Computational Type Theory

Robert L. Constable

October 13, 2008

Abstract

Computational type theory provides answers to questions such as: What is a type? What is a natural number? How do we compute with types? How are types related to sets? Can types be elements of types? How are data types for numbers, lists, trees, graphs, etc. related to the corresponding notions in mathematics? What is a real number? Are the integers a subtype of the reals? Can we form the type of all possible data types? Do paradoxes arise in formulating a theory of types as they do in formulating a theory of sets, such as the circular idea of the set of all sets or the idea of all sets that do not contain themselves as members? Is there a type of all types?

What is the underlying logic of type theory? Why isn't it the same logic in which standard set theories are axiomatized? What is the origin of the notion of a type? What distinguishes *computational* type theory from other type theories? In computational type theory, is there a type of all computable functions from the integers to the integers? If so, is it the same as the set of *Turing computable functions* from integers to integers? Is there a type of computable functions from any type to any type? Is there a type of the *partial computable functions* from a type to a type? Are there computable functions whose values are types? Do the notations of an *implemented* computational type theory include programs in the usual sense? What does it mean that type theory is a *foundational theory* for both mathematics and computer science? There have been controversies about the foundations of mathematics, does computational type theory resolve any of them, do these controversies impact a foundation for computing theory? This article answers some of these questions and points to literature answering all of them.

1 Type Theories

1.1 Overview

Computational type theory was assembled concept by concept over the course of the 20th century as an explanation of how to compute with the objects of modern mathematics, how to relate them to data types, and how to reason about properties of computations such as termination, structure, and complexity. Among the many building blocks of computational type theory are some mentioned here dating back to Aristotle, Kant, and Leibniz. This account features insights of five notable figures who have had a major impact on this theory in the past forty years and who personally shaped my views and contributions; they are Alonzo Church, N.G. de Bruijn, Errett Bishop, and Per Martin-Löf. Their contributions are cited by name.

A salient feature of computational type theory is that it has been publically implemented and used to do hard work, especially in computer science. What does this mean? To say that a logical theory has been *publically implemented* means that the following were accomplished: 1. every detail of the theory was programmed, creating a *software system*[1] 2. many people used the system to find, check, and publish hundreds of proofs 3. articles and books were published about the formal theory, its system, and how to use it. This kind of *formalization in extremis* became possible only in the 20th century and will be common in the 21st century as such theories advance computer assisted thought.

The scientific work done using an implemented computational type theory (CTT) includes finding new algorithms, building software systems that are *correct-by-construction*, solving open problems in mathematics and computing theory, providing formal semantics for modern programming languages (including modules, dependent records, and objects) and for natural languages, automating many tasks needed to verify and explain protocols, algorithms, and systems, and creating courseware that is grounded in fully formalized and computer checked explanations of key computing concepts. In addition computational type theory sheds light on philosophical disputes in epistemology and the foundations of mathematics.

Computational type theory is distinguished as a publicly implemented *theory of computation* already advancing science and technology. Among

[1]The systems are called by various names such as: theorem provers, provers, proof assistants, proof checkers, proof development systems, proof systems, proof refinement logics, problem solving environments, logical programming environments, logic engines, and so forth.

the other implemented theories are first-order logic, a theory of numbers and lists, set theory, domain theory, and other type theories. Numerous computer systems implement type theories including Agda, Alf, Automath, B-tool, Coq, GDLO, HOL, HOL-Light, Isabelle-HOL, LCF, Lego, MinLog, MetaPRL, Nuprl, PVS, and Twelf. Both Nuprl [Nuprl Book, Nuprl-Home] and MetaPRL [MetaPRL] were used to implement a specific formalization of computational type theory that is referenced here as *CTT*.

1.2 Origins

Computational type theory, like all type theories, is related to a foundational theory for mathematics originating with Bertrand Russell [Russell08a, Russell08b] in 1908 as an attempt to deal with certain contradictions such as *Russell's Paradox* about the set R of all sets that are not members of themselves, denoted $\{x|x \notin x\}$, a circular definition. The paradox arises from asking whether R belongs to R. Russell's theory of *logical types* was designed to prevent so called *vicious circles* in definitions which led to contradictions in mathematical reasoning. *Principia Mathematica* (*PM*) [WR25] is a type theory designed by Bertrand Russell and Alfred North Whitehead to consistently develop classical mathematics. It was never completely formalized and couldn't be implemented because in 1925 there were no computers nor high-level programming languages. *PM* was the prototype for Church's 1940 simplification to the *Simple Theory of Types* (*STT*) [Church40]. A variant of this theory was implemented as HOL [GM93]; it was not designed to formalize all classical mathematics, but rather *applied classical mathematics*. In contrast to *PM*, *STT* is an elegantly small theory whose semantics can either be taken as intuitive or modeled in simple axiomatic set theories such as Zermelo set theory (*Z*).

