On Building Constructive Formal Theories of Computation
Noting the Roles of Turing, Church, and Brouwer

Robert L. Constable
Cornell University

***Abstract*—In this article I will examine a few key concepts and design decisions that account for the high value of implemented constructive type theories in computer science. I'll stress the historical fact that these theories, and the proof assistants that animate them, were born from a strong partnership linking computer science, logic, and mathematics.**

I will recall how modern type theory researchers built on deep insights from the earliest pioneers: Turing – the first computer scientist, Church – the patriarch of logic in computer science, and Brouwer – a singular pioneer of intuitionism and constructive mathematics. They created solid intellectual ground on which to build a formal implemented constructive theory of computation whose influence will be felt well beyond computing and information science alone. All generations of constructive type theory researchers since this beginning have had leaders from all three disciplines.

Much of the seminal modern work creating these type theories and their proof assistants was presented in LICS proceedings, and LICS could be a natural home for future work in this flourishing area which is the epitome of logic in computer science.

***Index Terms*—attack-tolerance, automated reasoning, Coq, completeness, FLP theorem, intuitionistic logic, logic of events, Nuprl, proof assistants, propositions-as-types, protocol synthesis**

I. Introduction

A. Background

Computer scientists report a growing influence of *implemented constructive type theories*. This is due in part to their role in programming language research, operating system verification, fault-tolerant and attack-tolerant protocol synthesis, access control systems, security research, and other core computing topics. These implemented theories embody deep and compelling discoveries from computer science, logic, and mathematics that become tightly entangled in the implementations. Such theories have the potential to provide an integrated *foundational theory of computation* suitable to all three disciplines and beyond. Current circumstances portend an increased effort to enrich these theories and improve their implementations in proof assistants that become widely shared distributed resources "residing in the clouds".

One of many examples of a growing influence of constructive type theories is that at the premier conferences on programming languages such as POPL we see results established using proof assistants like Coq or HOL or Twelf; and some researchers claim that it is becoming a requirement for certain kinds of results in programming language (PL) research that they be formally proved and checked by a proof assistant. In other meetings we see protocol correctness [14] established by Nuprl and properties of the seL4 microkernel verified using Isabelle-HOL [36] and certified system code using Coq [80]. The Coq and Nuprl proof assistants are based on related versions of constructive type theory, the *Calculus of Inductive Constructions* for Coq [10], [75] and *Computational Type Theory* (CTT) [26], [2] for Nuprl.[1] There is also a constructive version of Isabelle-HOL type theory [9], [8] whose semantics is given using Intuitionistic ZF set theory [22]. One of the most notable verifications using a proof assistant is Gonthier's proof of the Four Color Theorem in Coq [39]. Coq and Nuprl were both used in settling open mathematical problems [39], [71], and HOL is being used to formalize and check Halles proof of the Kepler conjecture [83]. The Minlog system [7] has been used in very interesting proof transformations. These results depend on advanced proof technology, and at the same time they illustrate some of the deep intellectual contributions of computer science – predicted by AI researchers in the 50's and examined by philosophically trained logicians who tie this work to enduring questions of epistemology [34], [35], [41].

I believe that research investments by industry, governments, and universities will expand the role of constructive type theories, deepen the science behind them, and encourage their implementation in the cloud. New mathematical and logical results, e.g. Homotopy Type Theory [87], [5], [57], will not only make the theories more expressive, they will also help us implement them better. Research in programming languages and programming environments could make *correct-by-construction programming with dependent types* standard [16]. Proof assistants will share formal theories and integrate other formal methods tools into an advanced proof technology built around them. The inherently compositional nature of *proof assistant based formal methods* will multiply the value of proof assistants as they share theories. Critical applications will justify adding computing power to the proof assistants; that power will extend their reach deeper into science and

[1]For example, both theories use the results of Mendler [69], [70] on recursive types, both are based on propositions-as-types, both use the LCF tactic mechanism [40], and the systems share some implementation elements.

mathematics. We will also apply these theories and their proof assistants to themselves to significantly improve their implementations, illustrating the autocatalytic nature of research in this field.

