Two Lectures on Constructive Type Theory

Robert L. Constable

July 20, 2015

Abstract

Main Goal: One goal of these two lectures is to explain how important ideas and problems from computer science and mathematics can be expressed well in constructive type theory and how proof assistants for type theory help us solve them. Another goal is to note examples of abstract mathematical ideas currently not expressed well enough in type theory. The two lectures will address the following three specific questions related to this goal.

Three Questions: One, what are the most important foundational ideas in computer science and mathematics that are expressed well in constructive type theory, and what concepts are more difficult to express? Two, how can proof assistants for type theory have a large impact on research and education, specifically in computer science, mathematics, and beyond? Three, what key ideas from type theory are students missing if they know only one of the modern type theories? The lectures are intended to complement the hands-on Nuprl tutorials by Dr. Mark Bickford that will introduce new topics as well as address these questions. The lectures refer to recent educational material posted on the PRL project web page, www.nuprl.org, especially the on-line article *Logical Investigations*, July 2014 on the front page of the web cite.

Background: Decades of experience using proof assistants for constructive type theories, such as Agda, Coq, and Nuprl, have brought type theory ever deeper into computer science research and education. New type theories implemented by proof assistants are being developed. Moreover, mathematicians are showing interest in enriching these theories to make them a more suitable foundation for mathematics. The scope of type theory includes providing a formal semantics for programming languages and expanding their capabilities. It includes specifying, defining, and building the next generation of compilers, operating systems, software defined networks, distributed systems and other critical software systems. It applies to *specifying* computational problems well and finding algorithms and systems that provably solve them. The scope includes establishing computational complexity bounds on problems and solutions. Type theory is also valuable in providing a precise semantics for natural languages and in understanding the limits and hazards of deep learning, and more generally in exploring epistemology in the Age of Artificial Intelligence.

Elements of constructive type theories are now taught to computer science undergraduates because of their relevance to the goals just mentioned. Moreover, proof assistants for these theories liberate students to confidently explore open problems. It is likely that teaching these *new tools for thought* will spread into the secondary school system in due course. It might have already happened in the most advanced secondary schools.

Topics: The first lecture will explain why the same *core ideas* appear in all of the major implemented constructive type theories. These ideas have also been widely studied by the *semantics community*.[1] There is one major difference in the order of presentation of the core ideas, even between Coq and Nuprl, despite their strong core commonality and their nearly thirty years of co-evolution. That difference is fundamental and goes back to Curry versus Church as well as Gödel and Herbrand versus Kleene. We now understand that it is much more fundamental than it seemed in the 20th century, as will be explained in the first lecture. The second lecture will explore ideas that are not present in Coq, yet heavily used in Nuprl.

New Results: Both lectures will touch briefly on foundational issues such as the relationship between set theory and type theory and the ability of each kind of theory to express abstractions from mathematics – past, present, and future. We will mention results from recursive function theory that illuminate the value of *universal* versus *subrecursive* type theories. We will also present briefly a new *evidence semantics* for classical logic using constructive type theory as an illustration of a fundamental problem and ways to attack it using type theory.

Constructive type theory and higher-order logics have played a noteworthy role in mathematics as well, both in mechanically confirming the solution of important problems such as the Four Color Theorem and Odd Order Theorem (Feit-Thompson Theorem) and in settling the Kepler Conjecture. More examples, large and small, will come from continuing use of proof assistants in mathematics. Also the clean embedding of *constructive set theory* into these type theories makes another strong connection to mathematics. The Field's Medalist Vladimir Voevodsky has been using proof assistants for type theory in his mathematics. The role he advocates is important both in the publication and validation process and in enriching the foundational base for mathematics. Other mathematicians have also agreed with Nicolas G. de Bruijn who called for the use of proof assistants to check and confirm the results of mathematics. These mathematicians believe that too many important results are now beyond the ability of referees to confirm them in a timely and cost effective way.

Voevodsky has initiated a research program to understand how to classify the various important varieties of modern type theories (using his C-systems). His track record suggests that this line of research will be important in mathematics as well as in computer science. However, the preferred foundation among mathematicians remains set theory. We will briefly discuss the relationship between type theory and set theory as seen from CTT and also consider the long and productive tradition of referencing *Platonic reality* to understand mathematical truth. This tradition may not be as strong in computer science because proof assistants do not yet and may never enjoy access to this reality. It is not clear that they ever will acquire it first hand, but we might teach them to consider it in explaining their results to us.

1 Lecture Outlines and Key Topics

Type theories implemented by the Agda, Coq and Nuprl proof assistants co-evolved over the past 30 years. We can learn a great deal from that evolution. We can even try to predict the "shape of theories to come." It is already predictable is that both CIC and CTT will continue to evolve and new type theories will appear implemented by new proof assistants. We already see this from Idris [24] and F* [146], and their role in education and research will continue to expand.

[1] We distinguish between Theory A, Algorithms and theory and Theory B, Formal Models and Semantics, following the classification in the titles of the two volumes of the *Handbook of Theoretical Computer Science* [159, 160].

There is also a strong tie between type theory and Higher-Order Logic (HOL) which is a version of Church's Simple Theory of Types (STT40) [33, 60]. It is the oldest of the implemented type theories [76, 127, 84]. Both CIC and CTT are *constructive* theories, HOL is a classical theory, but its core has a constructive semantics in IZF set theory [37].[2]

1.1 Common starting point

Below are brief outlines of the two lectures mentioned in the title. We have an excellent opportunity at this summer school to approach type theory with considerable *common knowledge* about a particular such theory, the *Calculus of Inductive Constructions* (CIC15) implemented by the Coq proof assistant. Our goal is to build on this common knowledge to discuss a broad view of type theory and examine some elements of *Constructive Type Theory* (CTT15), whose semantics is formalized in Coq's CIC15. CTT15 is implemented by the Nuprl distributed proof assistant. It evolved gradually over thirty one years from the CTT implemented in 1984, CTT84.

Let us consider four key features of CIC15 and refresh our common understanding about them and relate them to how CTT15 formalizes them.

- The CIC15 theory contains a programming language implemented by the proof assistant, call it *CoqPL*. Its functional programs are typed, they are *total* on their types, type checking is decidable (as in programming languages). Equality of programs is decidable. Such a logic is called *intensional*. Typically programming languages implement *partial functions*. Therefore type checking for programming carries less information than it does for CoqPL.

- The CIC types include a common core of the *dependent logical types* presented in a parsimonious way. Namely there are the *function space* types and then all other logical types are defined *inductively*. These types provide higher order logic. This theory is very close to Gödel's system T [70] implemented by Takasu in 1978 [149]. The proof of termination is based on a fundamental method due to Tait [147] and extended by Girard in 1971 [68, 69]. The fundamental idea behind this definition of the logical operators is called by several names, propositions-as-types principle, the Curry-Howard isomorphism [51, 92, 55, 144] are common in computer science along with the related notion of proofs as programs. The idea has a long history in logic [144] and goes by other names as well. It is closely related to Kleene's concept of recursive *realizability* [100], and to what is known in logic as the Brouwer-Heyting-Kolmogorov (BHK) semantics for constructive logic [151, 152, 153] which was extended by Veldman [162] and is sometimes called BHKV semantics [35]. *This is a deep idea that is being steadily enriched as it gradually works its way into both computer science, mathematics, and philosophy.*

- The Agda, Coq, and Nuprl proof assistants can *extract* (functional) programs of dependent type $x : A \to B(x)$ from constructive proofs of propositions of the form $\forall x : A.\exists y : B(x).R(x, y)$. This notion goes back to the ideas of Kleene on *recursive realizability* [100] based on his understanding of Brouwer's notion of a mental construction. I presented Kleene's idea

[2] In 1985 HOL implemented a theory close Church's Simple Type Theory, say STT85. In 1984 Nuprl implemented a theory we call Constructive Type Theory, say CTT84 [44], based on the semantics of *partial equivalence relations* [6, 7, 83]. In 1988 Coq implemented a Calculus of Constructions [47]. Later, this theory changed substantially to become the Calculus of Inductive Constructions [18].

in the context of programming languages, suggesting in 1971 [38] that we could implement constructive logics as programming languages. This led to the slogan *proofs-as-program* or programs-as-proofs which Joe Bates and I realized in the early 1980's [16].

- Whether two objects of a type are equal depends on the type. Equality in function types, such as $A \to B$, is based on syntactic (*intensional*) equality of the expressions denoting the type.

We believe that the propositions-as-types idea will change the way we specify programs and the way we teach programming, even in the first year and eventually even in secondary schools. It should also change the way we teach logic. That might take longer. The recommended reading for this lecture is the short article Naïve Computational Type Theory [41]. Also the *Scholarpedia* article *Computational Type Theory* treats some of the same issues. It has enjoyed many visits.

My charge from the organizers is to bring some historical perspective to the subject and to discuss a few key ideas that I think are promising for the future development of programming languages and type theory. So these lectures will have a blend of specific results and connections to what I see as the *big picture* for the area.