In *PM* and *STT*, types are used to render the notions of class and *propositional function* free from vicious circles. A type is the *domain of significance* of a propositional function, i.e. those objects which the proposition is about, and that domain can not include the proposition itself. What Russell believed is that there can't be a single universal type of all meaningful objects on which propositions are defined because that type would include the propositions being defined, creating a vicious circle. Instead, the universe must be divided into categories, Aristotle's terminology, or *types*. If $P(x)$ is a propositional function on a type T, then those objects for which it is true form a *class* denoted $\{x|P(x)\}$ or $\{x : T|P(x)\}$. Classes are fundamental in *PM* and are even used to define the natural numbers. For example 0 can be defined as the empty class, and the natural number 1 is defined in paragraph

*52 of *PM* as "the class of all unit classes", that is, the class of all classes which have exactly one element. The natural number 2 is the class of all classes with exactly two elements, and so forth. These definitions can be made non-circular by first defining the idea of a *correspondence*, but we will not go into these details which are now part of the standard mathematical curriculum.

In the HOL implementation of *STT*, the natural numbers are defined more simply as an inductive class over the primitive infinite type of *individuals* denoted *ind*. The other primitive type in HOL consists of the two truth values, *true* and $false$; it is called *bool*. Since *ind* is infinite, it is non-empty, and in HOL an element of *ind* is arbitrarily chosen to be 0 using a built in *choice operator* denoted @. A successor function, *succ*, is defined using the axiom that *ind* is infinite; these two choices allow the definition of the natural numbers as 0, *succ*(0), *succ*(*succ*(0)), etc. The type *num* is defined as a class over *ind* containing these elements. This is a rather abstract definition compared to how we teach young people about numbers. We will see that in computational type theory the numbers are abstract and also innately related to their common decimal numerals.

1.3 Philosophical issues

Principia Mathematica was also designed to demonstrate that mathematics could be reduced to logic, say to *truth values* (Booleans) and higher-order functions over them – specifically to *predicative logic* which avoids the concept of all functions or all propositions. This is one reason that *PM* did not start with a primitive type of natural numbers nor with a type of individuals as in HOL, namely, Russell and Whitehead wanted to derive mathematical concepts such as natural numbers and real numbers from more basic logical concepts. Unfortunately, the core *PM* foundation is inadequate for classical mathematics, even for natural numbers, without an *axiom of reducibility*[2] in the typing rules, which is a way to introduce so called impredicative notions.[3]

It is noteworthy that computational type theory resolves some of the

[2]The axiom of reducibility proved to be unpopular among mathematicians and other logicians. Poincare wrote harshly about it and about the cumbersome nature of PM. There were no computer scientists around to make the case that machines could fill in vast amounts of detail and present a version of PM that would look much more like ordinary mathematics than the raw formalism.

[3]Impredicative typing allows that in defining functions from type α into type β, denoted $\alpha \to \beta$, it is possible to use functions of a higher type, say those from $(\alpha \to \beta) \to (\alpha \to \beta)$. Gödel's 1958 system T used the same impredicative typing structure.

philosophical disputes about the foundations of mathematics – summarized well in Dummett's book on Intuitionism [Dummett77]. For example, the formalized theory *CTT* illustrates David Hilbert's belief that mathematical truth can be reduced to mechanically checkable claims in a formal system, a belief called *Formalism*. Founding *CTT* on the irreducible concepts of natural numbers and computation validates Brouwer's claim, to be discussed below, that these are the fundamental intuitions on which meaningful mathematics rests. These are the beliefs of *Intuitionism* as adjusted by Bishop's *constructivism*. *CTT* also follows Brouwer in equating mathematical truth with proof. Finally, the fact that among the primitive constructions of computational type theory are those that validate the laws of logic shows that logic is inseparable from mathematics, not quite in the way Russell imagined in his belief called *Logicism*, but in a way that makes his notion of type fundamental to both logic and mathematics. Thus through the lens of *CTT* we recognize that each of the major philosophical schools contributed part of the underlying foundations of mathematics and computing. We see the interplay of these philosophical ideas in the next section.

2 Computation and Data Types

2.1 Building computation into the basic primitives

The small set of irreducible primitive notions in Church's Simple Type Theory does not include primitive concepts from *effective mathematics*, *recursive mathematics*, *computational mathematics*, *Intuitionistic mathematics*, *constructive mathematics*, or *computer science* – these labels refer to different ways in which computational ideas have been systematically integrated into mathematical reasoning. Computational ideas have been an integral part of mathematics since the Greeks with their stress on ruler and compass constructions and since the Arabs with their stress on written algorithmic processes executed by humans; this is the tradition of effective mathematics.[4] Computer science is the systematic body of knowledge built up about computing with digital information using a variety of physical machines from personal computers to the Internet.

