CS6180: Introduction to Constructive Type Theory

Robert Constable

August 29, 2017

1 Course Summary and List of Topics

These notes briefly summarize the content and course mechanics for fall 2017 CS6180. Constructive type theory is currently "hot," so this brief introduction highlights only a few reasons why type theory is important in contemporary computer science. In the course we explore several other reasons, and we will speculate on the future of the subject and its ties to related topics in programming languages, AI, theory, systems, and information science.

1.1 course requirements

There will be lecture notes and readings covering nearly all of the material. Students can participate at various levels. Those who simply want to know type theory at a high level and hear about its potential and most interesting results can meet the requirements by writing a paper on the subject and submitting three of the suggested exercises. In addition all students can help by *proof reading* and making small extensions for one or two of the lecture notes among the 20 to 25 that will be posted. Participation at that level will see grades in the A to A- range.

Students who are more interested can help in the exposition by writing tutorial notes on some of the conceptually most difficult material such as Brouwer's Bar Induction principle or his Fan Theorem or about the need for free choice sequences and their value in computer science.

Students willing invest more and aspiring to an A+ grade and possibly a publishable article can explore one of the *several research questions that will come up regularly.* For example, type theory is not well developed for proving *computational complexity results.* On the other hand there have been suggestions for how to formalize complexity arguments which would make a good start and be the basis for a possibly publishable result (ppr). A fresh look at the issues would definitely merit a project paper. There are some new issues arising from the fact that the type theory we will study implements Brouwer's notion of *free choice sequences.* There should be opportunities for new results about these interesting constructive objects. Some questions along these lines will be proposed.

Other suggestions are mentioned later in these introductory notes.

1.2 proof assistants

Constructive type theory is "implemented" in *proof assistants.* The course will briefly discuss and compare proof assistants. These implementations "bring type theories to life" and enable users to apply them to designing and coding algorithms and software systems and proving that they have certain properties. Knowing properties of algorithms and systems helps us understand their capabilities and limitations. We can compare those system properties to what we *intend* to do. As a very simple example, if we write an algorithm to recognize whether a given natural number is prime or composite, we expect to be able to prove based on the code that when the program is given a natural number, it will halt with a Boolean value telling us whether the input is prime or not. The code might produce factors if the number is not prime, depending on how we specify the task. Given that we have a very precise mathematical account of this task, we expect to be able to prove that the code does the right thing even if the argument is mathematically complex.

1.3 research questions

One of the research themes we will investigate is how to use *synthetic geometry*, as in Euclid, to develop algorithms in *computational geometry* that are provably correct and efficient. A major effort worthy of summer support is to create a "compiler" from synthetic algorithms to analytical algorithms (using the constructive real numbers). Dr. Bickford has a working example of this approach that he will lecture about. A related topic would be to apply a result from homotopy theory to a practical geometric algorithm. The PRL group researcher Ariel Kellison is working on developing Nuprl material on Euclidean geometry to establish outreach to Ithaca High School. Many interesting research questions have come up in this project, and she and I could supervise a project in this area.

In some cases we expect to know the cost of executing the algorithm. We will see why it is challenging to prove *computational complexity bounds* on algorithms in type theory. There have been PhD theses using Nuprl on this topic [20, 21, 13]. Since that work, we have considered other approaches worth exploring. They will come up during the course. The more complex an algorithm, the more we require ways to guarantee that it does the right thing. The importance of correctness is sometimes made by reporting on the consequences of trusting faulty algorithms, e.g. the Intel bug in a chip for arithmetic. There are books built around such stories [79, 90]. *Type checking* algorithms for programming languages with rich type systems help us understand programs. Model checking techniques systematically search for errors using a *model* of the task. Proof assistants work by helping designers create a formal proof that code has certain properties that can be precisely expressed in the prover's *specification language.* Typically these proofs provide much stronger guarantees than the type checkers do. We will also very briefly discuss previous strong (misguided) criticism of proof assistants, as in the 1979 article by De Millo, Lipton, and Perlis [42].[1]

[1]This article is reminiscent of the attacks on AI by the Dreyfus brothers which now seem so woefully misguided

Proof assistants provide considerable help with getting the many details of a formal proof correct. A good proof assistant can even *extract algorithms* that express the *computational content* of a proof. In that case, we end up with *two results for the price of one.* First, we find an algorithm that solves a problem. Second, we create a detailed formally checked explanation of why the algorithm is correct. One reason to implement a very expressive type theory is to provide the capability to precisely define a large variety of computing tasks. *It is impossible that we will ever capture fully and completely the capabilities of a complex software system.* There are limits that we know from Gödel's famous *incompleteness theorem* [53]. On the other hand, rich expressiveness is a very worthy goal, and it motivates providing a formal system that can express as much computer science as possible. That is one sense in which constructive type theory is appropriate. It is the richest relevant formal theory we know. This lends credibility to the idea that constructive type theory could be an appropriate *foundation* for computer science; we'll see that it touches essentially all of the subjects in modern computer science. We will examine this theme throughout the course, starting now at the very beginning.

Different versions of type theory are implemented in various proof assistants. We will mention three of them from time to time: Agda [25] from Sweden, Coq [22] from France, and Nuprl [37] from the US. Moreover, at least three new proof assistants are being built to implement variants of constructive type theory – JonPRL and RedPRL at CMU and Lean at Microsoft Research.[2] There are also proof assistants for *higher order logic* (HOL). One of them is called HOL [55, 60, 70, 69, 50, 32]. Although this is not a course on proof assistants, we will illustrate their role with Nuprl examples from time to time.