Lecture One: Basic Concepts in Constructive Type Theory (CTT)

1. The core logical types
- dependent function, dependent product [3]
- options (or) A + B
- atomic types: equality $s =_T t$, Void
- universes $\mathbb{U}_i$
- CTT vs CIC on the core - *extensional* vs intensional type theory

2. History and understanding the core
- From Brouwer's intuitionism to Kleene's realizability
- From Bishop's sets to partial equivalence relations
- Brouwer/Heyting/Kolmogorov (BHK) semantics
- Propositions-as-types aka Curry-Howard Isomorphism

3. Beyond the core
- quotient types $A//x, y : E$
- set types $\{x : A|\ B(x)\}$
- intersection types and uniformity, $\bigcap_{x:A} B(x)$
- subtyping $\sqsubseteq$
- dependent intersection and dependent records
- uniform validity and constructive completeness of iFOL
- $\neg\neg(P \vee \neg P)$ (Brouwer's proof), CTT proof
- Kolmogorov's insight, $\neg\neg P \Rightarrow P$

4. Propositions and Types (two way street)
- long evolution, many lessons
- written material (see www.nuprl.org) OPLSS15
- subtyping, intersection, uniformity
- uniform validity

5. Understanding classical logic
- Kolmogorov continued $\neg\neg P \Rightarrow P$
- set type and *virtual evidence* $\{Unit|P\}$
- Classical Intro rule $\neg\neg P \Rightarrow \{P\}$
- virtual evidence and classical logic

6. Computational aspects of CTT (from Dr. Bickford's lectures)
- Terms and the type Base
- computation rules
- equality, extensional equality of functions

[3] There is a tradition in the Coq and Agda line, going back at least to Martin-Löf, to call the *dependent product* a *sigma type*, Σ. We have preferred to note Decartes and his *Cartesian product*.

Lecture Two: Foundational matters

1. Partial types
- CTT84 was designed for computer science
- types $\bar{A}$
- expressing the halting problem on $\bar{\mathbb{N}}$

2. Subrecursive languages
- CIC vs CTT as a programming language
- subrecursive languages, primitive recursion
- puzzle about subrecursive languages

3. Blum Size Theorem
- why subrecursive programs are longer than partial recursive ones
- sketching a proof of Blum's theorem

4. Partial types and Universality
- Turing completeness
- Kleene's partial recursive functions
- Base: a basis for recursive function theory
- Rice's theorem as an example
- Semantics for Hoare Logic

5. Abstraction in CTT versus ZFC
- why mathematicians like sets
- Tarski-Grothendieck set theory (Mizar)
- expressing sets in CTT
- other mathematical primitives: points, line, geometry

6. Conclusion and Discussion
- revisiting our three questions
- prospects for the next decade

2 Lecture One

We will take advantage of the fact that Agda, Coq, and Nuprl all express the core logical types in essentially the same way, *as types in a rich programming language*. Coq and Agda start with a typed programming language, while Nuprl starts with an *untyped programming language* in the manner of Lisp and uses that to define the type system.

The constructive semantics for logical types can also be given using Brouwer's notion of *mental constructions*. That path follows the history of the idea. Already by 1907 Brouwer began to think of proofs as a mental constructions, so some accounts of the semantics of the core logical types

speak about proofs. This is a conceptually more difficult path now that programming languages are so widely known and used. Formal logical systems are a less common and more complex idea. They require that we explain proofs and explicit proof rules first and then computation.

The terminology we prefer when discussing the Nuprl implementation of these ideas is that we explain the semantics of propositional logic in terms of ways of effectively transforming evidence for how we know a proposition, say A, into evidence for knowing other propositions.

A topic that is less widely discussed is that in the constructive account of logic in terms of types, it is important to know that we are talking about a sensible proposition or a sensible notion of evidence for a proposition. Frege [65] made us aware of this obligation. In Agda and Coq, *knowing that a proposition is sensible is done by the type checker.* Nuprl takes quite a different approach to ensuring that an expression is sensible. Our design idea is that we *simultaneously establish that a proposition or type makes sense* during our efforts to construct evidence for it. In Coq and Agda, this is a syntactic first step, handled by a type checker. *We note that in mathematics, it is sometimes necessary to invoke proofs even to know that a symbolic expression is meaningful.*

In the early days of designing Nuprl, we were strongly influenced by Bishop [20, 21] and constructive mathematical practice. It seemed to us that various mathematical notations and propositions were too complex to be checked by a type checker. So we followed Martin-Löf's early idea [113] that it was necessary to *prove* that a expression was a legitimate type expression. He subsequently abandoned this idea in the current theory implemented by Agda and Coq [114]. *Nuprl introduced the added feature that it is possible to prove that a proposition is sensible simultaneously with providing evidence for knowing it.* We left open the option of proving sensibility first.

2.1 The core logical types and the nature of evidence

The material presented in this subsection is also covered in an expository article written with Anne Trostle and posted on the PRL web cite (www.nuprl.org) front page. It can be accessed at: http://www.nuprl.org/MathLibrary/LogicalInvestigations/index.html

We discuss the function type first, $A \Rightarrow B$ where A and B are types. This is the type of computable functions from A to B, and the canonical elements are taken to be lambda terms $\lambda(x.b(x))$, using *abstract syntax* in the style of McCarthy [116] to write the lambda terms. Based on the standard computation rules for the lambda calculus, it is easy to see that these functions are computable. This point was stressed by Church who proposed his famous Thesis that all effectively computable functions can be expressed as lambda terms. That is certainly a far better notation than Turing machines for presenting a high-level programming language. It took many years for this simple point to be widely appreciated. John McCarthy saw it very early [115, 117].[4]

The function type and implication illustrate well the basic ideas connecting constructive type theory to logic. We elaborate by looking at the standard first-order propositions. This section is covered on the PRL group web page in a July 2014 posting by Anne Trostle and me entitled *Logical Investigations*, mentioned earlier.

[4]Gödel believed that the Turing machine was perhaps the only absolute concept in all of logic and that it was the superior formalism for expressing computable functions. In 1937 Turing showed how to compile λ terms into Turing Machines [156].

Implication as a function type The function type explains the constructive interpretation of implication. This interpretation does not refer to truth or Boolean expressions. It is based on Brouwer's understanding of implication as a computational method of converting any mental construction a by which we know A, into a mental construction $b(a)$ by which we know B. The lambda term $\lambda(x.b(x))$ expresses this method computationally, and it serves as a mental construction by which we know A *implies* B.

It is also necessary to know that $A \Rightarrow B$ is constructively meaningful. This depends on knowing that both A and B are constructively meaningful propositions. Implication is sensible as a type because we can grasp the idea of a mental operation that transforms objects of type A into objects of type B. To grasp this idea we need to understand the notion of a function as an operation or mental construction. For Nuprl, this idea comes first and is part of the computation system or programming language of the type theory. We know this concept operationally. It is developed using the basic concepts of the *untyped lambda calculus* or equivalently the *combinators*. This calculus is a way of making precise the idea of a mental construction. It is perhaps not essential for humans that we write down these constructions symbolically because we can be taught how to perform them mentally. But to get help from machines in performing them, we must encode them symbolically.

The typical Nuprl proof rule for implication ties all these notions together. It uses the style of refinement logic advocated by Joseph Bates in his PhD thesis [15] and supported by the LCF tactic mechanism [77]. Here is the way that rule is presented in this style.

For each rule we provide a name that is the *outermost operator* of a proof expression with slots to be filled in as the refinement style proof is developed. The proofs are organized as a tree generated in two passes. The first pass is top down, driven by the user creating terms with slots to be filled in on the algorithmic bottom up pass once the downward pass is complete. Here is a simple proof of the intuitionistic tautology $A \Rightarrow (B \Rightarrow A)$.

$$H, A, B : Prop_1 \vdash A \Rightarrow (B \Rightarrow A) \text{ by } \lambda(x.slot_1(x))$$

$$H, A, B : Prop_1, x : A \vdash (B \Rightarrow A) \text{ by } slot_1(x)$$

In the next step, $slot_1(x)$ is replaced at the leaf of the tree by $\lambda(y.slot_2(x, y))$ to give:

$$H, A, B : Prop_1 \vdash A \Rightarrow (B \Rightarrow A) \text{ by } \lambda(x.slot_1(x))$$
$$H, A, B : Prop_1, x : A \vdash (B \Rightarrow A) \text{ by } \lambda(y.slot_2(x, y)) \text{ for } slot_1(x)$$
$$H, A, B : Prop_1, x : A, y : B \vdash A \text{ by } slot_2(x, y)$$

When the proof is complete, we see the slots filled in at each inference step as in:

$$H, A, B : Prop_1 \vdash A \Rightarrow (B \Rightarrow A) \text{ by } \lambda(x.\lambda(y.x))$$
$$H, A, B : Prop_1, x : A \vdash (B \Rightarrow A) \text{ by } \lambda(y.x)$$
$$H, A, B : Prop_1, x : A, y : B \vdash A \text{ by } x$$

In this example we don't discuss the reasoning about A and B as propositions. Instead we use the simple abbreviation $Prop_1$ to denote the universe $\mathbb{U}_1$. The Nuprl type theory follows Martin-Lóf's idea of using large types called *universes* to formulate a *predicative and cumulative hierarchy*

of large types, $\mathbb{U}_1, \mathbb{U}_2, ..., \mathbb{U}_i, ...$ This is a simplified version of Russell's notion of a hierarchy of types. Agda follows the same idea whereas Coq also introduces a notion of *impredicative type*, denoted *Prop*.

We will not discuss this topic here except to note that our use of $Prop_1$ above is not the same as the use of *Prop* in Coq. Typically constructive theories avoid impredicative objects. In Coq, the impredicative universe is a means of expressing certain non-constructive notions.

Typically sequent style proof rules are presented in pairs. First comes the rule for introducing the constructor of evidence, called *introduction rules* or right hand side rules. Then comes the rule for showing how to use the evidence when it is assumed among the hypotheses. These are called *elimination rules* or de-construction rules or left hand side rules.