Intuitionistic mathematics is based on a specific philosophy of mathematics originating circa 1907 with L.E.J.Brouwer [Brouwer75, Heyting71] and related to Kant's understanding of synthetic *a priori* knowledge. A key

[4]Nowadays the term *effective mathematics* refers to ways to evaluate the teaching of mathematics in school, nevertheless inside logic and mathematics, the term has a technical meaning.

principle of this philosophy is that the primitive truths of mathematics are based on innate human intuitions about natural numbers and constructions on them, that logical reasoning is justified by the most general constructions, and that propositions are true only if we can prove them by these general constructions from the primitive truths. This Intuitionsitic mathematics includes among its primitive concepts effective computation on the natural numbers. Brouwer developed this mathematics beyond law-like computation on numbers to include other *mental constructions* encompassing infinite sets and free choice sequences as objects of mathematics. He used free choice sequences to define real numbers (the continuum).[5]

Recursive mathematics includes as primitive computations those that can be done by humans or machines executing the concept of an algorithm formalized by Church and Turing and referenced in the famous *Church-Turing Thesis* from 1936 that effective procedures over the natural numbers are precisely the *Turing computable functions* over the natural numbers. Any computing formalism equivalent to the Turing computable functions is said to be *universal*.

2.2 Constructive mathematics

Constructive mathematics is closely related to effective mathematics, recursive mathematics, and Intuitionistic mathematics. It shares the idea that computational concepts belong to the primitive foundation of mathematics and asserts that to understand recursive mathematics, one must know these intuitions. One of the seminal publications in constructive mathematics is the book *Foundations of Constructive Analysis* by Errett Albert Bishop [Bishop67]. In philosophical remarks in this book, Bishop disagreed with Brouwer about the continuum and defined real numbers using standard data types, in the way that they are now implemented as "infinite precision numbers" in some programming languages.[6] Bishop also believed that constructive mathematics should be consistent with classical mathematics, something Brouwer denied.[7] For example, in Brouwer's theory of

[5] Bishop was to say that Brouwer suspected that the continuum would turn out to be discrete unless he "personally intervened to prevent it" by introducing these free choice sequences. Bishop showed that even without them, the reals were not a discrete type [Bishop67], i.e. a type on which equality is decidable.

[6] For Bishop, a real number r is a computable function from the natural numbers to the rational numbers such that the absolute value of the difference between $r(n)$ and $r(m)$ is less or equal to $1/n + 1/m$.

[7] On page two of his book Bishop wrote "We feel about number the way Kant felt about space. The positive integers and their arithmetic are presupposed by the very nature of

real numbers, all functions from reals to reals are continuous, in Bishop's account they are not, and indeed one can read Bishop's book as a valid account of the classical real numbers, but one that inherently provides information about how to compute with them. One of the goals of computational type theory is to formalize the concepts in Bishop's book and to formalize all of mathematics constructively in such a way that the account can be understood classically yet there is enough information provided to enable practical computation.[8]

Like Intuitionistic and constructive mathematics, computational type theory axiomatizes digital computation on natural numbers and other recursive data types, but it does not adopt the Church-Turing thesis; instead the theory relies on Bishop's insight that unless a person is interested in studying *what cannot be computed*, it is sufficient to specify a natural universal computing formalism and leave it *open-ended* allowing the possibility of extending the underlying programming framework yet preserving the mathematical theorems.[9] Many logical questions can be explored starting from CTT because it can be extended in many directions, toward classical mathematics by adding the axiom P *or* $not P$, toward recursive mathematics by adding the Church-Turing Thesis as a logical principle, toward Intuitionistic mathematics by axiomatizing free choice sequences, and towards domain theory by typing partial functions in the sense of computer science.

In computational type theory, the collection of *all types* is also left open-ended, and it is understood that the types include the standard data types such as numbers, strings, tuples, arrays, lists, trees, etc. (the *first-order types*) as well as the *higher-order* types such as functions from one type to another, infinite lists (streams), infinite trees, and so forth.[10]

---

our intelligence ...."Bishop's constructivist thesis is "In principle, every natural number can be converted to its decimal form by a finite purely routine process."

[8]This approach fails only when it comes to understanding the partial computable functions on a type in the sense that computer scientists prefer versus the sense in which mathematicians prefer.

[9]There is no explicit thesis relating Intuitionistic computation to Turing computation, although Bishop [Bishop67] stated a thesis that a computational account of classical mathematical ideas does not require the Intuitionistic concept of free choice sequences. Kleene related the computable functions of Intuitionistic number theory [Kleene45] and analysis [KV65] to the Turing computable functions.

