viewed as a calculation. It would appear that we have a fairy-tale ending. Can it really be?

Chapter 7. The Tempest.

<Slide 46>

While all this exciting progress was taking place, there was an umbrella of doom hanging over the

entire enterprise. A theorem proved by Henry Gordon Rice in his PhD well before much of the work I have described said that there could exist no mechanical device for proving program correctness. The proof is based on a reduction to the *Halting Problem.* Imagine you want to know if a variable *x* has the value 0 when a program exits. If all your program does is assign a value to *x*, then you know it will be 0 at the end if the value is 0. But one could easily put a large piece of code just before this assignment that does something very complicated. For example, the code could search for a counterexample to some conjecture in number theory. To conclude if the assignment takes place, the machine doing the reasoning now has to discover if the conjecture is true. Rice's theorem said: no matter what you do, you will eventually crash into a stone wall.

<Slide 47>

Undecidability was not the only problem. Ironically, Edmund Clarke, the same person who provided the crucial link to make these efforts feasible, proved that Hoare logic was in fact not the best fit for reasoning about programs in sophisticated programming languages. He showed in his PhD work that there are programming language constructs, such as certain kinds of recursive procedures, for which good Hoare-style rules do not exist. This threw open the question about how to reason about these complicated programs.

<Slide 48 -- 51>

If these setbacks were not enough, there was a paper written by Richard De Millo and Richard Lipton, both prominent researchers, and Alan Perlis the first recipient of the Turing award, questioning the basic premise of program verification. The provocative nature of the article is clear from the first sentence of the abstract:

"It is argued that formal verification of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics."

This created a lot of controversy. If you look at the paper, the authors are not actually ruling out the possibility or the utility of formal verification. They are only saying that proofs of programs have a very different place in the mathematical universe from proofs of theorems. But the paper is rife with antagonistic statements like

".. the proofs of even simple programs run into dozens of pages. ... The sad fact is that the kind of mathematics that goes on in proofs of correctness is not very good."

It definitely generated a lot of dialogue. Everyone from academics, to developers and even philosophers had something to say on the topic. There were several rebuttals written that argued against what they claimed but also did a lot of name-calling and silly nitpicking. I have quoted only a few excerpts for your amusement. It generated both positive and negative press for the area in general, annoyed a lot of people and confused some others.

Maybe all these setbacks were not a bad thing. To take some words of wisdom from the Danish scientist and poet Piet Hein, this probably tells us something good about the problem. I will now race through a few big ideas the field developed to cope with these setbacks.

Chapter 8,9,10. All's that Ends Well.

<Slide 53,54>

Rice's theorem and the other negative results are typical of the kind we have in computer science. They usually say that solving a problem in full generality is hard. However, by identifying useful restrictions, we can often get very far. This was the approach taken by Ed Clarke, who was by then a prof in Havard university and his PhD student Allen Emerson. The idea simultaneously occurred to Joseph Sifakis, a researcher in France and his PhD student Jean-Pierre Queille. Clarke, Emerson and Sifakis received the Turing award last year for this work, which is known as Model Checking. Instead of a full-blown programming language, they considered checking properties of finite state systems. This model can express design-level details of communication protocols and various mechanisms on a chip. A failure in such systems can have dire consequences, so their work was well appreciated.

I'll use a puzzle to illustrate the kind of reasoning one can do with a model checker.
Imagine there is a river and a wolf, a goat and a cabbage are to be taken to the other side by a human being. The person has a boat which can accomodate the human being and two more items. If the person is not around, the wolf will eat the goat and the goat will in turn try to eat the cabbage. The question is whether the person can convey all three to the other river bank. An algorithm for solving such a puzzle can proceed the way a chess playing algorithm is designed. One can construct a graph in which we explore all possible moves that can be made. If the wolf is left alone with the goat or the goat with the cabbage on either bank, we need to abandon the solution and explore another sequence of moves. This process can be highly optimised by considering different search heuristics, different representations for the graph and different kinds of objectives.

What does solving puzzles about wolves and goats have to do with program verification? That puzzle, and many others like the missionaries and cannibals, Sokoban and even something as simple as a maze are essentially constraint systems. They tell us a set of moves we can make, an objective we must satisfy and some dangerous situations we must avoid. The setting is similar to the design of sub-systems of a computer. If several programs that need to access the printer and the operating system has to basically solve the puzzle of giving every program access without allowing them to clash or wait indefinitely. The problems of designing a cache coherence protocol in hardware and various telecommunication protocols have a similar flavour.

<Slide 55,56>

Model checking overcomes the obstacles of Rice's theorem by focusing on a restricted space of programs. Another approach was adopted by a PhD student in Paris called Patrick Cousot. Cousot, in his PhD thesis proposed a method called abstract interpretation. Instead of restricting the solution space, Cousot said, let's analyse all problems but give approximate solutions. He worked within the setting of a fixed point theory and formalised what it means to find an approximate solution using a Galois Connection. There were several other results, but I will not go into them here.

The idea behind Cousot's approximation is that analysing a program is difficult because a computation is a complex object. If we can abstract away from irrelevant details of the computation, it may be easier to answer some questions. People use such intuition everyday to analyse problems. Here is an exotic example. Several people have pondered over the question of whether the Mona Lisa smiles and what it is about her smile that is so elusive. Margaret Livingstone, a neurobiologist in Havard university applied some filters to a image of the Mona Lisa and said, well if you pass the image through a low spatial frequency filter, the smile is more accentuated. Our peripheral vision is

sensitive to low spatial frequencies and maybe that is why we have the feeling that the Mona Lisa smiles only when we're not looking at her. That is the essence of the approximate method of analysing programs. If you lose a lot of detail, it may be easier to find the solution you are looking for. Of course, you may have lost some crucial information in which case the answer produced will be imprecise.

Epilogue. As You Like It.

That brings me to the end of this account of the early developments in my research area. Let me conclude by briefly summarising where we stand today. A lot of recent work has focused on how one can combine the strengths of the two areas of model checking and abstract interpretation. This has been fuelled by developments in another area I have not had time to talk about called automated theorem proving. That area focuses on building engines to reason in restricted mathematical theories. Earlier this year, I interned in Microsoft Research in India. They organised a summer school and some attendees made this poster. Today a researcher or practitioner has a range of tools and techniques to choose from. We are studying the interplay of these different ideas and taking steps towards making a powerful and useful program verifier a reality.

Thank you.