We already know that constructive type theory, like all sufficiently rich formal mathematical theories, cannot be *complete* in the sense that it is unable to prove all of its true statements. This is the consequence of Gödel's famous incompleteness theorem [53]. For type theory, we need to be clear about what "*intuitively true*" means. We will consider this question during the course.

1.4 constructive type theories

Why is *type theory* used in proof assistants? Could we use set theory instead, the foundational language of mathematics? The simplest and most naive answer is that we build software systems with programming languages, and they use types to specify tasks, not sets. Indeed, our work on type theory arose naturally from our efforts to design and implement a programming logic for the Cornell programming language PLC. Types were a key feature of this language, so we had to learn to formalize them. If our primary concern is the specification and correctness of algorithms and systems, type theory is the most appropriate choice. Some researchers see this as a mistake in that it complicates connections to a vast body of mathematics. Researchers with this concern in mind have shown how to express set theory in type theory. Peter Aczel has done very interesting work along these lines [4]. Furthermore, the practical value of expressing computational mathematics in

[45]. The views of George B. Dyson [47] are much more prescient. One thing he wrote in his book *Darwin among the Machines* is this famous line: "In the game of life and evolution, there are three players at the table: human beings, nature, and machines. I am firmly on the side of nature. But nature, I suspect, is on the side of machines".

[2] Lean seems to have another name as well, but not with "PRL" in it!

programming languages has led to incorporating most mathematical concepts within type theory. We will see many examples of this during the course.

For areas of mathematics that are concerned with constructions and algorithms, set theory was never "just right". Some constructive mathematicians started using notions such as "species" and others used the term "type" still others used "constructive sets" and so forth. This community is principally concerned with algorithms and constructions. It has a very long tradition, going back to Euclid and his ruler and compass constructions. Computation has been a component of numerical mathematics from at least the 9-th century. For instance, the Arabic source, al-wrizm "the man of wrizm" (also known as wAb Jafar Muhammad ibn Msa) gave us the notion of an *algorithm*. Computational mathematics did not fit well in set theory right from the start, so there have been numerous independent efforts to find better ways to accommodate it [61]. They have led us to constructive type theory. We will look at some historical milestones along that path when they help explain an important concept.

We will study two versions of constructive type theory. One is simply called *Constructive Type Theory* (CTT), and one very precise version was created at Cornell, starting with a major NSF grant in 1980 and a preliminary article [14] shortly after. Our initial constructive type theory was described in the 1986 book *Implementing Mathematics with the Nuprl Proof Development System* [37]. Recently we have enriched this theory with concepts from Brouwer's intuitionistic mathematics [26, 74, 27, 62] namely with *free choice sequences*, *Bar Induction*, and the *Continuity Principle*. The right name for this enriched theory is probably Intuitionistic Type Theory (ITT). However, Per Martin-Löf wrote two articles about his much weaker notion of intuitionistic type theory [80, 81, 89]. So we don't yet have a definitive name for our new theory, and we have not "rewritten the book yet". This is the first CS course in which some of these new ideas are being taught. Martin-Löf was not imagining an implemented theory, and a number of the most novel ideas in CTT came from computer science rather than logic. We added many novel concepts and methods that did not appeal to all constructive logicians. This resulted in a type theory much richer than Martin-Löf type theory as we will see, but Agda and Coq basically use a 1998 "intensional" version of Martin-Löf type theory [102].

At Cornell, some core type theory concepts are taught in the graduate programming language (PL) course CS6110, and some of the most practical ideas show up in CS4110 and even in CS3110. These concepts are important in *formal methods* where *precise task specifications* and *program correctness proofs based on specifications* are fundamental working notions. However, these ideas are not presented as elements of a comprehensive theory of constructive mathematics in those courses but rather as an account of the types implemented in more standard programming languages such as the ML family of languages [86, 56, 59, 85, 77, 87] that includes OCaml.[3] It is easy to imagine that establishing *program correctness will require proofs*, and proofs will require a logic. So we see that elements of logic will be taught in this course. In particular *constructive logic* is an important component for reasons we will discuss in detail. That was not a feature of Edinburgh LCF even though the ML language being defined used *polymorphic types* that capture the essence of *constructive propositional logic*, as we will soon see.

[3]The book *Edinburgh LCF: a mechanized logic of computation* [56] was life changing for many people, including this author.

Constructive logic is natural for computer science students. The basic constructive logical operations show up in typed functional programming languages. In constructive logic the primitives have *computational meaning*. Cornell has a particular interest in this aspect of type theory because many of our PhD graduates have made fundamental contributions, especially to a well established part of the subject captured in the slogan "*proofs as programs*." This was the title of one of our early articles [15], and it caught on. The idea was sketched also in the article *Constructive Mathematics and Automatic Program Writers* [33].

Our "proofs as programs" article led to an entire book on this topic, "Adapting Proofs-as-Programs, the Curry-Howard Protocol," [91]. The closely related idea of "programs as proofs" came from research that Mike O'Donnell and I did writing the book *A Programming Logic* [39]. In that book, PLI programs annotated with *assertions* were treated as proofs [38]. We will see examples of these "programs as proofs" in this course. There will be numerous examples of proofs treated as programs. Perhaps many students realize just from the words "type theory" that the topic has a logical component, and proofs will be important.

Proofs are important in *computer security*, both in system design and the study of attacks and defenses against them. Security issues are especially important for *distributed systems*, and we have done a considerable amount of funded work on that topic at Cornell with Ken Birman and Robbert van Renesse [23, 24, 78, 103].