[10]The delicate point from a foundational point of view is whether to include the *partial types* such as all partially computable functions from a type to a type. Nuprl implements partial types using its concept of a *bar type* [CS93], and that extension provides a constructive domain theory. However, this extension of CTT is not consistent with classical mathematics, just as Intuitionistic mathematics is not. Unlike the situation for computable functions, there is no candidate thesis about what formal concept captures the

2.3 A foundation for computer science

The challenge of computational type theory is to provide an integrated foundational account of all varieties of computational mathematics and to account for computing on all types needed in such mathematics, which will include all types in modern programming languages.[11] To meet that challenge, computational type theory builds in a universal computation system, i.e. a programming language and the computation rules that animate it, and a logic of computation, i.e., the rules of reasoning about algorithms. Thus the specific implemented theory CTT will include a universal programming language and "all" data types. That language can be compared to standard programming languages. Subsets of it resemble languages like O'Caml, $F^\sharp$, and other dialects of ML such as Classic ML [GMW79].

Beyond providing a programming language among the irreducible notions of type theory, CTT contains an automated *programming logic* based on its rules for reasoning about programs and relating them to assertions. This is a key feature of CTT, and we discuss it below. The programming logic subsumes the richest type systems known for programming languages.

In the extension of CTT with a logic of events [BC06], say CTT-E, the programs include *distributed systems*; thus the programming logic applies to virtually all known important classes of programs. This connection to programming explains why CTT has been used extensively in important areas of computer science, namely formal methods and mathematical software engineering. CTT-E also connects to the logic of knowledge and the theory of agents in AI.

CTT is expressive enough to account for the idea of computational complexity of algorithms [CC01]; so it is possible to express the ideas of complexity theory such as the famous $P \stackrel{?}{=} NP$ problem. It can also account for libraries of theorems and thus for systems [ABC+06].

It thus emerges that *computational type theory is a plausible foundation for computer science as well as for computational mathematics*. The famous unsolvability results, such as the unsolvability of the *halting problem*, are formulated in CTT by defining a specific formalism, such as Turing machines, inside the theory. Thus in CTT it has been proved that no Turing machine can solve the halting problem for Turing machines. In cases where

collection of all data types. Gödel compared the concept of computable function, which he said was *absolute* to the concept of a set, which is not. The same comparison holds between computable functions (possibly absolute) and data types (not absolute).

[11] No programming language contains all types needed in mathematics as CTT, CIC, and ITT strive to include.

it is important to express classical mathematical concepts that are not also naturally constructive, there are several accepted ways to do this. One is to assume an *oracle* for solving a problem; such an assumption is consistent because the computation system is open ended. Another approach is to extend the *evidence-based semantics* to classical mathematics that allows us to state the *propositions as types principle* in a classical form [ABC+06]. This idea is used in Morse set theory [Morse65].

3 Logic and Propositions-as-Types

3.1 Constructive existence

It might seem straight forward to achieve the goals of computational type theory by adding computational primitives to Church's Simple Type Theory and thus into HOL because the theories include a fragment of Church's famous *Lambda Calculus* [Chu51], one of the fundamental models of Church-Turing computability and the standard model for the semantics of programming languages. Adding computation rules for computable terms of HOL would define a programming language and a programming logic. The theory *LCF* (*Logic of Computable Functions*) [GMW79] is an effort to add both the computation rules and the logical rules based on domain theory, and it served as a paradigm for *CTT*.

However, the *LCF* approach does not account for the fact that computational mathematics and Intuitionistic mathematics require a stronger notion of mathematical existence and truth than does *STT* or *LCF*. This was the insight of Brouwer. In both *STT* and *LCF*, to prove that an object exists, it is sufficient to show that it is contradictory for it not to exist. But in computational mathematics, and in much of computer science, to say that an object exists, is to say that we know how to construct it and that programs as well as people can manipulate the object. For example, in *STT* it is possible to define a number c that is 0 if *STT* is consistent and 1 otherwise. We cannot decide in *STT* whether c is 0 or not, contrary to what we expect of the natural numbers. We expect to be able to compute with numbers and decide for any natural number whether or not it is 0.

To capture constructive existence, type theory needs a different logic, so called Intuitionistic or *constructive* logic. Changing the logical basis of *STT* is a radical revision requiring justification. Finding that justification was a serious problem in logic, and the solution adopted in *CTT* came from a partnership of logic, mathematics, and computer science, leading to one of the profound 20th century contributions to logic and computing theory, a

new principle called the **propositions-as-types principle**.[12]

3.2 Semantics of evidence

The propositions-as-types principle asserts that the meaning of a proposition is the type of terms that provide evidence for its truth. For axiomatically true atomic assertions such as $0 = 0$, the evidence is trivial and does not convey any more information than the belief that the assertion is true. So the type corresponding to $0 = 0$ is a non-empty collection of terms. In CTT $\{axiom\}$ is that collection.