B. *Organization*

In the next section I present some of the most important basic principles behind the design of Computational Type Theory (CTT) [2], the name we gave to the version of constructive type theory implemented by Nuprl as of 2006. This theory includes the types needed to build a theory of classes and objects as reported in work with Jason Hickey [46] and Alexei Kopylov [56] as well as work on the Formal Digital Library (FDL) that is the data base of definitions, tactics, and theorems used by Nuprl. Since the 2006 publication, we have taken steps to add distributed processes to the computation system and the type of *processes* and *events* to the logic [13]. We view this as extending the notion of *proofs as programs* [23], [6], [42] to that of *proofs as processes*.

II. Basic Principles

A. *The computation system and its data*

Constructive type theories include a programming language. For ITT82 [65], [74] and CTT [26], [2] this is an *applied untyped lambda calculus* based on Church's *pure untyped lambda calculus* [19], [20]. Instead of encoding into pure lambda terms the operators such as pairs and tags as well as *data*, e.g. Booleans, numbers, lists etc, these are added as new primitives. The computation rules are typically presented in the style of Plotkin's structured operational semantics [77]. An applied untyped lambda calculus with atoms and lists became the basis for Lisp [67] (in which Nuprl is implemented).

As Church, Kleene and Rosser investigated the untyped lambda calculus in the 30s, Church came to believe it captured the intuitive notion of *effective computability*, but Gödel and others were not convinced until they saw that the system was equivalent to Turing machines. We now say that the lambda calculus is *Turing complete*, recognizing the intuitive clarity of Turing's formulation [85]. The CIC type theory starts with an applied typed lambda calculus and is thus a theory based on *total computable functions* whereas CTT is based on *partial computable functions*. This difference was studied in detail in work with Smith [27] and Crary [28]. An efficient implementation of this underlying programming language guarantees that the type theory can express algorithms well enough for serious practical work. In the case of Nuprl, this efficiency is obtained by applying a series of transformations to the primitive evaluator, adding closures, continuations, and defunctionalization to end up with an efficient state machine as described by Danvy and Nielsen [30]. This and other techniques from functional programming provide a principled path to efficient evaluation.

a) Church's Thesis: None of CIC, CTT nor ITT adopt Church's thesis, and CTT is *open-ended* by design [24], [50]. If it is necessary to reason about Turing computable functions, they are defined as an explicit subtype.[2]

One of the major advantages of using an untyped computation system is that it can express the most efficient and compact algorithms. For example, the **Y** combinators are available for expressing arbitrary recursive algorithms. In Nuprl we use these combinators as realizers for *efficient induction principles*, a technique that Doug Howe used to great effect in his work on the Girard paradox [48]. For instance, the realizer for the following efficient induction principle is given by a **Y** combinator that is proved to be in the type of the induction principle: $(P(0)\&\forall n:\mathbb{N}.P(n\div 4)\Rightarrow P(n))\Rightarrow\forall n:\mathbb{N}.P(n)$. This is realized by a recursive function, say $f(x) = \ if\ x = 0\ then\ b\ else\ h(x, f(x\div 4))$.

B. *Computational equality*

Reasoning about untyped terms is critical in automated reasoning. In Nuprl this is based on Howe's results on equality in lazy computation systems [49]. We write $a \sim b$ when a is computationally equal to b and use the generalized rewrite package [33] that Paul Jackson wrote [52], [53] based on Paulson's rewrite package from Isabelle [76]. One of the most common rewrites is to replace a term by one computationally equal to it. This is an effective way to establish properties of efficient realizers, by showing them computationally equal to the primitive realizers.

C. *Typing judgments*

If we let fix be a simple fixed point combinator with the reduction rule that $fix(\lambda(f.b(f)))$ reduces in one step to $b(fix(\lambda(f.b(f))))$, then to use

$$fix(\lambda(f.\lambda(x.\ if\ x=0\ then\ b\ else\ h(x,f(x\div 4)))))$$

as a realizer for the above efficient induction formula, we only need to prove that it belongs to the type of the formula according to the propositions as types principle and the PER semantics based on it [3], [44]. This involves finding the right terms b and h. In CTT and ITT this is done by proving the *typing judgment* by ordinary induction. It is not established by a type checking algorithm because in general type judgments are undecidable. Here we see a contrast between CTT and CIC that would need to be resolved in any unified theory. One plausible unification is to employ a type checking algorithm and also *allow other type judgments to be established by proof.* This is essentially what happens in practice in Nuprl because the normal "auto-tactic" proves a very large percentage of the typing judgments.