1.5 theory creation and exploration

One of the seldom advertised features of *computer assisted theory creation* is that it provides much more than its final product. It also creates the "theory DNA" that tells how to rebuild and extend the theory. Thus it is a kind of "living formal theory" that can be extended, reproduced, improved, and integrated into more comprehensive theories. This feature allows us to explore various versions of a theory, say by changing the definitions or axioms and automatically attempting to rebuild the theory. If nothing breaks, we have a new theory. Otherwise we see exact places where the rebuild fails and needs to be adjusted.

In the early stages of exploring a topic with the help of a proof assistant, it is common to modify definitions, create new tactics to help build proofs, improve the efficiency of algorithms, and so forth. A mature proof assistant provides considerable help with all of these tasks. The Nuprl proof assistant provides many of these features in its *Formal Digital Library* (FDL) and its *tactic language*. As far as we know, many of these features remain unique to Nuprl. We will illustrate how they are used. In particular, our theory of *Euclidean plane geometry* is based on a list of 27 axioms and ruler and compass constructors. Geometry proof tactics were discovered by experimenting with ideas from Euclid's original theory [49, 48] and the book by Schwabhäuser, Szmielew, and Tarski [104]. We also learned new ideas and results from Beeson's work [16, 17, 18, 19], from Hilbert's book [63], and from other sources [72, 43, 40, 29].

Summary: Constructive type theory (CTT) is a formal logical theory for precisely defining a

wide variety of computing tasks and theorem proving tasks, from simple ones like implementing the fundamental theorem of arithmetic to very challenging tasks such as synthesizing Byzantine fault tolerant distributed protocols or implementing the whole of Euclidean plane geometry using the new axioms and new formal proofs. CTT is sufficiently rich to define a *constructive set theory* [1, 2, 3, 4, 5], from which it is natural to define a classical set theory (based on virtual evidence [36]) and thus provide the standard foundation for mathematics. CTT is expressive enough to extend computation from Church/Turing computability to Brouwer's broader notion that adds *non-lawlike computability* based on free choice sequences. When that is done, we can define the *intuitionistic type theory*, and that is the current type theory implemented by Nuprl.

Exploring this intuitionistic type theory is a new topic of investigation despite the fact that Per Martin-Löf defined type theories with the name "intuitionistic type theory", [80, 82, 83, 84]. His theories do not include the most fundamental ideas from Brouwer's articles, namely *spreads*, *Bar Induction*, the *Continuity Principle*, and *free choice sequences*. We will explore these bold ideas of Brouwer near the end of the course; they have the potential to provide a broader and deeper foundational theory for computer science. In this course we cover the the core ideas of CTT as well as some aspects of their implementation in proof assistants. We also discuss the on-going evolution of proof assistants. That leads to a discussion of methods from AI that are appropriate.

Historical background on type theory: Type theory was proposed as a logical foundation for mathematics by Bertrand Russell [101]. He and Whitehead wrote the three volume set of books entitled *Principia Mathematica* [108]. They introduced a *hierarchy of types* classified by levels to avoid what is now called *impredicative* definitions. Type theory was adopted by N.G. de Bruijn in 1978 [41] to build his *Automath* system. He used the hierarchy idea to classify types and claimed that almost all mathematics can be done with only three levels of his universe hierarchy.[4] Alonzo Church also defined a simpler type theory [30] that he proposed as an adequate foundation for mathematics. Church also gave us the lambda calculus as a language for writing programs, and he was the PhD advisor of Alan Turing.

Sir C.A.R. Hoare used types to classify the data used in programming languages such as Algol and Pascal [64, 65].[5] For functional programming languages the data includes functions and recursive data structures such as lists and trees. Incorporating function types and recursive types made type theory very expressive. In the same period, 1971, Cornell came on the scene with the observation that using constructive proofs and Kleene's realizability semantics [75] for the logical operators, it would be possible to create algorithms from constructive proofs of mathematical theorems. This would provide a new approach to program correctness and formal methods by building programs that were *extracted* from proofs [33]. Cordell Green [57] had a related idea based on AI techniques. These novel approaches sometimes go by the name *correct by construction programming*. In 1984 the Nuprl proof assistant was operational at Cornell, created with NSF funding starting in 1980. This approach to programming could be demonstrated on real problems.[6] The Nuprl system in-

[4] To play it safe, the first users of Nuprl established the default bound of 17 levels of universes. We have never had to go beyond three, so de Bruijn seems to have had a valid intuitive insight.

[5] His title is used to stress that the type theory "came from Britain". Brouwer called types "species."

[6] In 1981 Cordell Green formed a company, the *Kestrel Institute*, (http://www.kestrel.edu/home/people/green/) based on his AI approach. His team created the Kids system to implement correct by construction programming. Later they built a package of tools called Specware. We have worked with them from time to time.

fluenced the AI group at Edinburgh University to build a related system, called *Oyster-Clam* [28], because it had a PRL in it. The Edinburgh system was focused on *completely automating* certain proof search methods. We coordinated efforts over the years, and that collaboration along with our research on Robin Milner's LCF system [56], created very strong long term ties between Cornell and Edinburgh universities.

In the same period there was a vigorous study of how to formulate a constructive type theory by two strong rivals, Girard [52] and Martin-Löf [81]. At some point we might briefly compare these theories.