For a compound proposition such as $(0 = 0)$ & $(1 = 1)$, the evidence is an ordered pair, $< axiom, axiom >$. In general if A and B are propositions and $\{A\}$ and $\{B\}$ are the *evidence sets*, then the evidence set for $A\&B$ is the cartesian product of $\{A\}$ and $\{B\}$, namely $\{A\} \times \{B\}$. The evidence for $A \vee B$ is the disjoint union $\{A\} + \{B\}$. This semantics for disjunction requires that when we are proving $A \vee B$, we prove one of either A or B, and we know which one we proved.

The evidence for $A \Rightarrow B$ are the constructions (computable functions) that map $\{A\}$ into $\{B\}$. Intuitively this means that $A \Rightarrow B$ is constructively true when we can exhibit a function f that takes any evidence for A and converts it to evidence for B. Thus evidence for $A \Rightarrow A$ is the *identity function*, call it id. Note that if a is any evidence in $\{A\}$, then $id(a)$ is evidence for A because $id(a) = a$.

The evidence for the assertion that for any element a of type T, the propositional function $P(x)$ is true of a, is a function f from $\{T\}$ to $\{P(x)\}$. Notice that the type of the function f is a so-called *dependent function type* because the type of the element $f(t)$ for t in $\{T\}$ is $\{P(t)\}$ which depends on t.

The rule for existence of mathematical objects clearly shows its constructive character. The evidence for the statement that there is an element t of type T such that the propositional function $P(x)$ is true of t is a pair of elements $< t, p_t >$ where p_t is an element of the type $\{P(t)\}$. The type of these ordered pairs is denoted $x : T \times P(x)$ and called a *dependent product*.[13]

[12] In some computer science literature, this principle is called the *Curry-Howard Isomorphism*, e.g. see the book *Lectures on the Curry-Howard Isomorphism* [SU06]. That is a poor name for at least three reasons: it stresses the notion of an isomorphism, when *identification* is a natural interpretation as well (see below), it fails to mention two of the most important contributors to the idea, N.G. de Bruijn and Per Martin-Löf, and it diminishes the philosophical and foundational nature of the principle.

[13] In some literature this is called a *dependent sum* because a disjoint sum (or union) is also a set of ordered pairs, $< t, p_t >$ where t is considered a *tag* telling to which member

We see clearly that to prove that an object of a type T exists with a certain property, we must actually construct a witness t of that type. Symbolically this existence statement is written with an existential quantifier as follows $\exists x : T.P(x)$; it has the same meaning as the dependent product.

The implemented type theories CTT,CIC, and ITT adopt a strong form of propositions-as-types by taking propositions A to be the types $\{A\}$. Based on this identification, the definitions of the logical operators reduce logic to mathematics in a precise way, confirming an intuition of Brouwer and implementing what is also called the Brouwer-Kolmogorov-Heyting (BKH) semantics for constructive logic. On the other hand, we will notice below that to fully express the ideas of type theory, we need to understand how to prove judgements, and that is a matter for deductive logic.

4 Elements of *CTT*

4.1 Terms and programs

To explain constructive truth and to justify rules of inference, we must know more about types. What is a type? To answer this question, we must first be more precise about computation; we will see that the notion of a type is ultimately grounded in computation, specifically in concrete linguistic expressions because computation in the physical world is ultimately symbolic. We are only interested in computation that is physically realizable by explicit, verifiable human actions and by machines that humans understand. Thus to explain types, we first need to explain *terms* and how to compute with them.

In this short article, it is taken for granted that readers have seen some systematic syntax for terms such as for expressions in a programming language. In CTT all terms have a simple uniform syntax which is essentially an *operator identifier* followed by a list of subterms each of which is in the scope of a list of *binding variables*. For example, the notation for functions has the form $\lambda(\bar{x}.body)$ where the body can use the binding variables in the list $\bar{x}$. Here is the identity function in this notation, $\lambda(x.x)$. Notice that this function is *untyped* and thus *polymorphic*, meaning that it belongs to many types, in particular to all types $A \to A$. This feature of starting with an untyped programming language that includes *all algorithms* (even non-terminating ones) is unique to CTT and Martin-Löf's Intuitionistic Type Theory (ITT) [Martin-Löf73, Martin-Löf82, Martin-Löf84].[14]

of the union p_t belongs.

[14] In *CIC* and HOL, all terms are typed – as in programming languages such as ML and

The application of a function f to an argument a is $ap(f; a)$. To make the official syntax readable, the Nuprl implementation of CTT uses *display forms* that allow terms to be displayed in a variety of ways, e.g. we could display the identity function as $\lambda x.x$ if we wished, and display function application as $f(a)$ or even as fa, the ML notation.