D. *Logic*

It is well known that the implemented constructive type theories are based on the *propositions as types principle*. This fundamental logical principle is used to support the extraction

[2]There are researchers who believe that Martin-Löf intended to adopt Church's thesis in ITT based on his writings of the time. I don't know if that is the case, but for CTT we never intended to add a version of Church's thesis.

of programs from proofs. I regard this principle as one of the most basic ideas in logic – yet it does not have a standard name. Computer scientists, especially in the LICS community, like to call this idea the *Curry-Howard isomorphism*. Indeed that is the title of one of the comprehensive books on the subject, *Lectures on the Curry-Howard Isomorphism*, [81]. However, that book cites over twenty contributors to the concept, from logic, computer science, and mathematics. The origin of the name might stem from Martin-Löf's writings [65], [66] in part because he was very influenced by Howard [47] when he visited him in the late 60's.

The idea occurs clearly as an isomorphism in the book *Combinatory Logic* [29] by Curry and Feys, pages 312-315. However, the central idea goes back to Brouwer 1907 [86], [18], and among logicians it is called the Brouwer/Heyting/Kolmogorov or *BHK semantics* for intuitionistic logic, see [84], [82]. N.G. deBruijn called it propositions as types (PAT) or *proofs as terms* (PAT), and he used the idea for all Automath theories [32], [73] which are classical. It seems that he learned this from Heyting when they were colleagues in Amsterdam. The connection from early Brouwer to recursive realizability was made by Kleene already in 1945 [54] and extended in the book *Foundations of Intuitionistic Mathematics* with Vesley in 1965 [55]. Let's call this semantics *Brouwer realizability* when indexings of partial recursive functions are not used. I suggested the name *evidence semantics* when I showed that the principle could be applied to classical logic using oracle computations (called *magic*) to justify the law of excluded middle [24], [50], [51], and the resulting semantics is classically equivalent to Tarski's semantics for first-order logic.

b) Completeness problem: The BHK semantics was problematic because logicians could not prove completeness for Intuitionistic First-Order Logic (iFOL) relative to the BHK semantics even though Beth [11] made a valiant effort in 1947 which gave us tableaux systems.

The most faithful constructive completeness theorem for *intuitionistic validity* is by Friedman in 1975 (presented in [84]), and there is a classical proof for the Brouwer-Heyting-Kolmogorov semantics for intuitionistic first-order logic by Artemov using *provability logic* [4]. Results suggest how subtle completeness theorems are since constructive completeness with respect to full intuitionistic validity contradicts Church's Thesis [58], [84] and implies *Markov's Principle* as well [68].[3]

My colleague Mark Bickford and I have made progress on this problem recently by observing that intuitionistic completeness is not the right notion. It turns out that the proof rules for iFOL yield a stronger semantic notion, *uniform validity*, because the realizers are all polymorphic terms. In a submitted article "Intuitionistic completeness of first-order logic" [21], we established completeness for *intuitionistic first-order logic, iFOL*, by showing that a formula is provable if and only if its embedding into minimal logic, *mFOL,* is *uniformly valid* under the *Brouwer Heyting Kolmogorov (BHK)* semantics, the intended semantics of both iFOL and mFOL. Our proof is intuitionistic, and it provides an effective procedure Prf that converts uniform minimal evidence into a formal first-order proof.

Theorem: Completeness of Minimal First-Order Logic: Given a valid evidence structure, $\bar{h} : \bar{H}(D, \bar{R}) \models G(D, \bar{R}), evd(\bar{h})$, with uniform evidence $evd(\bar{h})$ over the domain D and atomic relations $\bar{R}$ for a logical formula $G(D, \bar{R})$ in first-order minimal logic, we can build a minimal logic proof, $pf(\bar{h})$, of the goal formula $G(D, \bar{R})$, from hypotheses $\bar{H}$, that is, $\bar{h} : \bar{H} \vdash^1_{min} G(D, \bar{R})\ by\ pf(\bar{h})$.