Coverage: This course will cover the *core ideas of modern constructive and intuitionistic type theory* and show how these ideas are used with proof assistants to build reliable software, to formalize and extend computational mathematics and to explore, formalize and automate novel features of computation that have the potential to deepen and broaden computer science. The ideas integrated in constructive type theories will have the potential to provide a foundational theory for computer science. We might discuss from time to time whether that notion makes sense for computer science, a field so broad, fast moving, and technology driven.

Examples will be drawn from modern proof assistants such as Agda [25], Coq [22], Nuprl [37, 7], and possibly HOL [55, 54, 60]. When appropriate the lectures will mention developments underway in industry such as Lean at Microsoft Research (MSR). Lectures will compare at a high level the type theories implemented by Coq and Nuprl and discuss the interesting relationship that Drs. Anand and Rahli have created by using Coq to verify Nuprl proof rules [11, 10, 94, 9]. Currently Dr. Rahli is working on a way to check and run Nuprl proofs in Coq. We will discuss the expected evolution of proof assistants into AI tools of enormous power to advance the research agenda of computer science and contribute new capabilities for exploring and implementing novel mathematical ideas and theories. In the realm of knowledge creation, believed for centuries to be the sole province of humans, we now see human computer collaboration advancing scientific knowledge at an ever faster pace. This autocatalytic processes, characteristic of artificial intelligence (AI) and increasingly manifest in the expanding use of proof assistants in both research and education, is showing us the outlines of a brave new world.

It is common to think that careful proofs of the correctness of algorithms comes *after* we have thoroughly understood a problem, explored different algorithms for solving it, perhaps even implementing them and observing their behavior. Then when we know we have a good algorithm and a sensible explanation of why it works, we might try to carefully prove its correctness. It is easy to see a role for proof assistants in this scenario, they help create a formally correct proof that the algorithm solves the problem.

This "standard scenario" leaves out one of the most important roles for proof assistants, that is helping researchers explore the space of possible solutions and experimenting with different ways of understanding and explaining algorithmic solutions in detail. This process helps us develop an integrated understanding for how to precisely define an algorithmic task as well as how to produce a provably correct solution to it. Details of algorithms can be driven by proof strategies, and proof strategies can be revealed by algorithms. Kleinberg and Tardos describe this process of explo-

ration in the preface to their well known book on algorithms, *Algorithm Design* [76]: "Algorithmic problems form the heart of computer science, but they rarely arrive as cleanly packaged, mathematically precise questions." They go on to say:"As a result, the algorithmic enterprise consists of two fundamental components: getting to the mathematically clean core of the problem, and then identifying the appropriate algorithm design techniques, based on the structure of the problem." It is by supporting this integrated process that proof assistants play a doubly effective role, one in helping state precisely the mathematical questions, and two in finding answers to them by helping explore the solution space.

Main Topics: The specific lecture topics will include these:

1. Introduction to the course content (this document) and the nature of assignments.
2. Constructive propositional logic and first-order logic and their operational semantics as programming languages as well as logical systems.
3. Basic CTT types in Nuprl: void, integers, n-tuples, disjoint unions, functions, recursive types; see [106, 12, 34, 35, 31].
4. Brief overview of constructive type theory for expressing basic concepts, theorems, and methods of computer science. Illustrating the use of *proofs as programs* on simple examples. High level discussion of automating formal reasoning using the Nuprl proof assistant based on a *distributed LCF tactic mechanism* and the Nuprl *Formal Digital Library* (FDL).
5. Discussion of the relationship of type theory (CTT) to set theory (ZFC). Set theory has been the foundational theory for mathematics since approximately 1895 with a long record of success. Understanding the relationship between type theory and set theory will inform efforts to use type theory as a foundational framework for computer science. There are important logical results on this topic including a formulation of a constructive set theory in type theory [1, 4, 6].
6. Further CTT types in Nuprl: Atoms, set types, quotient types, intersection types, dependent intersection types [35].
7. Brief discussion of other logics and their proof assistants, including classical Higher-Order Logic and the HOL prover and the Coq total type theory and its proof engine. Coq is a widely used proof assistant, perhaps the best known, and we will briefly discuss its architecture and its role in proving the soundness of the Nuprl rules.
8. Expressing classical logics and *classical proofs as programs* using the Nuprl concept of *virtual evidence*.
9. Euclidean geometry in constructive type theory as an approach to designing and verifying *synthetic geometry algorithms* and as a basis for implementing provably correct analytical computational geometry algorithms, typically using the algebraic reals or the constructive reals or the intuitionistic reals.
10. Real numbers and constructive analysis in the Bishop style.

11. Options for expressing *computational complexity* in extensional constructive type theories.

12. Computability beyond the Church-Turing notion using *free-choice sequences*, Bar Induction, Continuity Principle, exceptions.

13. Applying the Continuity Principle in intuitionistic mathematics, *Bickford's inseparability theorem* for the continuum.

14. Intuitionistic synthetic geometry and its applications in computational geometry.

15. Models of distributed computation using free choice sequences, event logic in constructive type theory, proving distributed protocols correct and deriving them from proofs. Constructive FLP theorem.

16. Further discussion of distributed proofs assistants and their role in AI.

17. Formalizing Homotopy Type Theory, specifically Cubical type theory in Nuprl, presented by Dr. Bickford.

18. Next generation proof assistants.

Requirements: Students will write a research paper on a mutually agreed on topic. Students will help by looking at one set of lecture notes and making suggestions for how to improve them. There will be four or five exercises on the lecture material that students will submit in writing.