The rule for using an implication, say $f : A \Rightarrow B$ is to find a value of type A, thought of as evidence for the proposition A, and then apply f to it to get an element $f(a)$ for B. The style of this elimination rule is delicate because we need a name for the evidence in B, but all of the names in the list of hypotheses are simply variables, and this name is naturally of the form $f(a)$. Indeed, we can't find this name $f(a)$ until we have completed the proof of A, and that might be a substantial part of the proof, a part we are not yet sure about. On the other hand, we would like to explore how to use the fact that we think we can get evidence for B. So the natural form of the rule is to that assume evidence v for B is available, say $v : B$. Later, when we have proved A, we can substitute $f(a)$ for v in the evidence assembled for the original goal. So the format of this rule in a refinement logic is this.

$$
\begin{array}{l}
H, A, B : Prop_1, f : A \Rightarrow B, H' \ \vdash \ G \ by \ apseq(f; ...; v.g(v)) \\
\quad H, A, B : Prop_1, f : A \Rightarrow B, v : B, H' \ \vdash \ G \ by \ g(ap(f; ...)) \\
\quad H, A, B : Prop_1, f : A \Rightarrow B, H' \ \vdash \ A \ by \ a
\end{array}
$$

In this form, the variable v is not used in the extract and is present only to indicate where the application of the function named by f is to be applied. The operator $ap(f; a)$ is simply the application of the function f to a normally written as $f(a)$. The operator $apseq(f; ...; v.g(v))$ is only used when a is yet to be determined.

Conjunction as a Cartesian product type For Brouwer, to know a conjunction of propositions, $A\&B$, is to experience mental constructions a for A and b for B. We assemble them into an ordered pair $pair\{a; b\}$. This pair is a single general construction that is necessary and sufficient evidence for knowing $A\&B$.

A refinement proof rule codifies this evidence as part of a goal directed method for seeking it and for simultaneously confirming that the conjunction is a sensible proposition. This evidence can be structured using a proof rule in the refinement style.

And Construction

$$H, A, B : Prop_1 \ \vdash \ A\&B \ by \ pair(slot_a; slot_b)$$
$$H, A, B : Prop_1 \ \vdash A \ by \ slot_a$$
$$H, A, B : Prop_1 \ \vdash B \ by \ slot_b$$

We use the terms $slot_a$, $slot_b$ to indicate parts of the proof to be filled in as the subgoals are proved. They are just place holders for the proof construction.

In Coq and Agda, the same general ideas are presented in a slightly different manner, and the Cartesian product is considered to be a special case of a *dependent sum*, called a Σ type. Confirming that the conjunction is well-formed, i.e. assembled from propositions or types, is done by the type checker.

With every construction rule of the logic, there is a corresponding rule for showing how to use the construction when it is available among the hypotheses. Here is the deconstruction rule for &.

And Decomposition

$$H, A, B : Prop_1, x : A\&B, H' \vdash G \ by \ spread(x; l, r.slot_g(l, r)) \ new \ l, r$$

$$H, A, B : Prop_1, l : A, r : B, H' \vdash G \ by \ slot_g(l, r)$$

The *spread* operator is the Nuprl expression for decomposing a pair into its left and right components. The term $slot_g(l, r)$ simply indicates that the evidence for the goal G can use an expression that references evidence for A and B obtained by decomposing the hypothesis $x : A\&B$.

Disjunction as disjoint union If we know that A and B are constructively sensible propositions, then one way to know their disjunction, A *or* B is to have been able to construct or be given evidence a for A and to know that it is evidence for A by tagging it as $inl(a)$, indicating that it is evidence for the left disjunct. The tag inl signals that this evidence is for the A disjunct. The only other way to know this disjunct is to construct or be given evidence b for the right disjunct B, tagged as $inr(b)$.

Brouwer noted that in many cases mathematicians are willing to accept weaker evidence for disjunctions by claiming that for any sensible proposition A, it will either eventually be the case that A is seen to be true or A will be seen to be false. To say it's false is to say that its negation, $\neg A$ is true or that if A is ever shown to be true, then we will know $False$. This general belief is called the *law of excluded middle*, a principle accepted in Aristotle's writings on logic. Brouwer argued that this is a misguided belief which leads people to think that they know propositions for which there is no evidence whatsoever, such as the belief that every well understood mathematical proposition can be decided one way or another. At any given time, there are many well understood propositions for which no one has found evidence one way or another. Any long standing open problem such as Goldbach's conjecture is a good example.

Here is the introduction rule for disjunction. It does not allow us to prove $A \vee \neg A$.

Or Construction

$$H, A, B : Prop_1 \ \vdash \ A \vee B \ by \ inl(slot_l)$$
$$H, A, B : Prop_1 \ \vdash A \ by \ slot_l$$

$$H, A, B : Prop_1 \ \vdash \ A \vee B \ by \ inr(slot_r)$$
$$H, A, B : Prop_1 \ \vdash B \ by \ slot_r$$

Or Decomposition

$$H, A, B : Prop_1, y : A \vee B, H' \vdash G \ by \ decide(y; l.leftslot(l); r.rightslot(r))$$

$$H, A, B : Prop_1, l : A, H' \vdash G \ by \ leftslot(l)$$

$$H, A, B : Prop_1, r : B, H' \vdash G \ by \ rightslot(r)$$

3 Historical Context

We can look at the rich history of constructive type theory at many scales. The fine scale takes us through the amazing story of the foundations of mathematics played out in the early 20th century. The *dramatis personae* include Kant, Frege, Russell, Whitehead, Poincare, Brouwer, Hilbert, Gödel, Church, Turing, Kleene, Kolmogorov, Markov as main actors and many of their students as significant players including Heyting, Kreisel, Scott, Friedman, and the key players in building Agda, Coq, and Nuprl, going back to the 1980's.

3.1 Large scale background

At a large scale the story is simple. Starting in 1907 L.E.J. Brouwer propounded an approach to mathematics based on *mental constructions* rather than a Platonic reality. For Brouwer, proofs were mental constructions of a particular sort (or type). Kleene saw that based on Church's Thesis he could make Brouwer's ideas precise in terms of *partial recursive functions* on the most primitive mathematical objects, natural numbers. He proposed what is called a *realizability interpretation* of intuitionistic arithmetic (e.g. Heyting Arithmetic, HA)[100].

Computer scientists noticed that this was a good way to code correct programs, derive them from proofs that computational problems are solvable, so called *proofs as programs* [38, 80, 16]. Bishop [20, 21] found an elegant way to use key ideas from Brouwer's approach to develop constructive analysis without Church's Thesis and without Brouwer's Bar Induction and Continuity axioms. Martin-Löf found a way to organize the ideas into a type theory [112, 113] adequate for constructive analysis. The PRL group at Cornell designed a constructive type theory, CTT84, in this Martin-Löf style but based on a partial equivalence relation semantics and extended with recursive types and partial functions *a la* Kleene. They made the case that the several additions made the type theory adequate for computer science as well as mathematics.

The PRL group designed and built a proof assistant to implement it, Nuprl [44].[5] Those students in turn have greatly advanced research in type theory. Soon after the Coq type theory and proof assistant were designed [94, 47, 49] augmenting Girard's type theory [69]. The style

[5]Most of the key students and researchers in the PRL group went on to distinguished academic careers and produced outstanding second and third generation students who continue to work in the area.

and method of working in these theories was explored in the book *Programming in Martin-Löf's Type Theory* [128]. By 2004 the Coq prover [18] was based on intensional Martin-Löf type theory [114]. This system recently won the ACM software systems award . One of the key developers was Chetan Murthy from the PRL group, well known for his outstanding work on a constructive proof of Higman's Lemma [124, 125] using Friedman's A-translation [66].

3.2 Fine scale background - logic and mathematics

Type theory was first developed by Bertrand Russell [139, 138] to provide a foundational theory adequate for mathematics. It is the basis for Russell and Whitehead's monumental *Principia Mathematica* [164]. Constructive type theories are more recent, and they are important to the foundations of both classical and constructive mathematics. They have also become broadly relevant in computer science because they have been implemented by *proof assistants* and widely applied to specify programming tasks and demonstrate that programs satisfy these specifications. Moreover, proof assistants provide programming languages with extremely rich *dependent type systems*. There are too many good examples to cite. One of the most widely mentioned is the verification of the back end of a C compiler, *Formal certification of a compiler back-end or: programming a compiler with a proof assistant*, by Xavier Leroy [106]. This work continues, formalizing more of the compiler, and it has been presented in detail at OPLSS in the past. In addition, proof assistants have been used to build *correct by construction* protocols [140] and other software artifacts large and small. Thus constructive type theories serve as an integrated approach to formal methods in the manner of *programming logics* [1, 13].

These implementations provide rich specification languages for programming tasks. However, because they are designed to be a foundation for mathematics, they typically provide more comprehensive foundational theories and a very wide scope for applications in mathematics and engineering, e.g. in robotics and cyber-physical systems. Thus they are also more broadly relevant, as in providing a basis for *natural language semantics* [134] and in addressing problems in the philosophy of knowledge [79, 148]. All of these many roles can be integrated, and thus it seems very plausible that constructive type theories can play a foundational role in both computer science and mathematics.

While it might seem too simplistic to propose this dual foundational role for constructive type theories, there is something deep in this conception, and these two lectures will try to convey it. Simply put, the idea is that *mathematics demands great generality and abstractness as well as computation in all its diversity.* Generality and abstractness may account for the success of set theory as the "official foundation" for modern mathematics [104, 23]. Constructive type theory also provides concreteness and precision for computation. It is undeniable that this is also a core part of mathematics and has been for over 2,400 years.

In the past, very distinguished mathematicians such as Nicolaas G. de Bruijn [52, 53, 54], and currently Vladimir Voevodsky [163, 131] sought to bring more abstract ideas into the core of type theory and to provide the abstraction and generality that the discipline requires without sacrificing the computational meaning of type theory. These lectures will make the case that this union is feasible, necessary, and important. One consequence will be a richer theory that serves computer science even more effectively and will be broad enough to count as a foundation for both mathematics and computer science. A key reason that this is a likely and useful outcome is that computer science is both one of the most mathematical of the sciences and one of the most broadly

applicable. Indeed it has some of the qualities of a meta-science as we will see just below from the history of type theory and its role in the birth of computer science.