Moreover, from the proof of this completeness theorem, we can extract a proof building procedure Prf that yields the proof $pf(\bar{h})$. We have implemented Prf. Uniform validity is defined using the intersection operator as a universal quantifier over the domain of discourse and atomic predicates. Formulas of *iFOL* that are uniformly valid are also intuitionistically valid, but not conversely. Our strongest result requires the Fan Theorem; it can also be proved classically by showing that Prf terminates using König's Theorem.

We intuitionistically prove completeness for *iFOL* as follows. Friedman showed that *iFOL* can be embedded in minimal logic (*mFOL*) by his A-transformation, mapping formula F to F^A. If F is uniformly valid, then so is F^A, and by our completeness theorem, we can find a proof of F^A in minimal logic. Then we intuitionistically prove F from F^{False}, i.e. by taking $False$ for A and for $\perp$ of mFOL. This result resolves an open question posed by Beth in 1947.

Corollary: **Completeness of Intuitionistic First-Order Logic:** F is a uniformly valid formula of iFOL if and only if it is provable in iFOL.

III. Implementations

We look briefly at some of the key ideas and issues in the implementations of constructive type theories. This is a realm in which the computer scientists were indispensable both to the design of the theories and to their deployment and application. On the other hand, the task could not be separated from logical issues nor from the norms of mathematical practice.

A. LCF-style tactics

It is noteworthy that the Coq, HOL, Isabelle, and Nuprl proof assistants are all implemented using the tactic mechanism from the LCF system (Logic for Computable Functions) reported in the book *Edinburgh LCF* [40], subtitled "A Mechanised Logic of Computation". This means that proof automation is based on tactics and tacticals and goal directed proof. Tactics generate *primitive proofs* that can be independently checked by relatively simple programs. These *tactic-based provers* also integrate various decision procedures and fully automatic provers, JProver [78], which generate primitive proofs when they succeed.

The power of the tactic mechanism comes from its abstract and heuristic nature. Tactics are not guaranteed to find a proof, they search and they guess and they try. On the other hand,

[3]Church's Thesis was not an issue for us because we do not assume it in CTT, our metalogic.

they need to be stable enough to replay when they succeed. Tactics can be assembled into "super-tactics" that start to look like automatic theorem provers for certain classes of results [12].[4] In Nuprl there is a very powerful tactic called MA-Auto that can often finish a proof. In Coq a corresponding super tactic is called "Crush." In addition, special super-tactics are assembled for domain specific verification. These tactics are designed to completely solve the task they are designed for, and because they work most of the time, they are allowed to run for many hours.

B. Programming Environments

Environments for using constructive proof assistants also provide evaluators and compilers. Formal Digital Library (FDL) capabilities are especially important once the formal library is very large with many users. It is important to know whether a concept is already defined or a theorem already proved by another user (or oneself in the past). Jason Hickey built a *logical programming environment*, LPE, and a proof assistant for CTT called MetaPRL [46]. MetaPRL is a logical framework [45] in which theories can be formally related, e.g. using Aczel's implementation of CZF [1] into CTT. Logical frameworks like MetaPRL allow users to share related theories and share results [37], [72]. They are important to the study of Mathematical Knowledge Management which is a topic that will become increasingly important as the proof assistants tackle larger tasks and need to cooperate. On this topic it will be essential for computer scientists, information scientists, logicians and mathematicians to cooperate.

The Nuprl FDL includes an on-going development of constructive analysis following closely the book *Constructive Analysis* [15] by Bishop and Bridges which extends Bishop's 1967 book that brought the new generation of constructive mathematicians including Bridges and Richman [17] into the partnership. They reexamined ties to Brouwer. Bishop inspired logicians to find a formalism for his mathematics, including Martin-Löf [65] and Scott [79] and computer scientists like me striving to design implementable formalisms for mathematics and the theory of computation [23].