Free text book: *Type Theory and Functional Programming* [106] by Simon Thompson, pdf available on-line on the Web. Various educational and research articles will be handed out as course material. The 2008 book Logicomix: *An Epic Search for Truth* [44] by Doxiadis and Papadimitriou is related recreational reading.

The lecture notes will constitute a small booklet for this approach to constructive type theory.

Related Readings: Here is a short list of recommended articles. Some of these will be made available as course material such as :[12, 37, 8].

1. *Type Theory and Functional Programming* [106], free on line.

2. *Do-it-yourself Type theory* [12].

3. *Naïve Computational Type Theory* [34].

4. *On the Meaning of the Logical Constants and the Justification of the Logical Laws* [82] handout.

5. *Implementing Mathematics with the Nuprl Proof Development System* [37].

6. *Innovations in Computational Type Theory using Nuprl* [8].

2 Importance of Type Theory in Mathematics and Computer Science

Types played a very important and prominent role in providing a foundation for mathematics, going back to the 1925 monumental three volume work by Alfred North Whitehead and Bertrand Russell, *Principia Mathematica* [108] (also featured in Logicomic cited above.) Type theory was simplified by Church [30] in his *Simple Theory of Types,* and types gradually appeared in computer science via programming languages such as Algol 60 and FORTRAN and later Pascal [66] and Forsythe [100]. In these languages, types are used to distinguish different kinds of data, e.g. fixed point numbers, floating points numbers, arrays and so forth. As programming languages became more expressive, their type systems were steadily enriched. Hoare and Reynolds wrote very influential articles that stressed the importance of types [64, 66, 65, 95, 98, 99]. As it became more and more important to reason carefully about whether programs accomplished the tasks they were intended to accomplish, the importance of types increased further. Languages such as Algol 68 had quite rich type systems. John Reynolds wrote extensively on this topic and designed the language Forsythe [100]. He encouraged the further enrichment of types systems for programming [96, 97, 99]. The programming language Pascal [66, 71] played a major role in showing the advantages of a rich type system for explaining programming tasks.

3 First-Order Logic and Peano Arithmetic a la Kleene

We will use Kleene's account of first order logic and Peano Arithmetic [73] on page 82 and recommend a simple presentation of first order logic by Smullyan [105] as optional supplementary reading that might be mentioned from time to time. Here we list the axioms. Then we discuss how these are used in a formal system to build theories. If these logical formulas are totally mysterious, then it is best to take the course CS4860 (Math4860) on *Applied Logic* which I will teach in the spring semester.

Below are the axioms from Kleene's book. They are listed here to illustrate formal logic. The course assumes some familiarity with logical notation but not necessarily with formal theories. So most students should find these axioms readable, but not that many will know how to create a useable formal theory from them. We will explore that issue already in the first two lectures.

Propositional Calculus Axioms

1a. $A \Rightarrow (B \Rightarrow A)$.
1b. $(A \Rightarrow B) \Rightarrow ((A \Rightarrow (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$.

Inference Rule 2: $\dfrac{A, A \Rightarrow B}{B}$

3. $A \Rightarrow (B \Rightarrow A\&B)$.
4a. $(A\&B) \Rightarrow A$.
4b. $(A\&B) \Rightarrow B$.
5a. $A \Rightarrow (A \vee B)$.
5b. $B \Rightarrow (A \vee B)$.
6. $(A \Rightarrow C) \Rightarrow ((B \Rightarrow C) \Rightarrow ((A \vee B) \Rightarrow C))$.
7. $(A \Rightarrow B) \Rightarrow ((A \Rightarrow \sim B) \Rightarrow \sim A)$.
8*. $\sim\sim A \Rightarrow A$. **classical**

Predicate Calculus Axioms

Inference Rule 9: $\dfrac{C \Rightarrow A(x)}{C \Rightarrow \forall x.A(x)}$

10. $\forall x.A(x) \Rightarrow A(t)$.
11. $A(t) \Rightarrow \exists x.A(x)$.

Inference Rule 12: $\dfrac{A(x) \Rightarrow C}{\exists x.A(x) \Rightarrow C}$

Number Theory Axioms

13. $A(0)\&\forall x.(A(x) \Rightarrow A(x')) \Rightarrow A(x)$.
14. $a' = b' \Rightarrow a = b$.
15. $\sim (a' = 0)$.
16. $a = b \Rightarrow (a = c) \Rightarrow b = c$.
17. $a = b \Rightarrow a' = b'$.
18. $a + 0 = a$.
19. $a + b' = (a + b)'$.
20. $a \times 0 = 0$.
21. $a \times b' = (a \times b) + a$.

How do we use these axioms to create proofs? Kleene gives this example on page 85. This is an example of what logicians call a Hilbert style axiom system in which there is only one proof rule in addition to the axioms. The is the rule of *modus ponens*, if we know A and $A \Rightarrow B$, then we can deduce B.

1.$A \Rightarrow (A \Rightarrow A)$. Axiom schema 1a.
2.$(A \Rightarrow (A \Rightarrow A)) \Rightarrow (A \Rightarrow ((A \Rightarrow A) \Rightarrow A))(A \Rightarrow A)$. Schema 1b.
3.$(A \Rightarrow ((A \Rightarrow A) \Rightarrow A))$. Axiom schema 1a.
4.$(A \Rightarrow A)$. Rules 2,4,3.