3.3 Fine scale background - computer science

Computer science is a distinct scientific discipline with diverse intellectual roots and a broad research agenda. Its first roots emerged from 20th century mathematical logic as it was developed by Frege [65, 97, 57], Russell [139, 138, 164], Hilbert [86, 87], Brouwer [25, 26, 161, 157], Gödel [72, 61], Church [30, 32, 31, 33, 34], Turing [156, 121], and Kleene [100, 99, 98] among others. The initial impetus for the research that led to computer science came from the need to cope with a "foundational crisis" in mathematics originating in the 19th century.[6] Foundational issues in mathematics were already investigated by the ancient Greeks [12, 59]. They created the axiomatic method based on the notion of "self-evident" truths. Those ideas arose first in logic, perhaps the oldest academic discipline [12]. The early approach to logic drew on Plato's notion of truth in an ideal world and his theory of forms, after 350 BCE.

Platonic reality as a basis for mathematical truth has been pondered for over two thousand three hundred years and still motivates work in logic and the foundations of mathematics [150]. However, the traditional methods were inadequate to the task of resolving the 19th century foundational questions in mathematics. *Bold new ideas were required, and they led over the course of sixty years to a new foundation for mathematics and to the intellectual foundations of computer science – perhaps the first meta-science.* We will look at these ideas retrospectively as they apply to constructive type theory and to the task of these lectures. We will also consider the interesting possibility that type theory enriched by both computer science and mathematics could become a plausible new foundation for mathematics. It is already the case that proof assistants for mathematics have changed the way mathematicians understand their field. At the 1999 *Visions in Mathematics* conference in Tel Aviv, the Fields Medalist W.T. Gowers said "Briefly, my contention is that during the next century computers will become sufficiently good at proving theorems that the practice of pure mathematical research will be completely revolutionized ...." [78].

Now in the 21st century, computer science has been deeply enriched by contributions from physics, engineering, linguistics, psychology, economics, sociology, biology, and other disciplines. It now stands as one of the newest and most dynamic sciences because of its intimate partnership with machines in creating new modes of social interaction, new industries, a computational branch of nearly every science and academic subject [42] and a pathway to an extremely deep and autocatalytic connection between human intelligence and *machine intelligence* [120]. We look briefly at some of the most important ideas in the early history of how computer science was enriched by mathematical logic. *The goal is to situate elements of type theory in an historical context and to give high level intuitions and motivations for some of the central concepts of modern type theory and for the role of logic and type theory in the broad reach of computer science.*

3.4 Unifying view

Elements of all of these influences are visible in modern type theories if one knows where to look, as we shall see. Moreover, all three major approaches to foundational questions in mathematics

[6] Church in 1936 and Turing in the same year, independently solved a version of Hilbert's second problem, the Entscheidungsproblem, or Decision Problem.

created versions of the same fundamental ideas expressed in different languages. We reference them through the thoughts of leading figures for each approach. Bertrand Russell believed that *typed logic* [139, 138] was the bedrock foundation for mathematics in which axioms were to be stated. He proposed the notion of a type to express the *range of significance of a proposition*. The range of significance is related to Frege's notion of the *sense* of a proposition [65].

David Hilbert thought that a *completely formal axiomatic system* was the bedrock of mathematical certainty if the axioms of a *formal theory* could be proved consistent using only *finitistic constructive reasoning* about symbols [86, 87]. The type theories are *completely formal* in the very precise sense of being implemented in a computer system. L.E.J. Brouwer believed that logic rests on the *mental constructions* that we use to understand propositions and come to believe them [25, 26]. Nowadays that sort of justification is achieved using the *propositions as types principle* also called the *Curry-Howard isomorphism*. This is still an evolving principle, and we might find even better names for it. The mental constructions now include computations (from the "minds of machines"). This idea is related to Russell's primacy of *typed logics*.

After visiting Brouwer, Kleene proposed a very precise formulation of Brouwer's ideas. By 1945 he called his approach *recursive realizability* [100, 99]. Kleene's approach for constructive number theory and later for constructive analysis was based on Church's thesis. By 1965 he had discovered a fundamental result showing that Brouwer's Fan Theorem is not consistent with Church's Thesis. Very early on, Curry [51] also noticed a connection between his combinatory calculus and Brouwer's notion of truth, and Howard made this connection very precise by 1968 [92] and finally published his ideas in 1980 [92] leading to the Curry-Howard correspondence. Moreover, the great Dutch mathematician N.G. de Bruijn learned of these ideas from Heyting and applied them to classical classical logic in his famous *Automath system* [52].

Before 1970 Dana Scott had sketched a theory of *constructive validity* based on these ideas [141]. By 1972 Per Martin-Löf organized the ideas into his *intuitionistic type theory* (ITT) [113]. Meanwhile the author had proposed in 1971 using constructive mathematics based on realizability as a programming logic [38], and as early as 1979 J. Bates had produced a working version of these ideas [14] that eventually became λ-PRL [16], the predecessor of the Nuprl proof assistant [44]. Now there are entire books devoted to the topic by title [55, 144].

4 Lecture 2: Foundational Matters

As we try to imagine a foundational theory for computer science, it is important to understand the nature of that discipline as an intellectual pursuit as well as a transformative technology. One way to gain a perspective is from the history of the discipline as we have noted above. There is sufficient history already mentioned to appreciate that computer science has the character of a *meta-science* in the sense of Hilbert (and perhaps Brouwer as well if we can rely on his private notes). Core computing technology is about designing programming languages, software systems and domain specific formalisms and implementing them. A great deal of mathematics has grown up around these activities, and it has guided the implementation of a large variety of core artifacts such as compilers, programming languages, browsers, networks, operating systems, database systems, symbolic computing systems, proof assistants, language translators, and so forth.

4.1 Computer science as a meta-science

Typically a formal theory is introduced in the context of an *informal metatheory*. In the case of theories such as constructive type theory that are designed to be implemented, certain elements of the theory are formalized by the programming language which implements them. It is possible to go even further and use another implemented formal type theory as the meta-language. This has been done for *Constructive Type Theory* [5] where the metalanguage is the Coq type theory [18, 29] and the implementation is by Anand and Rahli [10]. We abbreviate this theory as CTT14; it is implemented in Common Lisp as its mother language. Details of the implementation are on line [11].

The idea of a *meta-language* was introduced by Hilbert and also by Brouwer who called such languages *second-level* mathematics. The idea was developed in detail by Hilbert and his associates.

This approach to defining formal theories was adopted by Edinburgh LCF [77], and the meta language there included the programming language ML, standing for Meta Language. This became the basis for the highly successful family of ML programming languages including Standard ML [122, 129], OCaml [123], and other less well known variants.

4.2 Computer science and its dynamic technologies

Many of the practical applications of proof assistants are in support of important deployed software systems. This area of work raises new foundational and methodological questions. The abstract mentioned examples of constructive type theory providing a formal semantics for programming language [50], compilers [106], operating systems [143], and distributed systems [133, 140].

This kind of work is quite different from formally proving fundamental results in mathematics. The difference arises from the fact that all of these practical applications share the property that the languages and systems about which formal properties are established change over time and the environments in which they operate change as well. Therefore many of the fundamental assumptions in the specification of the artifact change over time. This means that that the verifications must be redone and updated even though the original authors are no longer available to do the work or even consult about it. So far there is no convincing technology to deal with this important issue. Once it is developed, it will also be valuable in other areas of science and engineering where systems, books, course materials and so forth must be reliably updated.

There is new work on this very important problem. It will be an integral part of maintaining a dynamic library of formalized mathematics and science. There are interesting hints about the depth of this problem in early work on *formal digital libraries* [107, 36, 8].

4.3 Presenting constructive type theory

There are many ways to approach the study of constructive type theory. Already we have seen fit to first introduce the notion of a metatheory and then to make the case that questions about explaining and implementing any theory belong to its metatheory. We also note that historically, before any formal theory has been created, we have conceived an informal one that fulfills some purpose worthy of further significant further effort.

The context stressed in these notes is that mathematicians and philosophers had created by the

mid 1900's a very rich and satisfactory foundational theory for mathematics, namely set theory. Moreover, the group of French mathematicians known as Bourbaki had announced a plan to publish over a several year span a definitive encyclopedic account of modern mathematics formulated in terms of set theory [23] to be called *The Elements of Mathematics* [22]. There were several interesting features of this program, one being that the authors were a somewhat secretive collective of eminent mathematicians who tried to reach consensus on the topics they presented. *One of their aims was to demonstrate that set theory was an adequate foundation by using it to cover every important topic.* One of their decisions was not to completely formalize the theory. (Another interesting decision was to use Hilbert's epsilon operator as part of the underlying informal logic.)

Much later, the Mizar group [155] led by Wojciech Trybulec created the *Journal of Formalized Mathematics* based on Tarski-Grothendieck set theory [154], a generalization of ZFC. Their goal was to follow the program of N. G. de Bruijn [52, 126] and create completely formal proofs that would be checked by a computer program before they were published in the journal. It is noteworthy that de Bruijn's program was based on type theory rather than set theory, with the idea that sets were one kind of type important for mathematics, but not the only one. Moreover, de Bruijn knew a great deal about intuitionistic mathematics, and was not only open to constructive ideas, but the Automath language was based on the lambda calculus and embraced ideas from intuitionistic mathematics, including the interpretation of propositions as types.

At about the same time as de Bruijn's work, computer scientists were inspired to build *programming logics* to formalize reasoning about computer programs as a way to avoid dangerous and costly errors in deployed software. This work was inspired by the writings of C. A. R. Hoare [89, 90, 91] and Edgar W. Dijkstra [56], John Reynolds [135], David Gries [82], Kleene [103], and others. Among the first implemented systems were PLCV at Cornell [46, 45] and Pascal based systems at Stanford [108] and Texas [75, 74]. The Cornell systems used constructive logic to take advantage of the strong connection between constructive proofs and programs stressed by this author [38, 39]. In PLCV programs were essential elements of proofs, and one could view programs as algorithmic proofs in the logic.