IV. Formal Distributed Computing Model

Ever since our commercialized work on the Ensemble distributed system [63], [59], we have devoted more effort to verifying and synthesizing distributed protocols. This has led us to model distributed computing concepts and primitives in the constructive type theory of Nuprl. Our first efforts used the IOA model of Lynch [64] and the logic of events [14] derived from a variant of the IOA model. Since then we have made the process model progressively more abstract and now are using our General Process Model [13]. Here is a brief overview of this model including key concepts for reasoning about *event structures* created from computations in this model. A *system* consists of a set of *components*. Each component has a *location*, an *internal* part, and an *external* part. There may be more than one component with the same location. The internal part of a component is a *process*—its program and internal (possibly hidden) state. The external part of a component is its interface with the rest of the system. Here the interface is a list of *messages*, containing either *data* or processes and labeled with the location of the recipient. The "higher order" ability to send a message containing a process allows such systems to grow by "forking" or "bootstrapping" new components. A system executes as follows. At each step, the *environment* may choose and remove a message from the external component. If the chosen message is addressed to a location that is not yet in the system, then a new component is created at that location, using a given *boot-process*, and an empty external part.

[4]The term *super-tactic* was used by Miriam Leeser and her group [61] and applied by Bickford to PCL as well [31], [12].

Each component at the recipient location receives the message as input and computes a pair that contains a process and an external part. The process becomes the next internal part of the component, and the external part is appended to the current external part of the component. A potentially infinite sequence of steps, starting from a given system and using a given boot-process, is a *run* of that system. From a run of a system we derive an abstraction of its behavior by focusing on the *events* in the run. The events are the pairs, $\langle x, n\rangle$, of a location and a step at which location x gets an input message at step n, i.e. *information is transferred*. Every event has a location, and there is a natural *causal-ordering* on the set of events, the ordering first considered by Lamport [60]. This allows us to define an *event-ordering*, a structure, $\langle E,\ loc,\ <,\ info\rangle$, in which the causal ordering $<$ is transitive relation on E that is well-founded, and locally-finite (each event has only finitely many predecessors).[5] Also, the events at a given location are totally ordered by $<$. The information, $info(e)$, associated with event e is the message input to $loc(e)$ when the event occurred. We have found that requirements for distributed systems can be expressed as (higher-order) logical propositions about event-orderings. To illustrate this, I present a very simple example of *consensus* in a group of processes.

c) A simple consensus protocol: TwoThirds: Each participating component will be a member of some group and each group has a name, G. The groups have $n = 3f + 1$ members, and they are designed to tolerate f failures. When any component in a group G receives a message $\langle [start], G\rangle$ it starts the consensus protocol whose goal is to decide on values received by the members from *clients*.

We assume that once the protocol starts, each process has received a value v_i or has a default non-value. The simple TwoThirds consensus protocol is this: A process P_i that has a value v_i of type T starts an *election* to choose a value of type T (with a decidable equality) from among those received by members of the group from clients.

Begin
Until $decide_i$ **do**:
1. **Increment** el_i; 2. **Broadcast** vote $\langle el_i, v_i\rangle$ to G;
3. **Collect** in list Msg_i $2f + 1$ votes of election el_i;

[5]These event structures and orderings are similar in spirit to Winskel's [88].

```
4. **Choose** $v_i := f(Msg_i)$;
5. **If** $Msg_i$ is unanimous **then** $decide_i := true$
**End**
```

The elections are identified by natural numbers, el_i initially 0, and incremented by 1, and a Boolean variable $decide_i$ is initially $false$. The function from lists of values, Msg_i to a value is a parameter of the protocol. If the type T of values is Boolean, we can take f to be the majority function.

We describe protocols like this by classifying the events occurring during execution. In this algorithm there are *Input*, *Vote*, *Collect*, and *Decide* events. The components can recognize events in each of these *event classes* (in this example they could all have distinctive headers), and they can associate information with each event (e.g. $\langle e_i, v_i \rangle$ with Vote, Msg_i with Collect, and $f(Msg_i)$ with Decide). Events in some classes *cause* events with related information content in other classes, e.g. Collect causes a Vote event with value $f(Msg_i)$. In general, an *event class* X is function on events in an event ordering that *effectively partitions* events into two sets, $E(X)$ and $E - E(X)$, and assigns a value $X(e)$ to events $e \in E(X)$.