Kleene also shows on page 84 how to prove $a = a$ for numerical variables a in 17 lines of axioms and inference rules. This kind of proof makes formal logic and formal mathematics seem *impossibly tedious* and essentially incomprehensible. What computer science brought to this research early on is the notion that machines could conduct these tedious proofs in the background and achieve an exceptionally strong degree of formal correctness. In their classic paper, *Empirical Explorations with the Logic Theory Machine: A case study in heuristics* [88], Newell, Shaw, and Simon demonstrated that computers could execute the tediously long proofs and check all the details so that humans would not be required to function at these low levels. This was a revolutionary advance in automated reasoning that inspired decades of further research to create the AI tools that make proof assistants possible.

We will examine an example from Kleene showing how to structure proofs as trees using Frege's turnstile symbol separating hypotheses from the goal to be proved, $H \vdash G$ where the hypotheses H is a list of formulas and variable declarations and the goal is a single formula. These expressions are called *sequents*. In the Nuprl book they are written as $H >> G$. Some formalisms, such as tableaux, allow multiple goals, but Nuprl does not.

4 Zermelo Fraenkel Set Theory with Choice (ZFC)

There are several versions of "standard set theory." Below is the way Fraenkel wrote about it in 1958. For his subset axiom, he uses a variable B ranging over first-order formulas. Some axiomatizations use variables for the notion of a *class*, which is a set like object that is not in the range of quantification over all sets. It is interesting that set theory needs such a notion to provide for a "first-order" axiomatization. The set theory implemented in the Mizar system is called *Tarksi-Grothendieck Set Theory*, TG. There are a number of other variants of axiomatized set theory including the one used by the Bourbaki project to create an encyclopedia of all of mathematics. That set theory uses the Hilbert epsilon operator. One of the best informal accounts of set theory is in the book by Halmos *Naive Set Theory*[58]. This inspired my "popular" account of type theory, *Naive Computational Type Theory*[34].http://www.nuprl.org/html/NaiveTypeTheoryRevisions.html

Here is how Fraenkel [51] presented the axioms for the set theory partly named after him, ZF. The theory is defined in First-Order Logic (FOL). There are several axiomatizations of set theory with different features, but we will only cover two since this is not a subject we will pursue in detail. Indeed, type theory is a "competing foundation" for mathematics. We will see the reasons why set theory would not be the chosen foundational theory for computer science if there were one. Typically the *Axiom of Choice* is included among the axioms, and the theory is called **ZFC**.

Definitions of Equality: Fraenkel discusses the options for defining equality on sets. We look at this discussion because in type theory, every type comes with its definition of equality, and that is critical to understanding the type. In set theory, there is a fixed primitive definition of set equality which underlies equalities that might arise for certain defined sets, e.g. for the set theoretic rational numbers and real numbers.

5 Proof Systems

We have seen two sets of axioms presented in the Hilbert style. There are several other logical frameworks for presenting rules such as *Natural Deduction* [92, 93], *Sequent Calculi* [46] and the *Refinement Logic* of Nuprl. We will look at all three briefly and then stay with the Nuprl framework. It has the advantage of supporting a highly parallel implementation and readable proofs. A further step toward readability would be to translate the proofs into text. We have experimented with such translations [68, 67].

References

[1] Peter Aczel. The type theoretic interpretation of constructive set theory. In A. MacIntyre, L. Pacholski, and J. Paris, editors, *Logic Colloquium '77*. North Holland, 1978.

[2] Peter Aczel. The type theoretic interpretation of constructive set theory: Choice principles. In S.S. Troelstra and D. van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium*. North Holland, 1982.

[3] Peter Aczel. The type theoretic interpretation of constructive set theory: Inductive definition. In *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier Science Publishers, 1986.

[4] Peter Aczel. On relating type theories and set theories. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs: International Workshop, TYPES '98, Kloster Irsee, Germany, March 1998*, volume 1657 of *Lecture Notes in Computer Science*, pages 1–18, 1999.

[5] Peter Aczel and Michael Rathjen. Notes on constructive set theory. Technical Report 40, Mittag-Leffler, 2000/2001.

[6] Peter Aczel and Michael Rathjen. Notes on constructive set theory. Technical Report 40, Mittag-Leffler, 2000/2001.

[7] Stuart Allen, Mark Bickford, Robert Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[8] Stuart Allen, Mark Bickford, Robert Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and Evan Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[9] Abhishek Anand, Mark Bickford, Robert Constable, and Vincent Rahli. A type theory with partial equivalence relations as types. In *TYPES 2014*. 2014.

[10] Abhishek Anand and Ross Knepper. ROSCoq: Robots powered by constructive reals. LNCS, pages 34–50.

[11] Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. In *International Conference on Interactive Theorem Proving*, pages 95–197, 2014.

[12] R. C. Backhouse, P. Chisholm, and G. Malcolm. Do-it-yourself type theory (part II). *EATCS Bulletin*, 35:205–245, 1988.

[13] Eli Barzilay and Stuart Allen. Reflecting higher-order abstract syntax in Nuprl. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Theorem Proving in Higher Order Logics; Track B Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002), Hampton, VA, August 2002*, pages 23–32. National Aeronautics and Space Administration, 2002.

[14] J. L. Bates and Robert L. Constable. Definition of micro-PRL. Technical Report 82–492, Cornell University, Computer Science Department, Ithaca, NY, 1981.

[15] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions of Programming Language Systems*, 7(1):53–71, 1985.

[16] Michael J. Beeson. Constructive Geometry. *Proceedings of the tenth Asian logic colloquium*, pages 19–84, 2009.