In the same period, both mathematicians and computer scientists were very inspired by the work of Errett Bishop [20, 21] showing that constructive mathematics could be presented in a way that was *readable either classically or constructively.* The constructive proofs were a bit more detailed than those presented in classical textbooks, but as a compensating benefit, they carried strong computational meaning. The Cornell computer scientists were keen on making this computational content fully useable [109]. This goal was part of the inspiration for formulating a foundational theory adequate for Bishop's mathematics. The first attempt was to extend the programming logics along the lines sketched in 1971. This effort was advanced by studying the work of Per Martin-Löf [112] and later [113]. Building on the PhD thesis of Joseph Bates [14], we started to implement the LambdaPRL system [16] and then by 1982 with NSF funding, Nuprl [44], operational by 1984 and still being used.

4.4 Where to start, Language, Concepts, Categories, or Computation?

Readers expect concepts to be presented in a particular linear order, later ones depending on earlier ones. The ordering implies that some notions are more primitive than others. But it's often not clear when a particular order is necessary. In the case of constructive type theory, certain key ideas are concrete, and they must be presented in terms of a concrete syntax. In that instance, the

syntax must come first. This is also the case with programming languages.

Other concepts are less dependent on syntax and on order of presentation and on the exact form of rules. In this sense they are more abstract and more general. The notion of a set has this character. It does not depend intrinsically on a specific syntax, although the theory ZFC does depend on knowledge of first-order logic (FOL), and that entails certain syntactic details.[7] A commonly held view among mathematicians is that set theory is part of the *Platonic reality* behind mathematics. A mathematician strives to grasp this reality by whatever means. Although Hilbert held this view of reality, his method of securing mathematics depended on his idea of a *formal system* that allows us to also view mathematics as a precisely defined game, like chess.

To Hilbert, the important feature of the game of mathematics is that it should be impossible to derive a contradiction. Proving this feature of a formal system adequate for all of mathematics such as set theory or the type theory of *Principia Mathematica* and doing it using extremely elementary constructive methods (even what Hilbert called *finitistic methods*) is Hilbert's route to a secure foundation. Gödel's incompleteness theorem showed that this goal is not achievable as initially conceived [71]. Gentzen [67] and others suggested using stronger logical methods in the metatheory as a way to further investigation of Hilbert's program. The appeal of using constructive methods for this task was strong because they seemed safer and appropriate for the task as Herbrand also demonstrated [73].

Computational versus abstract approaches Constructive type theory is built on computation and can be explained in computational terms. That explanation requires a specific syntax and careful rules for transforming one expression into another. *If we start with the computational aspect of type theory, then we are immediately in the realm of computer science and programming languages.* That is how we will eventually present certain the details.

For many subjects in mathematics and computer science, it is possible to be more abstract and avoid computation. We can draw pictures for geometry and topology. Indeed, one geometric definition of π is the ratio of the circumference of a circle to its diameter, and it does not refer at all to computation.[8] On the other hand, we are nowadays fascinated by the fact that *pi* is also a constructive real number, and we can compute it to *arbitrary decimal precision*. It has been done to many millions of digits! But its definition is something we can imagine in Plato's ideal world.

We discover in this Platonic world other interesting abstract facts related to computation such as the asymptotic running time of the problem of computing the lower envelope of Davenport-Schinzel sequences, known to be on the order of the *inverse Ackermann function*, denoted α. So even features related to *primitive recursive* function definitions show up in the Platonic world.[9] The geometrically inspired insights of Grothendieck also show up naturally in type theory [145], in the definition of universes, and in set theory, specifically *Tarski-Grothendieck Set Theory* [154, 145]. This set theory is formalized in the Mizar project [137]. It is a generalization of ZFC that allows Grothendieck universes as sets. In these universes, all the subsets of a set are members.

We can study simple algebraic structures such as groups and rings using a variety of nota-

[7] This dependence on FOL was something that Zermelo stove to avoid, but Fraenkel's views prevailed, and the dependence on FOL [88] is standard in modern set theory, especially for two axioms, comprehension and replacement. ZFC thus presupposes elements of first-order logic in the foundations of mathematics.

[8] Another important definition is that *pi* is the first positive root of the sine function.

[9] Ackermann's function was used to show that there are recursive definitions that are not primitive recursive, and his function is a simple way to majorize any primitive recursive function.

tions. We study computational complexity within certain bounds that allow several different precise models of computation to justify the same asymptotic complexity results. Moreover, many of the landmark results and open questions, such as does P = NP, and the possibility of feasible probabilistically checkable proofs, make sense for a wide variety of computing models and formalisms. On the other hand, when we want to formalize these results, we must pick a particular computation system and define it in complete detail. Ideally we want to execute these algorithms as part of the process of understanding them and explaining them, so the details matter and must be formalized and implemented.

A good methodology for the *computational foundations of type theory* are the methods used by computer scientists. One of the earliest and most relevant examples is the approach of John McCarthy and his students who defined the programming language Lisp [117, 116] based on Church's lambda calculus [34]. The lambda calculus is a programming notation adequate to express all computable functions. On the other hand, Lisp is almost a completely untyped programming language, and so is the untyped lambda calculus. Thus the comparison only goes part way in explaining type theory![10] We will turn to this computational approach after we have discussed other options.

Contrast with set theory Mathematicians discovered early in the 20th century, when they focused on building the logical foundations of their field, that the language of set theory is adequate [166, 64, 62, 63]. *Set theory does not depend on computation.* That means that presentations can be much less strict about syntax. Indeed, the foundations of set theory are not required to say anything about computation, not even numerical computation. *Computation is explicit and important only in the meta-language of set theory.* In the object theory itself, we could only find some semblance of computation in the interpretation of the logic used to express the axioms. This logic is first-order logic, and that is an important feature of the set-theoretic foundation. So normally only the details of first-order logic might shed light on how computation could enter the foundations of mathematics in its object theory. But normally this possibility is never discussed. Below is a sample introduction to set theory. It stresses the concepts deemed most important and striking about this parsimonious foundation.

Quote from Kenneth Kunen's book *Set Theory* [104] (italics in quote are mine):

"Set theory is the foundation of mathematics. All mathematical concepts are defined in terms of the primitive notions of set and membership. In axiomatic set theory we formulate a few simple axioms about these primitive notions in an attempt to capture the basic obviously true set-theoretic principles. *From such axioms, all known mathematics may be derived.* However, there are some questions which the axioms fail to settle, and that failure is the subject of this book."

The syntax for denoting sets could be quite vague since it is not necessary or even sensible to do serious computation with sets – they were not designed for that purpose. Lists and bags could be used to approximate sets, and they have clear computational meaning. Peter Aczel eventually found a systematic way to embed set theory into Martin-Löf's intuitionistic type theory [2, 3, 4]. His approach was implemented by Hickey in Meta-PRL [85, 165]. We do not discuss that approach here. We offer a different comparison that does not presuppose understanding the details of Martin-Löf's

[10] John McCarthy was a graduate student at Princeton, and it is clear from his work that he learned the lambda calculus in that environment and used it as a model for Lisp. On the other hand, one suspects that he was uninterested in types, perhaps because he did not study that aspect of Church's work. The historians might sort that out at some point.

intuitionistic type theory nor CTT.

Mathematicians also discovered that they could explain many of their foundational ideas using category theory. Here too it is not necessary to carry out numerical computations, instead one can draw arrows and chase around diagrams, more as in geometry but now even more abstractly. It is possible to present constructive type theory in a manner closer to set theory or in a manner closer to category theory. There are excellent textbooks and articles in this style [142, 17, 95, 96], and the connections to type theory can be explained abstractly as well, defining the topos of sets [105, 110]. Recent work on the connections between homotopy theory and type theory is a promising approach to creating a more abstract account of type theory that is an adequate foundation for both mathematics and computer science [131]. On the other hand, existing rich type theories such as CTT can do a good job.

4.5 Starting with types – the type of sets

Given that our aim is to use type theory as a foundational theory, is there a way to use types that is close to the abstract approach offered by set theory? The closest approximation to starting with sets is starting with types rather than starting with a programming language and its computing rules. We can discuss the architecture of type theory both in parallel to and in contrast with that of set theory. In set theory, we start with the empty set ϕ and say that it has no elements. We define all other sets as sets of sets. The collection of all sets cannot itself be a set without risking paradox. So the meta-theory of set theory allows us to think of the collection of all the sets we plan to build. But that collection is called a *class* not a set. The idea is that the collection of all sets is too big to be a set itself.

Comparing types and sets In the case of type theory, if we start by forming the type of sets, then we will face a similar size problem. Is there a type of all sets? If so, is it too big to be a type itself? Do we need types corresponding to classes as well? In type theory there is a natural hierarchy. Type theory includes the notion of *large types* just as set theory has large sets, called classes. Large types that are closed under the type constructors are called *universes.*

In set theory, we start building all other sets from the *empty set*, denoted ϕ. The set whose only element is the empty set is written as $\{\phi\}$. It has precisely one element. Just below we use the membership relation, $\in$, to define this equality relation. We say $\phi \in \{\phi\}$. This membership relation is a *primitive relation* of set theory along with the equality that can be defined from it. By making equality a primitive of the theory, the rules of first-order logic with equality guarantee that *all definitions and operations respect equality.*

In type theory, there is an empty type, it is denoted `Void`. It has no elements. We can also form a type whose only element is `Void`. It requires that we mention the first universe, forming $\{x : \mathbb{U}_1 | x = \texttt{Void in } \mathbb{U}_1\}$. The existing universe structure modeled on Martin-Löf's ITT [113] is not sufficient to support modeling set theory in this way. It is not clear that an extensive enrichment of the universe structure is worthwhile for constructive mathematics and computer science, and there are more elementary alternatives for capturing set theory in type theory. We examine these briefly.