CTT takes the decimal numerals as basic terms, they are officially written in the uniform syntax and displayed in the standard way, $0, 1, 2, 3, 4, 5, ...$ There are also terms for two basic arithmetic operations, $+$ and $\times$ on integers, and a computable term for the *induction principle*.

Before we can define types, we need to introduce a computation system on the terms, and that is provided by a reduction relation in the style of *operational semantics*. We write $t \rightarrow t'$ to mean that term t reduces to the term t' in some finite number of steps. For example $ap(\lambda(x.x+1); 0)$ reduces to 1. The terms which are irreducible in the reduction relation serve as the *canonical* elements to be used in defining types.

4.2 Defining a type

To define a type we specify a collection of canonical terms which are the *canonical elements* of the type, and we define an equality relation declaring when two canonical terms denote the same abstract object. The equality relation creates *abstract objects* out of terms. Moreover, any term t which reduces to a canonical term of the type is itself a term of the type. For example, the type of *integers* includes the decimal numbers with the normal equality, and it also includes terms such as $ap(\lambda(x.x + 1); 0)$ which reduces to the canonical interger 1. Likewise, $10 + 7$ is an integer whose canonical form is 17.

A short version of this definition is to say that a type is a collection of equivalence classes of terms.[15] This leads to the so-called *PER* semantics, for *partial equivalence relation semantics*. Stuart F. Allen [All87a] gave a very elegant and precise semantics for CTT and ITT using this approach.

Java, and unlike the terms of Lisp and Scheme. In *CTT*, even the famous **Y** combinator is allowed in assertions and used to define recursive functions.

[15] These equivalence classes are understood in a computational way, not as sets of sets, but as types distinguished by their equality relations. This concept is formalized in *CTT* as the *quotient type*, a type that overcomes the computational limitations of the standard definition of equivalence classes to which Bishop objected.

5 Judgements and Propositions

5.1 Asserting propositions

How do we make logical claims in computational type theory? A simple logical claim is to assert that $(1 + 1) = 2$. In CTT this is trivial because $1 + 1$ reduces to 2 and thus is 2. As noted above, the *equality judgement* is part of the type definition, and for any type A, it is written $a = b\ in\ A$. The other kind of logical claim made in CTT is the *judgment* that a belongs to A. This can be reduced to the equality claim by writing $a = a\ in\ A$. Now according to the propositions-as-types principle, propositions are types, thus to assert a proposition A is to claim that there is an element in its evidence set, and if the proposition is A, then this claim is $a = a\ in\ A$ for some term a.

The basic form of logical judgment in CTT is to assert that a type is inhabited under the assumption that other types are in habited. The form of this judgement in the Nuprl implementation of CTT is in the style of *sequents*. Here is an example, $A : Type, B : Type, y : B \vdash (A \Rightarrow B)$. To prove this sequent, we need to find an element of $A \Rightarrow B$ using the assumption that y is an element of B and the assumptions that A and B are types. The proof of this claim is to exhibit the witness in type $A \Rightarrow B$. One good witness is the function $\lambda(x.y)$, and we can express this by putting the witness after the goal of the sequent, calling it the witness or the *extract*. So here is a true sequent which we could build using Nuprl rules: $A : Type, B : Type, y : B \vdash (A \Rightarrow B)\ ext\ \lambda(x.y)$. Note, this is valid evidence even if A is empty. The canonical empty type is *void*, and any empty type is evidence for an unprovable, hence *false*, proposition.

5.2 Proofs

Proofs in CTT are finite trees whose nodes are sequents and whose leaves are axiomatically true sequents. In Nuprl the root of the tree is the goal to be proved, and it is presented at the top, so the tree is "upside down". The proof is built by *refining the goal* into subgoals until all subgoals are axiomatic sequents, for example $\vdash 0 < 1\ ext\ axiom$ is axiomatic.

Once the proof tree is built, it is possible to fill in the extracts for every sequent of the proof. At the top level this extract will be the *computational content* of the proof. If the goal has the form $For\ all\ x : A\ we\ can\ find\ y : B\ such\ that\ R(x, y)$, then the extract will be a computable function from type A to type $y : B \times R(x, y)$. Thus the proof of the goal can be seen as creating a program to compute a function meeting a certain specification.

We say that the *proof acts as a program.* This *proofs-as-programs* principle is a consequence of the propositions as types principle. Systems like Nuprl and MetaPRL *extract* programs from constructive proofs in *CTT*. The Coq system does this for proofs in the type theory *CIC*, and Agda does it for proofs in *ITT*.