[17] Michael J. Beeson. Logic of ruler and compass constructions. In S. Barry Cooper, Anuj Dawar, and Benedict Loewe, editors, *Computability in Europe 2012*, Lecture Notes in Computer Science, pages 46–55. Springer, 2012.

[18] Michael J. Beeson. Proof and computation in geometry. In Tetsuo Ida and Jacques Fleuriot, editors, *Automated Deduction in Geometry*, volume 7993 of *Springer Lecture Notes in Artificial Intelligence*, pages 1–30. Springer, 2013.

[19] Michael J. Beeson. A constructive version of Tarski's geometry. *Annals of Pure and Applied Logic*, 2015.

[20] Ralph Benzinger. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 11(1):3–31, January 2001.

[21] Ralph Benzinger. *Automated Computational Complexity Analysis*. PhD thesis, Cornell University, 2001. Cornell Technical Report TR2002-1880.

[22] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[23] Ken Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–161, Hilton Head, SC, 2000. IEEE Computer Society Press.

[24] Kenneth P. Birman and Robbert van Renesse, editors. *The Isis Book: Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[25] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *LNCS 5674, Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[26] L.E.J. Brouwer. Intuitionism and formalism. *Bull Amer. Math. Soc.*, 20(2):81–96, 1913.

[27] L.E.J Brouwer. *Brouwer's Cambridge Lectures on Intuitionism.* Cambridge University Press, 1981.

[28] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In Mark E. Stickel, editor, *Proceedings of the 10^{th} International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 647–648. Springer-Verlag, 1990.

[29] Burnikel, Fleischer, Mehlhorn, and Schirra. Efficient exact geometric computation made easy. In *COMPGEOM: Annual ACM Symposium on Computational Geometry*, 1999.

[30] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:55–68, 1940.

[31] R. L. Constable. The structure of Nuprl's type theory. In Helmut Schwichtenberg, editor, *Logic of Computation*, volume 157 of *Series F: Computer and Systems Sciences*, pages 123–156, Berlin, 1997. NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 25–August 6, 1995, Springer.

[32] Robert Constable and Wojciech Moczydlowski. Extracting programs from constructive HOL proofs via izf set-theoretic semantics. In *Proceeding of 3rd International Joint Conference on Automated Reasoning (IJCAR 2006), LNCS 4130*, pages 162–176. Springer, New York, 2006.

[33] Robert L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233. North-Holland, 1971.

[34] Robert L. Constable. Naïve computational type theory. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability, Proceedings of International Summer School Marktoberdorf, July 24 to August 5, 2001*, volume 62 of *NATO Science Series III*, pages 213–260, Amsterdam, 2002. Kluwer Academic Publishers.

[35] Robert L. Constable. Computational type theory. *Scholarpedia*, 4(2):7618, 2009.

[36] Robert L. Constable. Virtual evidence: A constructive semantics for classical logics. Technical Report arXiv:1409.0266, Computing and Information Science Technical Reports, Cornell University, 2014.

[37] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, NJ, 1986.

[38] Robert L. Constable, S. Johnson, and C. Eichenlaub. *Introduction to the PL/CV2 Programming Logic*, volume 135 of *Lecture Notes in Computer Science.* Springer-Verlag, NY, 1982.

[39] Robert L. Constable and Michael J. O'Donnell. *A Programming Logic*. Winthrop, Cambridge, Mass., 1978.

[40] H.S.M. Coxeter. *Introduction to Geometry Second Edition*. John Wiley and Sons, New York, NY.

[41] N. G. de Bruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[42] R. A. De Millo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Communications of the Association of Computing Machinery*, 22:271–280, 1979.

[43] M deBerg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry*. Springer, 2008.

[44] A Doxiadis and C. Papadimitriou. *Logicomix: An Epic Search for Truth*. Ikaros Publications, Greece, 2008.

[45] H. Dreyfus. *What Computer's Still Can't Do*. MIT Press, 1992.

[46] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *The Journal of Symbolic Logic*, 57(3), September 1992.

[47] George B. Dyson. *Darwin Among the Machines: The Evolution of Global Intelligence*. Perseus Books, 1997.

[48] Euclid. *The Elements*. Green Lion Press, Santa Fe, New Mexico, 2007.

[49] Euclid. *Elements*. Dover, approx 300 BCE. Translated by Sir Thomas L. Heath.

[50] Amy P. Felty and Douglas J. Howe. Hybrid interactive theorem proving using Nuprl and HOL. In *CADE 97, LNAI 1249*, pages 351–365. Springer.

[51] A. A. Fraenkel and Y. Bar-Hillel. *Foundations of Set Theory*, volume 67 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 2nd edition, 1958.

[52] J-Y. Girard. The system F of variable types: Fifteen years later. *Journal of Theoretical Computer Science*, 45:159–192, 1986.

[53] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. English version in [107].

[54] Michael Gordon. HOL: A machine oriented formalization of higher order logic. Technical Report 68, Cambridge University, 1985.

[55] Michael Gordon and Tom Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, 1993.

[56] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[57] C. C. Green. An application of theorem proving to problem solving. In *IJCAI-69—Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 219–239, Washington, DC, May 1969.

[58] Paul R. Halmos. *Naive Set Theory*. Springer-Verlag, New York, 1974.

[59] Robert Harper, D.B. MacQueen, and R. Milner. Standard ML. Technical Report TR ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, University of Edinburgh, 1986.

[60] John Harrison. HOLLight: A tutorial introduction. In *Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[61] Peter Henrici. *Applied and Computational Complex Analysis*, volume 1–3. John Wiley and Sons, New York, 1988.