Defining sets in type theory We consider briefly adding to type theory the type *Set*. It is important to note right at the start that there is an *equality relation* on every type, thus there is

an equality relation on *Set*. We want it to be the natural *extensional equality*. Two sets x_1 and x_2 are *extensionally equal* if and only if they have the same elements. So this definition tells us that sets have no repeated elements in that the set $\{\phi, \phi\}$ is equal to the set $\{\phi\}$. In some textbooks the authors say explicitly that sets do not have repeated elements. Collections that allow repetitions are called *bags*.[11] There is a sense that this equality relation is essential to mathematics. We call the equality *extensional* because it depends on the elements of the set, i.e. the set's *extent* and not on the way the set is defined, its *intensional* structure. The two descriptions above are clearly not equal, they use different symbols and one explicitly repeats an element. So the description of the two sets is different. The description is called an *intensional feature* of the set. But in normal foundational set theory, we do not take account of the description when we axiomatize the equality relation.

Set theory must also provide operations for building sets. But "building" is a synonym for defining sets. We can't really build the most interesting sets, the so called *infinite* ones. Perhaps the next most key axiom is the one that says there is a particular infinite set. It arises from the process of defining a certain sequence of sets that has the following structure. We think of ϕ as the set "zero"; denote it as 0. Then we form the set $\{\phi\}$. We can think of this as the set $\{0\}$, and it has precisely one element the empty set. Call this the set "one" and abbreviate it as 1.

Next form the set $\{\phi, \{\phi\}\}$. This set has exactly two elements, 0 and 1. We abbreviate this set as 2. Now form the set $\{\phi, \{\phi\}, \{\phi, \{\phi\}\}\}$. It has three elements, and we abbreviate it as 3. We see that the pattern is to build the sets, 0, $\{0\}$, $\{0, 1\}$, $\{0, 1, 2\}$, $\{0, 1, 2, 3\}$, and so forth. We are building *representations of the natural numbers*, 0,1,2,3,4,.... as sets. We can clearly keep doing this. Set theory provides a set called ω that is this infinite set. The axiom is called appropriately the Axiom of Infinity.

Set theory is organized to have axioms that justify specific methods for building sets. It requires an axiom that says we can form the *union* of two sets. We use the union to keep extending the sequence. This process does not stop with the infinite set ω since we can *imagine* extending it by the same process, we add ω to the collection and keep going. It is tempting to say that we are "building" these sets, but the process we have just described is not actually building anything because we as humans nor we with our machines cannot actually carry out a process "forever". In type theory, we will examine what we and our computational partners can actually construct in our minds, and we will create our mathematics around those *mental constructions*. Hence the name *constructive* type theory.

Set theory postulates the *Power Set* of a set, that is the set of all subsets of the set. The power set of ω, sometimes written 2^ω or $\mathcal{P}(\omega)$ is the set of all subsets of ω. We can form the power set of any set. One way to define interesting subsets of a set is using the comprehension axiom scheme. It allows sets of the form $\{x : S | \phi(x)\}$ where $\phi(x)$ is a first-order formula in the variable x that defines particular elements of S. The only kind of subset we can define in the set theory ZFC is one that can be described by a first-order formula in the language of sets. The CTT type theory has a version of comprehension that is written in nearly the same way, namely $\{x : T | P(x)\}$ where P is a propositional function over the type T.

[11] Some textbooks then discuss why set theory is not based on collections that allow repetition, a more fundamental notion. This shows that even for set theory, there are some arbitrary decisions made in its axiomatization and design. There is basically universal agreement to use sets for mathematics and not bags, although we will see that in type theory, bags are very critical.

Set theoretic functions as abstractions of rules In CTT the type of integers $\mathbb{Z}$ is defined in terms of computation, and the natural numbers are $\{z : \mathbb{Z} \mid 0 \leq z\}$. Let this type be abbreviated as $\mathbb{N}$. The computable functions of one argument on the natural numbers belong to the type $\mathbb{N} \to \mathbb{N}$. Their canonical form is $\lambda(x.b(x))$ where we know that for any n in the type $\mathbb{N}$, we know that $b(n)$ reduces in a finite number of steps to a canonical natural number m. Given a specific function f, we can now define an abstraction of it into the set-theoretic functions on ω by defining a computable mapping of $\mathbb{N}$ into ω in the obvious way, and defining the set of ordered pairs to be the injection of $pair(n, f(n))$ into a single valued relation in Set.

So we can carry out computations in the type theory and inject the results into the abstract set-theoretic model of functions.

The set theoretic definition of a function is given in terms of relations, which are sets of ordered pairs. So the set of functions from ω to ω is the set of relations F on $\omega \times \omega$ such that if $F(n, m)$ and $F(n, m')$, then $m = m'$ *in* ω, and such that for every n in ω, there is a unique m in ω such that $F(n, m)$. It is this last clause that allows us to make a connection to computation through the interpretation of the first-order formula $\forall n : \omega.\exists m : \omega.F(n, m)$. The type theory realizer for this first-order formula is a function $\lambda(n.f(n))$. In order to provide a computational interpretation of this function, we need to be clear about the type theoretic semantics of first order logic. We can use virtual evidence semantics [43] to accomplish this. That semantics uses a computational interpretation of functions, and yet it is adequate to express classical logic because some of the evidence can be virtual. Nevertheless, virtual evidence has a computational meaning. When the body of the function, $f(n)$ is built from the standard type theory constructors, then the function is computable.

4.6 Starting with computation

The primitive objects of this calculus are λ-terms. Among them are *abstractions* written as $\lambda(x.b(x))$. The letter x is the *binding variable* of the abstraction and the subexpression $b(x)$ is a λ-term which might contain the letter x and is called the body of the λ-term. The syntax also shows the *scope* of the binding variable x, which is exactly the term b. We say that a variable x of a λ-term is bound if it is in the scope of a binding operator $\lambda(x.)$.

The other kind of λ-term is called an *application*; it has the form $ap(f; a)$ where f and a are λ-terms. We say that f is in the *function position* and a is in the *argument position*. In the example $\lambda(x.\lambda(y.ap(y; x)))$, in the subterm $\lambda(y.ap(y; x))$ the variable x is free and y is bound, by $\lambda(y.)$. The term $\lambda(y.ap(y; x))$ is not considered a value because it has a free occurrence of x. A λ-term with no free variables is called *closed*. The *values* are the closed terms of the form $\lambda(x.b(x))$. Values are also called *canonical expressions*, meaning that they cannot be reduced by the computation rules.

The applications are non-canonical terms. They can be reduced by the computation rules. The rule for evaluating $ap(f; a)$ is to first evaluate the expression f. This process may not terminate, in which case the application does not have a value. If the evaluation of f results in a value, say $\lambda(x.b(x))$, then there are two standard choices for how the evaluation might continue, called *lazy evaluation* (or call by name) or *eager evaluation* (or call by value). In lazy evaluation, the expression a is not evaluated, but rather is substituted for the variable x in expression $b(x)$.

In eager evaluation, the expression a is evaluated. This process might not result in a value, in which case the application does not have a value. If a evaluates to a value, say val, then this is

substituted for x in $b(val)$, and the result is evaluated.

This evaluation process is typical for the applied lambda calculus. We can see from attempting to evaluate the following expression, $ap(\lambda(ap(x;x));\lambda(x.ap(x;x))$ that the reduction rule produces the exact same term, $ap(\lambda(ap(x;x));\lambda(x.x))$. This term is sometimes called Ω, and it simply diverges under the computation rules.

It was a remarkable discovery by Church, Kleene, and Rosser that with this simple calculus of functions and with the appropriate coding of natural numbers as λ-terms, they could define any computable function they could imagine. Once Turing defined his Turing machines, he was able to prove that he could compute any lambda computable function, and Kleene proved the converse. Based on this research Church was led to conjecture his famous thesis that λ-terms could compute any effectively computable function of the natural numbers.

KEY PAPERS [10, 9, 132, 19]

5 Types unique to CTT and their value

In this section we examine some of the types that have been added to CTT over the years and implemented in Nuprl. We clear up some history of these types as well.

5.1 Quotient types, set types, partial types, and dependent intersection types

Right from the beginning, say in CTT84, the type theory that was implemented by Nuprl in 1984 and presented in the the 1986 book, *Implementing Mathematics with the Nuprl Proof Development System* [44] included types that did not appear in other versions of constructive type theory such as Martin-Löf's ITT82 [113], Coq88 the Calculus of Constructions [47, 48], ALF [111], and the book *Programming in Martin-Löf's Type Theory* [128]. The recursive types were developed in Mendler's thesis [118, 119] and were investigated by Coquand and Paulin-Mohring [49], and a version appeared centrally in Coq. The set type and the quotient type came from an article in the proceedings of a 1983 conference *Topics in the Theory of Computation'83* [40]. They have not been adopted in other type theories, but were used in the Edinburgh version of CTT84 [27, 28], the Oyster-Clam system (which had a "pearl" in it).

Quotient Type The *quotient type* allows defining a new equality over a type. For example, to define the integers modulo 5 over $\mathbb{Z}$ we define the equality modulo five relation, say $Eq_5(x,y)$ is defined as $\exists n:\mathbb{N}.x-y=n\times 5$. Then the type expression $\mathbb{Z}//x,y.Eq_5(x,y)$ denotes the type whose elements are integers and whose equality is the equivalence relation just defined. This is a simpler explanation than defining the elements to be "equivalence classes modulo 5." It is very easy to compute over this type, and we can also inject the elements into the equivalence classes that would be defined in the standard way over *Set*. This shows that the quotient type is a computational implementation of equivalence classes.