Example 1. *Consensus specification*

Let P and D be the classes of events with headers $propose$ and $decide$, respectively. Then the *safety specification* of a consensus protocol is the conjunction of two propositions on (extended) event-orderings, called *agreement* (all decision events have the same value) and *responsiveness* (the value decided on must be one of the values proposed):

$$\forall e_1, e_2 : E(D).\ D(e_1) = D(e_2)$$
$$\forall e : E(D).\ \exists e' : E(P).\ e' < e\ \wedge\ D(e) = P(e').$$

We can prove safety and the following *liveness property* about TwoThirds. We say that activity in the protocol *contracts to a subset* S of exactly $2f + 1$ processes if these processes all vote in election n say at $vt(n)_1, ..., vt(n)_k$ for $k = 2f + 1$ and collect these votes at $c(n)_1, ..., c(n)_k$, and all vote again in election $n + 1$ at $vt(n + 1)_1, ..., vt(n + 1)_k$, and collect at $c(n + 1)_1, ..., c(n + 1)_k$. In this case, these processes in S all decide in round $n + 1$ for the value given by f applied to the collected votes. This is a *liveness* property. If exactly f processes fail, then the activity of the group G contracts to some S and decides. This fact shows that the TwoThirds protocol is *non-blocking*, i.e. from any state of the protocol, there is a path to a decision.

A. Realizers and Strong Realizers

If ψ is a proposition about event orderings, e.g. liveness, we say that a system *realizes* ψ, if the event-ordering of any run of the system satisfies ψ. We extend the "proofs-as-programs" paradigm to "proofs-as-processes" for distributed computing by making constructive proofs that requirements are *realizable*. For compositional reasoning, it is desirable to create a *strong realizer* of requirement ψ—a system that realizes ψ *in any context*. Formally, system S is a strong realizer of ψ if the event-ordering of any run of a system S' such that $S \subseteq S'$, satisfies ψ. If S_1 is a strong realizer of ψ_1 and S_2 is a strong realizer of ψ_2, then $S_1 \cup S_2$ is a strong realizer of $\psi_1 \wedge\ \psi_2$. One of our main tools is that propagation rules like those used in the consensus example have strong realizers. A realizer for a propagation rule $A \overset{f}{\Rightarrow} B@g$ is a set of components that can each, as a (computable) function of the history of inputs at its location, recognize, and compute the value v of events in class A that occur there and send messages that will eventually result in an events in class B with value $f(v)$ at each location in $g(v)$. We call the classes A that can be so recognized *programmable*.

V. Attack Tolerance

We assume that deployed systems will be attacked. We can protect protocols by formally generating a large number of *logically equivalent variants*, stored in an *attack response library*. Each variant uses distinctly different code which a system under attack can install *on-the-fly* to replace compromised components. Each variant is known to be equivalent and correct. We express threatening features of the environment formally and discriminate among the different types. We can do this in our new GPM model because *the environment is an explicit component about which we can reason.*

A. Synthetic Code Diversity

We introduce diversity at all levels of the formal code development (synthesis) process starting at a very high level of abstraction. For example, in the TwoThirds protocol, we can use different functions f, alter the means of collecting Msg_i, synthesize variants of the protocol, alter the data types, etc. We are able to create multiple provably correct versions of protocols at each level of development, e.g. compiling TwoThirds into Java, Erlang, and $F^{\#}$. The higher the starting point for introducing diversity, the more options we can create. We can also inject code diversity into the fully automatic verification of authentication protocols in *Protocol Composition Logic* (PCL) [31] implemented in Nuprl.

d) The Environment as Adversary: The standard version of the Fisher/Lynch/Paterson theorem [38] is that no deterministic algorithm can solve the consensus problem for a group of process in which at least one process might fail. This is a negative statement, producing only a contradiction, yet implicit in all proofs is an *imagined construction* of a nonterminating execution in which no process decides, they "waffle" endlessly. That imagined execution is an interesting object, displaying what can go wrong in trying to reach consensus. The hypothetical execution is used to guide thinking about consensus protocol design. In light of that use, a natural question about the classical proofs of FLP is whether the hypothetical infinite waffling execution could actually be constructed from any purported consensus protocol **P**; that is, given **P**, can we exhibit an algorithm α such that for any natural number n, $\alpha(n)$ is the n-th step of the indecisive computation? It appears that no such explicit construction could be carried out following the method of the classical proof because there isn't enough information given with the protocol, and the key concept in the standard proofs, the notion of *valence* (*univalence* and *bivalence*), is not defined effectively, i.e. it requires knowing the results of all possible executions. Only a *proof* can show that it will run forever.