[62] A. Heyting, editor. *L. E. J. Brouwer. Collected Works*, volume 1. North-Holland, Amsterdam, 1975. (see On the foundations of mathematics 11-98.).

[63] David Hilbert. *Foundations of Geometry*. Open Court Publishing.

[64] C. A. R. Hoare. Notes on data structuring. In *Structured Programming*. Academic Press, New York, 1972.

[65] C. A. R. Hoare. Recursive data structures. *International Comput. Inform. Sci.*, 4(2):105–32, June 1975.

[66] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:333–335, 1973.

[67] Amanda Holland-Minkley. Planning proof content for communicating induction. In *Proceedings of Second International Natural Language Generation Conference*, pages 167–172, 2002.

[68] Amanda Holland-Minkley, Regina Barzilay, and Robert L. Constable. Verbalization of high-level formal proofs. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 277–284. AAAI, July 1999.

[69] Douglas J. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125, of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, Berlin, 1996.

[70] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 85–101. Springer-Verlag, Berlin, 1996.

[71] K. Jensen and N. Wirth. *PASCAL user manual and report*. Springer-Verlag, New York, 1974.

[72] G. Kahn. Natural semantics. In G. Vidal-Naquet F. Brandenburg and M. Wirsing, editors, *STACS '87*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, Berlin, 1987.

[73] S. C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand, Princeton, 1952.

[74] S. C. Kleene and R. E. Vesley. *Foundations of Intuitionistic Mathematics*. North-Holland, 1965.

[75] S.C. Kleene. On the interpretation of intuitionistic number theory. *J. of Symbolic Logic*, 10:109–124, 1945.

[76] Jon Kleinberg and Eva Tardos. Algorithm design. To appear, 2004.

[77] Xavier Leroy. *The Objective Caml System: Documentation and User's Manual*, 2002. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Availiable from `http://www.ocaml.org/`.

[78] Xiaoming Liu, Robbert van Renesse, Mark Bickford, Christoph Kreitz, and Robert Constable. Protocol switching: Exploiting meta-properties. In Luis Rodrigues and Michel Raynal, editors, *International Workshop on Applied Reliable Group Communication (WARGC 2001)*, pages 37–42. IEEE, 2001.

[79] Donald MacKenzie. *Mechanizing Proof*. MIT Press, Cambridge, 2001.

[80] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, Amsterdam, 1973.

[81] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[82] Per Martin-Löf. On the meaning of the logical constants and the justification of the logical laws. Lectures in Siena, 1983.

[83] Per Martin-Löf. *Intuitionistic Type Theory*. Number 1 in Studies in Proof Theory, Lecture Notes. Bibliopolis, Napoli, 1984.

[84] Per Martin-Löf. An intuitionistic theory of types. In Sambin and Smith [102], pages 127–172.

[85] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1991.

[86] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17:363–371, 1978.

[87] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml*. O'Reilly, Beijing, Cambridge, 2014.

[88] A. Newell, J.C. Shaw, and H.A. Simon. Empirical explorations with the logic theory machine: A case study in heuristics. In *Proceedings West Joint Computer Conference*, pages 218–239, 1957.

[89] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Sciences Publication, Oxford, 1990.

[90] Ivars Peterson. *Fatal Defect: Chasing Killer Computer Bugs*. Vintage Books, 1996.

[91] Iman Poernomo, John Crossley, and Martin Wirsing. *Adapting Proofs-as-Programs, the Curry-Howard Protocol*. Springer-Verlag, New York, 2005.

[92] D. Prawitz. *Natural Deduction*. Almquist and Wiksell, Stockholm, 1965.

[93] D. Prawitz. *Natural Deduction*. Dover Publications, 1965.

[94] Vincent Rahli, Mark Bickford, and Abhishek Anand. Formal program optimization in nuprl using computational equivalence and partial types. In *The 4th Conference on Interactive Theorem Proving (ITP 2013)*, Rennes, France, July 2013.

[95] John C. Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur, la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–23. Springer-Verlag, New York, 1974.

[96] John C. Reynolds. The essence of Algol. In J. de Bakker and J. van Vliet, editors, *International Symposium on Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.

[97] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983.

[98] John C. Reynolds. Polymorphism is not set-theoretic. In *Proceedings International Symposium on Sematics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156, Berlin, 1984. Springer.

[99] John C. Reynolds. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software, International Conference TACS '91*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700. Springer-Verlag, Sendai, Japan, 1991.

[100] John C. Reynolds. Design of the programming language forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996.

[101] Bertrand Russell. *The Principles of Mathematics*. Cambridge University Press, Cambridge, 1908.

[102] Giovanni Sambin and Jan M. Smith, editors. *Twenty-Five Years of Constructive Type Theory*, volume 36 of *Oxford Logic Guides*, Oxford, 1998. Clarendon Press.

[103] Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Mark Bickford, and Robert L. Constable. Developing correctly replicated databases using formal tools. pages 395–406, 2014.

[104] W. Schwabhäuser, Wanda Szmielew, and Alfred Tarski. *Metamathematische Methoden in der Geometrie*. Springer Verlag, Berlin, 1983.

[105] Raymond M. Smullyan. *First-Order Logic*. Dover Publications, New York, 1995.

[106] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.

[107] J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, MA, 1967.

[108] A.N. Whitehead and B. Russell. *Principia Mathematica*, volume 1, 2, 3. Cambridge University Press, 2nd edition, 1925–27.