Set Type The *set type* looks very much like the definition of a subset in ZFC. We write it as $\{x:A|P(x)\}$, and its elements are those members of the type A for which we have evidence that the predicate P holds. The important feature of the set type is that the evidence is "hidden" in the

sense that in order to claim that an element a of the type A belongs to $\{x : A|P(x)\}$, we must prove $P(a)$. From the proof we can extract evidence pa that a has the required property. However, that evidence is not stored with the type. One reason for hiding the evidence is a matter of efficiency in presenting evidence. By hiding this evidence, we are saying that it will not be needed in further computation. If we wanted to keep the evidence, we would use the product type, $x : A \times P(x)$ whose elements are pairs, $pair(a; p)$ where a belongs to A and p belongs to $P(a)$.

Partial Types The partial types are used to capture the fundamental notion of the *partial recursive functions*. Programming languages are built on this notion in the sense that a program f which accepts inputs x of type A and computes outputs of type B (or type $B(x)$ if we use dependent types) is actually computing a *partial function* f which on a specific input a of type A produces an output $f(a)$ of type $B(a)$ *provided it terminates*. On some inputs it might not terminate. Normal type checkers do not attempt to establish termination; that is a non-computable task. One way to express that $f(a)$ might diverge is to name diverging computations and treat them as partial objects. One convenient mode of expression is to name these diverging computations, say with the term $\perp$.[12] We say that partial types have $\perp$ as a member, but a member with no canonical form. For any type A it is possible to write expression that are in this type if and only if they terminate. Expressions with this property are said to be in the type $\bar{A}$.

Intersection Types Dependent intersection types were introduced by Kopylov in 2000 as a generalization of the intersection types studied by Reynolds [136] and Pierce [130] and used in increasingly interesting ways thereafter [101, 102, 35]. They lead naturally to the notion of polymorphic logics and *uniform validity* [35] mentioned earlier. They can be used to give an elegant definition of dependent records, and they led to Kopylov's definition of objects in CTT [102].

Kopylov's investigations established quite clearly that many types have a logical interpretation and that the propositions-as-types correspondence is as open-ended as our imaginations. It is easy to imagine that if type theory serves well as a foundation for computer science, it will also play a significant role in the foundations of mathematics. We see signs of this in the efforts to formulate a Homotopy Type Theory for mathematics.

A Formal Semantics for CTT From the beginning, the CTT type theory was based on a semantic definition of types as *partial equivalence relations* as defined by Stuart F. Allen [6, 7] and given an alternative definition by Robert Harper [83]. Allen also provided an induction principle adequate for proving correctness of the rules for CTT and for Martin-löf type theory [113], but he did not make it formal as was done by Dybjer in 2000 [58]. This rigorous formal semantics for CTT combined with Howe's fundamental work on *computational equality* [93] and Crary's formalization of partial types [50] made it possible for Anand and Rahli to give a completely formal account of the CTT semantics as of 2014 in Coq [10] and prove the key rules formally correct.

[12] The typical diverging value in CTT is $fix(\lambda(x.x))$.

References

[1] Martin Abadi and Rustan M. Leino. A logic of object-oriented programs. In Nachum Dershowitz, editor, *Verification: Theory and Practice, Essay dedicated to Zohar Manna on the occasion of his 64th birthday*, Lecture Notes in Computer Science, pages 11–41. Springer, 2003.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In Gries [81], pages 215–224.

[7] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[8] Stuart F. Allen, Robert L. Constable, and Lori Lorigo. Using formal reference to enhance authority and integrity in online mathematical texts. *Journal of Electronic Publishing*, 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 Vincent Rahli. Towards a formally verified proof assistant. In *International Conference on Interactive Theorem Proving*, pages 95–197, 2014.

[11] Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. Technical report, Department of Computer Science, Cornell University, 2014.

[12] Aristotle. *Organon*. Approx. 350 BCE.

[13] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011.

[14] J. L. Bates. *A Logic for Correct Program Development*. PhD thesis, Cornell University, 1979.

[15] J. L. Bates. A logic for correct program development. Technical Report TR81-455, Cornell University, 1981. Revision of Cornell University Ph.D. thesis submitted August 1979.

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

[17] John L. Bell. *Toposes and Local Set Theories*, volume 14 of *Oxford Logic Guides*. Oxford University Press, Oxford, 1988.

[18] 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.

[19] Mark Bickford, Robert Constable, and Vincent Rahli. The Logic of Events, a framework to reason about distributed systems. Computing and Information Science Technical Reports http://hdl.handle.net/1813/28695, Cornell University, Ithaca, NY, 2012.

[20] E. Bishop. *Foundations of Constructive Analysis*. McGraw Hill, NY, 1967.

[21] E. Bishop. Mathematics as a numerical language. In *Intuitionism and Proof Theory*, pages 53–71. North-Holland, NY, 1970.

[22] N. Bourbaki. *Elements of Mathematics, Algebra*, volume 1. Addison-Wesley, Reading, MA, 1968.

[23] N. Bourbaki. *Elements of Mathematics, Theory of Sets*. Addison-Wesley, Reading, MA, 1968.

[24] Edwin Brady. Idris:systems programming meets full dependent types. In *Programming Languages meets Program Verification, PLPV 2011*, pages 43–54. ACM, 2011.

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

[26] L.E.J. Brouwer and A. Heyting. *Collected Works: Philosophy and foundations of mathematics, edited by A. Heyting*. Collected Works. North-Holland Pub. Co., 1976.

[27] 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.

[28] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The oyster-clam system. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 647–648, London, UK, 1990. Springer-Verlag.

[29] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, Cambridge, MA, 2013.

[30] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics, second series*, 33:346–366, 1932.

[31] Alonzo Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1:40–41, 1936.

[32] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Math*, 58:345–363, 1936.

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

[34] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1941.

[35] Robert Constable and Mark Bickford. Intuitionistic Completeness of First-Order Logic. *Annals of Pure and Applied Logic*, 165(1):164–198, January 2014.

[36] Robert Constable, Richard Eaton, Jon Kleinberg, and Lori Lorigo. A graph-based approach towards discerning inherent structures in a digital library of formal mathematics. 3119:220–235, 2004.

[37] 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.

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

[39] Robert L. Constable. A Constructive Pogramming Logic. In Bruce Gilchrist, editor, *Information Processing77*, pages 733 – 738. North-Holland, August 1977.

[40] Robert L. Constable. Constructive mathematics as a programming logic I: Some principles of theory. In *Annals of Mathematics*, volume 24, pages 21–37. Elsevier Science Publishers, B.V. (North-Holland), 1985. Reprinted from *Topics in the Theory of Computation*, Selected Papers of the International Conference on Foundations of Computation Theory, FCT '83.

[41] 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.

[42] Robert L. Constable. Transforming the academy: Knowledge formation in the age of digital information. *Physica Plus*, 9:428 – 469, 2007.

[43] 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.

[44] 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.

[45] 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.

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

[47] Thierry Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[48] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Lf and Grigori Mints, editors, *Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1988.

[49] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types, preliminary version. In *COLOG '88, International Conference on Computer Logic*, volume 417 of *LNCS*, pages 50–66. Springer, 1990.

[50] Karl Crary. *Type–Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, Ithaca, NY, August 1998.

[51] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic, Volume I*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958.

[52] N. G. de Bruijn. The mathematical language Automath: its usage and some of its extensions. In J. P. Seldin and J. R. Hindley, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer-Verlag, 1970.

[53] N. G. de Bruijn. Checking mathematics with computer assistance. *Notices of the American Mathematical Society*, 38(1):8–15, January 1991. Computers and Mathematics Column.

[54] N. G. de Bruijn. The mathematical vernacular, a language for mathematics with typed sets. In R. P. Nederpelt, J. H. Geuvers, and R. C. De Vrijer, editors, *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, pages 865–935. Elsevier, Amsterdam, 1994.

[55] Ph. de Groote. *The Curry Howard Isomorphism*. Academia, Louvain-La-Neuve, 1995.

[56] Edgar W. Dijkstra. Notes on Structured Programming. In *Structured Programming*, pages 1–82. Academic Press, New York, 1972.

[57] Michael Dummett. *Frege Philosophy of Mathematics*. Harvard University Press, Cambridge, MA, 1991.

[58] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Log.*, 65(2):525–549, 2000.

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

[60] William M. Farmer. A partial functions version of church's simple theory of types. *The Journal of Symbolic Logic*, 55(3):1269–1291, September 1990.

[61] Solomon Feferman et al., editors. *Kurt Gödel Collected Works*, volume 1. Oxford University Press, Oxford, Clarendon Press, New York, 1986.

[62] 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.

[63] A. A. Fraenkel, Y. Bar-Hillel, and A. Levy. *Foundations of Set Theory*, volume 67 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 2nd edition, 1984.

[64] Adolf Fraenkel. Untersuchungen über die grundlagen der mengenlehre. *Mathematishce Zeitschrift*, Vol.22:250, 1925.

[65] Gottlob Frege. Begriffsschrift, a formula language, modeled upon that for arithmetic for pure thought. In van Heijenoort [158], pages 1–82.

[66] Harvey Friedman. Classically and intuitionistically provably recursive functions. In D. S. Scott and G. H. Muller, editors, *Higher Set Theory*, volume 699 of *Lecture Notes in Mathematics*, pages 21–28. Springer-Verlag, 1978.

[67] Gerhard Gentzen. Investigations into logical deduction (1934). In M. Szalo, editor, *The Collected Paers of Gerhard Gentzen*. North-Holland, Amsterdam, 1969.

[68] J-Y. Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symposium*, pages 63–69. Springer-Verlag, NY, 1971.

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

[70] K. Gödel. Über eine noch nicht benütste erweiterung des finiten standpunktes. *Dialectica*, 12:280–287, 1958.

[71] 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 [158].

[72] Kurt Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. Dover Publications, New York, 1992.