The key to being able to build the nonterminating execution is to provide more information, which was done in [25] by introducing the notion of *effective nonblocking*. Effective nonblocking is a natural concept when protocols are verified using constructive logic.

P is called *effectively nonblocking* if from any reachable global state s of an execution of **P** and any subset Q of $n - f$ non-failed processes, *we can find* an execution α from s using Q and a process P_α in Q which decides a value v. Constructively this means that we have a *computable function*, $wt(s, Q)$, which produces an execution α and a state s_α in which a process, say P_α decides a value v.

Theorem (CFLP): Given any deterministic effectively nonblocking consensus procedure **P** with more than two processes and tolerating a single failure, we can effectively construct a nonterminating execution of it. Let the function produced by this proof be **flpc**, then for a consensus procedure, say the *TwoThirds* protocol given above and its nonblocking proof nb, we have that the environment can use **flpc**(nb) to create a message-order attack that will prevent TwoThirds from deciding.

What is noteworthy and perhaps alarming about this constructive result is that it says that if we take the trouble to prove that a consensus algorithm is non-blocking and correct, then we provide an adversary who knows the proof and controls the network (as in a data center), an undefeatable denial of service attack on a system that uses the protocol. This is not what we expect from protocol verification!

VI. Conclusion

I believe that the core partnership driving the development of implemented constructive type theories remains vital and productive and will be reinforced and strengthened by increasingly important applications, new discoveries about type theory in general, and new *prover-assisted programming languages* built to exploit the richness of the theories and the capabilities of their proof assistants.

References

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

[2] 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.

[3] Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In Gries [43], pages 215–224.

[4] Sergei Artemov. Uniform provability realization of intuitionistic logic, modality and lambda-terms. *Electronic Notes on Theoretical Computer Science*, 23(1), 1999.

[5] S. Awodey, N. Gambino, and K. Sojakova. Inductive types in homotopy type theory. Technical Report arXiv:1201.3898v1, 2012.

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

[7] H. Benl, U. Berger, H. Schwichtenberg, et al. Proof theory at work: Program development in the Minlog system. In W. Bibel and P. G. Schmitt, editors, *Automated Deduction*, volume II. Kluwer, 1998.

[8] Stefan Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Technische Universität München, 2004.

[9] Stefan Berghofer and Tobias Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J McKinna, and R. Pollack, editors, *Types for Proofs and Programs: TYPES'2000*, volume 2277 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[10] 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.

[11] Evert W. Beth. Semantical considerations on intuitionistic mathematics. *Indagationes mathematicae*, 9:572 – 577, 1947.

[12] M. Bickford, R. Constable, and V. Rahli. The Logic of Events, a framework to reason about distributed systems. Technical Report arXiv:1813:28695, CIS, Cornell University, 2012.

[13] Mark Bickford, Robert Constable, and David Guaspari. Generating event logics with higher-order processes as realizers. Technical Report http://hdl.handle.net/1813/23562, Cornell University, 2011.

[14] Mark Bickford and Robert L. Constable. Formal foundations of computer security. In *Formal Logical Methods for System Security and Correctness*, volume 14, pages 29–52, 2008.

[15] E. Bishop and D. Bridges. *Constructive Analysis*. Springer, New York, 1985.

[16] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In *LNCS 5674, Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[17] Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics*. Cambridge University Press, Cambridge, 1988.

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

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

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

[21] Robert Constable and Mark Bickford. Intuitionistic Completeness of First-Order Logic. Technical Report arXiv:1110.1614v3, Computing and Information Science Technical Reports, Cornell University, 2011.

[22] Robert Constable and W. Moczydlowski. Extracting programs from constructive HOL proofs via IZF set-theoretic semantics. In *IJCAR 2006, LNCS 4130*, pages 162–176. Springer, 2006.

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

[24] Robert L. Constable. The semantics of evidence (also appeared as Assigning Meaning to Proofs). *Constructive Methods of Computing Science*, F55:63–91, 1989.

[25] Robert L. Constable. Effectively nonblocking consensus procedures can execute forever: a constructive version of FLP. Technical Report 11512, Cornell University, 2008.