6 Applications of *CTT* Using Nuprl

6.1 Developing mathematical theories and computing systems

Proof systems for *CTT*, *CIC*, and *ITT* have been used as programming environments for specifying and developing verified algorithms for a variety of problems in computer science and computational mathematics thereby realizing a dream of Leibniz. The most famous example from mathematics is Georges Gonthier's proof of the Four Color Theorem in the Coq implementation of *CIC*.[16] Peter Aczel shows how to use *ITT* to define constructive set theory [Acz78], and Jason Hickey implemented this definition in *CTT* [Hickey01]. One of the most practical examples from computer science is the automatic optimization of protocol stacks in the Ensemble system [SOSP99] which was taken up by industry.

6.2 Solving open problems

Proof systems such as Nuprl [Nuprl Book, Nuprl-Home] have been used to solve open problems in mathematics by finding constructive proofs in cases where they were not known – one by Douglas Howe for the Girard Paradox, [Howe87a], and one for a constructive proof of Higman's Lemma by Chetan Murthy [Mur91a].

7 Further Reading

There are several books and articles about computational type theory that contain extensive bibliographies and present the subject from different points of view and at different levels.

The book *Implementing Mathematics with the Nuprl Proof Development System* [Nuprl Book] is on-line at the Nuprl home page [Nuprl-Home].

[16] www.research.microsoft.com/ gonthier/4colproof.pdf.

The text book of Simon Thompson from 1991 relates computational type theory, specifically *CTT*, to functional programming. *Type Theory and Functional Programming*, Addison-Wesley, New York, 1991.

The article Do-it-yourself Type Theory by Backhouse, Chisholm, and Saaman is a very readable introduction to the ideas behind *CTT* and *ITT*. It appears in *Formal Aspects of Computing*, Vol 1, 1989, 19-84.

The 1994 research monograph of Aarne Ranta includes a very clear explanation of logic in computational type theory. *Type-Theoretical Grammar*, Clarendon Press, Oxford, 1994.

The monograph of Bengt Nordström, Kent Petersson, and Jan Smith presents functional programming in the context of Intuitionistic Type Theory (*ITT*). *Programming in Martin-Löf's Type Theory*, Clarendon Press, Oxford, 1990.

Naive Computational Type Theory is a forty six page expository article on computational type theory available in the publications section of the Nuprl home page.[17]. A published version appears in *Proof and System Reliability*, edited by Helmut Schwichtenberg and Ralf Steinbruggen, Nato Science Series, #62, Kluwer Academic Publishers, Boston, 2002, 213-259.

A detailed technical account of ITT from the viewpoint of constructive logic appears in the research monograph of A.S. Troelstra and D. van Dalen, *Constructivism in Mathematics, An Introduction, Volume II*, North-Holland, Amsterdam, 1988.

An technical survey of computational type theory in the setting of proof theory is the long expository article on Types in Logic, Mathematics and Programming by Robert Constable that is chapter X in *Handbook of Proof Theory* edited by Sam Buss and published by Elsevier Science, New York, 1998, 683-786.

Numerous formal articles on topics in mathematics and computer science appear in the *Formal Digital Library* (*FDL*) appear as "books" written in Nuprl and some in HOL. These can be accessed at the Nuprl Home page [Nuprl-Home] under Math Library (selected parts).[18]

References

[Acz78] Peter Aczel. The type theoretic interpretation of constructive set theory, *Logic Colloquium '77*, Editors A. MacIntyre, L. Pacholski, and J. Paris, North Holland, Amsterdam, 1978.

[17] www.cs.cornell.edu/info/projects/nuprl/html/publication.html

[18] http://www.cs.cornell.edu/Info/Projects/NuPRL/Nuprl4.2/Libraries/Welcome.html

[All87a] Stuart F. Allen. A Non-Type-Theoretic Definition of Martin-Löf's Types, *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, Washington, 1987, 215-224.

[ABC+06] 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*, Elsevier Science, New York, 2006, 428-469.

[Bic08] Mark Bickford. Unguessable Atoms: A Logical Foundation for Security, *Verified Software: Theories, Tools, and Experiments*, *Lecture Notes in Computer Science*, Springer, New York, 2008.

[BC06] Mark Bickford and Robert L. Constable. A causal logic of events in formalized computational type theory, In *Logical Aspects of Secure Computer Systems, Proceedings of International Summer School Marktoberdorf 2005*, to appear. Earlier version available as Cornell University Technical Report TR2005-2010, 2005.

[deBruijn70] N. G. de Bruijn. The Mathematical Language Automath: its Usage and Some of its Extensions, *Symposium on Automatic Demonstration*, Editors J. P. Seldin and J. R. Hindley, *Lecture Notes in Mathematics, Vol 125*, Springer-Verlag, New York, 1970, 29-61.

[Bishop67] , Errett Bishop. *Foundations of Constructive Analysis*, McGraw Hill, New York, 1967.