[73] Warren David Goldfarb. *Logical Writings of Jacques Herbrand*. Harvard University Press, 1971.

[74] D. Good. Mechanical proofs about computer programs. In C. A. R. Hoare and J. C. Shepardson, editors, *Mathematical Logic and Programming Languages*, pages 55–75. Springer-Verlag, NY, 1985.

[75] D. Good, R London, and E. Bledsoe. An interactive program verification system. *IEEE Trans. Software Eng.*, pages 1:59–67, 1975.

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

[77] 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.

[78] W. T. Gowers. Rough structure and classification. *Geometric and Functional Analysis*, Special Volume:1–39, 2000.

[79] Johan Georg Granström. *Treatise on Intuitionistic Type Theory.* Springer, 2011.

[80] Cordell C. Green. An application of theorem proving to problem solving. In *IJCAI-69*, pages 219–239, Washington, DC, May 1969.

[81] D. Gries, editor. *Proceedings of the 2^{nd} IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, June 1987.

[82] David Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

[83] Robert Harper. Constructing type systems over an operational semantics. *J. Symbolic Computing*, 14(1):71–84, 1992.

[84] John Harrison. *Handbook of Practical Logic and Automated Reasoning.* Clarendon Press, Oxford, 2009.

[85] Jason J. Hickey. *The MetaPRL Logical Programming Environment.* PhD thesis, Cornell University, Ithaca, NY, January 2001.

[86] David Hilbert. Über da unendliche. *Math Analogue*, 95:161–190, 1926.

[87] David Hilbert. The foundations of mathematics. In van Heijenoort [158], pages 464–479. reprint of 1928 paper.

[88] David Hilbert and Wilhelm Ackermann. *Grundzge der theoretischen Logik.* Springer-Verlag.

[89] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

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

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

[92] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, NY, 1980.

[93] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the 4^{th} IEEE Symposium on Logic in Computer Science*, pages 198–203, Asilomar Conference Center, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[94] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant : A tutorial : Version 6.1. Technical report, INRIA-Rocquencourt, CNRS and ENS Lyon, August 1997.

[95] J. M. E. Hyland and A. M. Pitts. The theory of constructions: categorical semantics and topos-theoretic models. In J. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, pages 137–199. American Mathematical Society, 1989.

[96] B. Jacobs. *Categorical Logic and Type Theory.* Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

[97] Anthony Kenny. *Frege.* Penguin Books, London, 1995.

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

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

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

[101] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of 18th IEEE Symposium on Logic in Computer Science*, pages 86–95, 2003.

[102] Alexei Kopylov. *Type Theoretical Foundations for Data Structures, Classes, and Objects.* PhD thesis, Cornell University, Ithaca, NY, 2004.

[103] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110:366–390, 1994.

[104] Kenneth Kunen. *Set Theory.* College Publications, 2011.

[105] J. Lambek and P. J. Scott. *Introduction to Higher-Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics.* Cambridge University Press, Cambridge, UK, 1986.

[106] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33d ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54. ACM Press, 2006.

[107] Lori Lorigo. *Information Management in the Service of Knowledge and Discovery.* PhD thesis, Cornell University, 2006.

[108] D.C. Luckham, S.M. German, F.W. von Henke, R.A Karp, P.W. Milne, D.C. Oppen, W. Polak, and W.L. Sherlis. Stanford pascal verifier: User's manual. Technical Report STAN-X-79-73, Stanford University, 1979.

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

[110] Saunders MacLane and Ieke Moerdijk. *Sheaves in Geometry and Logic, a First Introduction to Topos Theory.* Springer-Verlag, New York, 1992.

[111] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs. International Workshop TYPES'93*, volume 806 of *Lecture Notes in Computer Science*, pages 213–237. Springer-Verlag, 1994.

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

[113] 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.

[114] Per Martin-Löf. An intuitionistic theory of types. In Giovanni Sambin and Jan M. Smith, editors, *Twenty-Five Years of Constructive Type Theory*, volume 36 of *Oxford Logic Guides*, pages 127–172, Oxford, 1998. Clarendon Press.

[115] J. McCarthy. Recursive functions of symbolic expressions and their computations by machine, part i. *Communications of the ACM*, 3(3):184–195, 1960.

[116] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[117] J. McCarthy et al. *Lisp 1.5 Users Manual*. MIT Press, Cambridge, MA, 1962.

[118] P.F. Mendler. Recursive types and type constraints in second-order lambda calculus. In Gries [81], pages 30–36.

[119] P.F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.

[120] Donald Michie. *Machine Intelligence 3*. American Elsivier, New York, 1968.

[121] Peter Millican and Andy Clark. *The Legacy of Alan Turing, Vol. 1: Machines and Thought*. Oxford University Press, New York, 1996.

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

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

[124] Chetan Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Cornell University, Department of Computer Science, 1990. (TR 90-1151).

[125] Chetan Murthy. An evaluation semantics for classical proofs. In *Proceedings of the 6^{th} Symposium on Logic in Computer Science*, pages 96–109, Vrije University, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[126] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected Papers on Automath*, volume 133 of *Studies in Logic and The Foundations of Mathematics*. Elsevier, Amsterdam, 1994.

[127] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

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

[129] L. C. Paulson. *Standard ML for the Working Programmer*. Cambridge University Press, 1991.

[130] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991. CMU-CS-91-205.

[131] Univalent Foundations Program. *Homotopy Type Theory*. Univalent Foundations Program, 2013.

[132] 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.

[133] Vincent Rahli, Nicolas Schiper, Robbert Van Renesse, Mark Bickford, and Robert L. Constable. A diversified and correct-by-construction broadcast service. In *The 2nd International Workshop on Rigorous Protocol Engineering (WRiPE)*, Austin, TX, October 2012.

[134] Aarne Ranta. *Type-theoretical grammar*. Oxford Science Publications. Clarendon Press, Oxford, England, 1994.

[135] 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.

[136] 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.

[137] Piotr Rudnicki. An overview of the Mizar project. Notes to a talk at the workshop on Types for Proofs and Programs, June 1992.

[138] Bertrand Russell. Mathematical logic as based on a theory of types. *Am. J. Math.*, 30:222–62, 1908.

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

[140] Nicolas Schiper, Vincent Rahli, Robbert Van Renesse, Mark Bickford, and Robert L. Constable. Developing correctly replicated databases using formal tools. In *DSN 2014: The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[141] D. Scott. Constructive Validity. In D. Lacombe M. Laudelt, editor, *Symposium on Automatic Demonstration*, volume 5(3) of *Lecture Notes in Mathematics*, pages 237–275. Springer-Verlag, NY, 1970.

[142] D. Scott. Data types as lattices. *SIAM J. Comput.*, 5:522–87, 1976.

[143] Zhong Shao. Certified software. *Communications of the ACM*, 53:56–66, 2010.

[144] M.H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006.

[145] Thomas Streicher. *Correctness and Completeness of a Categorical Semantics of the Calculus of Constructions*. PhD thesis, Universität Passau, 1988.

[146] Pierre-Yves Strub, Nikhil Swamy, Cedric Fournet, and Juan Chen. Self-certification: Bootstrapping certified typecheckers in F star with Coq. In *Proceedings of the ACM Symposium on Principles on Programming Languages (POPL'12)*, pages 571– 583. ACM, 2012.

[147] W. W. Tait. Intensional interpretation of functionals of finite type. *The Journal of Symbolic Logic*, 32(2):189–212, 1967.

[148] W. W. Tait. Against intuitionism: Constructive mathematics is part of classical mathematics. *Journal of Phil. Logic*, 12:173–195, 1983.

[149] A. S. Takasu. Proofs and programs. In *Proceedings of the Third IBM Symposium on Mathematical Foundations of Computer Science*, page 99 pages. Academic and Scientific Programs, IBM Japan, August 1978.

[150] Alfred Tarski. *The Concept of Truth in Formalized Languages*, pages 152–278. Clarendon Press, Oxford, 1956. In *Logic, Semantics, Meta-Mathematics*.

[151] Anne Sjerp Troelstra. *Metamathematical Investigation of Intuitionistic Mathematics*, volume 344 of *Lecture Notes in Mathematics*. Springer-Verlag, 1973.

[152] A.S. Troelstra. Realizability. In S.R. Buss, editor, *Handbook of Proof Theory*, pages 407 – 473. Elsivier Science, 1998.

[153] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, An Introduction*, volume I, II. North-Holland, Amsterdam, 1988.

[154] Wojciech A. Trybulec. Tarski-Grothendieck Set Theory. *Journal of Formalized Mathematics*, 1, 1989.

[155] Wojciech A. Trybulec. Groups. *Journal of Formalized Mathematics*, 2, 1990. `http://mizar.org/JFM/Vol2/group_1.html`.

[156] A. M. Turing. On computable numbers, with an application to the Entscheidungs problem. In *Proceedings London Math Society*, pages 116–154, 1937.

[157] Mark van Atten. *On Brouwer*. Wadsworth Philosophers Series. Thompson/Wadsworth, Toronto, Canada, 2004.

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

[159] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier and MIT Press, 1990.

[160] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier and MIT Press, 1990.

[161] Walter P. van Stigt. *Brouwer's Intuitionism*. North-Holland, Amsterdam, 1990.

[162] W. Veldman. An intuitionistic completeness theorem for intuitionistic predicate calculus. *Journal of Symbolic Logic*, 41:159–166, 1976.

[163] Valdimir Voevodsky. Notes on type systems. School of Math, IAS, Princeton, NJ, 2011.

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

[165] Xin Yu and Jason J. Hickey. The axiomatization of group theory: An experiment in constructive set theory. In Konrad Slind, editor, *Emerging Trends. Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*. University of Utah, 2004.

[166] Ernst Zermelo. Untersuchungen über die grundlagen der mengenlehre. *Math Ann*, 65:261–281, 1908.