[26] 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.

[27] Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. *Journal of Theoretical Computer Science*, 121:89–112, December 1993.

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

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

[30] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '01, pages 162–174, New York, NY, USA, 2001. ACM.

[31] A. Datta, A. Derek, J.C.Mitchell, and R. Roy. Protocol Composition Logic. *Electronic Notes Theoretical Computer Science*, 172:311–358, 2007.

[32] 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.

[33] Nachum Dershowitz and David Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning Vol 1*, pages 535–610. Elsevier, 2001.

[34] Michael Dummett. The philosophical basis of intuitionistic logic. In H.E. Rose J. Shepherdson, editor, *Logic Colloquium '73*.

[35] Michael Dummett. *Elements of Intuitionism*. Oxford Logic Series. Clarendon Press, 1977.

[36] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the sel4 microkernel. In *VSTTE '08: Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, pages 99–114, Berlin, 2008. Springer.

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

[38] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faculty process. *JACM*, 32:374–382, 1985.

[39] Georges Gonthier. Formal proof - the four color theorem. *Notices of the American Math Society*, 55:1382–1392, 2008.

[40] 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.

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

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

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

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

[45] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In Gries [43], pages 194–204.

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

[47] 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.

[48] Douglas J. Howe. The computational behaviour of Girard's paradox. In Gries [43], pages 205–214.

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

[50] Douglas J. Howe. On computational open-endedness in Martin-Löf's type theory. In LICS91 [62], pages 162–172.

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

[52] Paul B. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Cornell University, Ithaca, NY, January 1995.

[53] Paul B. Jackson. *The Nuprl Proof Development System, Version 4.2 Reference Manual and User's Guide*. Cornell University, 1996.

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

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

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

[57] D. Kozen, C. Kreitz, and E. Eichter. Automating proofs in category theory. In *IJCAR, LNCS 4130*, pages 392–407. Springer, 2006.

[58] G. Kreisel. Weak completeness of intuitionistic predicate logic. *Journal of Symbolic Logic*, 27:139–158, 1962.

[59] Christoph Kreitz. Building reliable, high-performance networks with the Nuprl proof development system. *JFP*, 14(1):21–68, 2004.

[60] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–65, 1978.

[61] Miriam Leeser. Using Nuprl for the verification and synthesis of hardware. *Phil. Trans. Royal Society of London*, 339:49–68, 1992.

[62] *Proceedings of the 6^{th} Symposium on Logic in Computer Science*, Vrije University, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[63] 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. In David Kotz and John Wilkes, editors, *17^{th} ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 33(5) of *Operating Systems Review*, pages 80–92. ACM Press, December 1999.

[64] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[65] 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.

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

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

[68] David McCarty. Completeness and incompleteness for intuitionistic logic. *Journal of Symbolic Logic*, 73(4):1315–1327, 2008.

[69] P.F. Mendler. Recursive types and type constraints in second-order lambda calculus. In Gries [43], pages 30–36.

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

[71] Chetan Murthy. An evaluation semantics for classical proofs. In LICS91 [62], pages 96–109.

[72] Pavel Naumov, Mark-Olivar Stehr, and José Meseguer. The HOL/Nuprl proof translator: A practical approach to formal interoperability. In *TPHOLS 2001, LNCS 2152*, pages 329–345. Springer.

[73] 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.

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

[75] Christine Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In J. F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[76] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1994.

[77] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, Aarhus University, Computer Science Department, Denmark, 1981.

[78] S. Schmitt, L. Lorigo, C. Kreitz, and A. Nogin. JProver: Integrating connection-based theorem proving into interactive proof assistants. In *IJCAR, LNAI 2083*, pages 421–426. Springer, 2001.

[79] 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, New York, 1970.

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

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

[82] A. S.Troelstra. Realizability. In S.R. Buss, editor, *Handbook of Proof Theory*, volume 137 of *Studies in Logic and the Foundations of Mathematics*, pages 407–473. Elsevier, 1998.

[83] George G. Szpiro. *Kepler's Conjecture*. Wiley, 2003.

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

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

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

[87] Valdimir Voevodsky. Notes on type systems. School of Mathematics, Institute for Advanced Study, Princeton, NJ, 2011.

[88] G. Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980.