[Brouwer75] L. E. J. Brouwer. *L.E.J. Brouwer, Collected Works, Volume 1*, Editor A. Heyting, North-Holland, Amsterdam, 1975.

[BYC04] Yves Bertot and Castéran Pierre. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science, Springer-Verlag, New York, 2004.

[Church40] Alonzo Church. A Formulation of the Simple Theory of Types, *Journal of Symbolic Logic*, 1940, 5, 55–68.

[Chu51] Alonzo Church. *The Calculi of Lambda-Conversion*, *Annals of Mathematical Studies*, Vol 6, Princeton University Press, Princeton, 1951.

[Nuprl Book] 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, Englewood Cliffs, 1986 (available at the Nuprl Home page under The Book on the first page).

[CC01] Robert L. Constable and Karl Crary. Computational Complexity and Induction for Partial Computable Functions in Type Theory, *Reflections on the Foundations of Mathematics: Essays in Honor of Solomon Feferman*, Editors Wilfried Sieg, Richard Sommer, and Carolyn Talcott, *Lecture Notes in Logic*, Association for Symbolic Logic, Natick, MA, 2001, 166-183.

[CS93] Robert L. Constable and Scott F. Smith. Computational Foundations of Basic Recursive Function Theory, *Theoretical Computer Science*, Vol 121, 1993, 89–112.

[Dummett77] Michael Dummett. *Elements of Intuitionism*, Clarendon Press, Oxford, 1977.

[GM93] Michael Gordon and Tom Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*, Cambridge University Press, Cambridge, 1993.

[GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, *Lecture Notes in Computer Science 78*, Springer-Verlag, New York, 1979.

[Heyting71] A. Heyting. *Intuitionism*, North Holland, Amsterdam, 1971.

[Hickey01] Jason Y. Hickey, *The MetaPRL Logical Programming Environment*, PhD Thesis, Cornell University, Ithaca, New York, 2001.

[MetaPRL] Jason Hickey, Aleksey Nogin, et al. MetaPRL – Modular Logical Environment, *Proceedings of the 16^{th} International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, *Lecture Notes in Computer Science, Vol 2758*, Springer-Verlag, New York, 2003, 287-303.

[Howe87a] Douglas J. Howe. The Computational Behaviour of Girard's Paradox, *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society Press, Washington, 1987, 205-214.

[Kleene45] Stephen Cole Kleene. On the interpretation of Intuitionistic number theory, *J. Symbolic Logic*, 10, 1945, 109-124.

[KV65] Stephen Cole Kleene and R. E. Vesley. *The Foundations of Intuitionistic Mathematics*, North-Holland, Amsterdam, 1965.

[SOSP99] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason J. Hickey, Mark Hayden, Kenneth Birman, and Robert Constable, Building Reliable, High-Performance Communication Systems from Components, *ACM Symposium on Operating Systems Principles (SOSP'99)*, *Operating Systems Review*, 33, 5 Editors David Kotz and John Wilkes, ACM Press, New York, 1999, 80-92.

[Martin-Löf73] Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part, *Logic Colloquium '73*, North-Holland, Amsterdam, 1973, 73-118.

[Martin-Löf82] Per Martin-Löf. Constructive Mathematics and Computer Programming, *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, North Holland, Amsterdam, 1982, 153-175.

[Martin-Löf84] Per Martin-Löf. *Intuitionistic Type Theory*, *Studies in Proof Theory, Lecture Notes* 1, Bibliopolis, Napoli, 1984.

[Morse65] A. Morse. *A Theory of Sets*, Academic Press, New York, 1965.

[Mur91a] Chetan Murthy. Classical Proofs as Programs : How, What, and Why, *Lecture Notes in Computer Science Vol 613*, Springer-Verlag, New York, 1991, 71-88.

[Nuprl-Home] Cornell PRL Group. see www.nuprl.org or http://www.cs.cornell.edu/Info/Projects/NuPRL/

[Russell08b] Bertrand Russell. *The Principles of Mathematics*, Cambridge University Press, Cambridge, 1908.

[NGDV94] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer, *Selected Papers on Automath*, *Studies in Logic and The Foundations of Mathematics Volume 133*, Elsevier, Amsterdam, 1994.

[Russell08a] Bertrand Russell. Mathematical logic as based on a theory of types, *American. J. Math*, Vol 30, 1908, 222-62.

[Scott70] Dana Scott. Constructive Validity, *Symposium on Automatic Demonstration*, Editors M. Laudelt and D. Lacombe, *Lecture Notes in Mathematics Vol 125*, Springer-Verlag, New York, 1970, 237-275.

[SU06] M.H. Sorensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, Elsevier, New York, 2006.

[WR25] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*, 2nd Edition, Volumes 1,2,3, Cambridge University Press, 1925.