Computation First: Rebuilding Constructivism with Effects

Liron Cohen
Ben-Gurion University, Beer-Sheva, Israel

Abstract

Constructive logic and type theory have traditionally been grounded in pure, effect-free model of computation. This paper argues that such a restriction is not a foundational necessity but a historical artifact, and it advocates for a broader perspective of *effectful constructivism*, where computational effects, such as state, non-determinism, and exceptions, are directly and internally embedded in the logical and computational foundations. We begin by surveying examples where effects reshape logical principles, and then outline three approaches to effectful constructivism, focusing on realizability models: Monadic Combinatory Algebras, which extend classical partial combinatory algebras with effectful computation; Evidenced Frames, a flexible semantic structure capable of uniformly capturing a wide range of effects; and Effectful Higher-Order Logic (EffHOL), a syntactic approach that directly translates logical propositions into specifications for effectful programs. We further illustrate how concrete type theories can internalize effects, via the family of type theories $\mathsf{TT}_C^{\square}$. Together, these works demonstrate that effectful constructivism is not merely possible but a natural and robust extension of traditional frameworks.

2012 ACM Subject Classification Theory of computation

Keywords and phrases Effectful constructivism, realizability, type theory, monadic combinatory algebras, evidenced frame

Digital Object Identifier 10.4230/LIPIcs.FSCD.2025.1

Category Invited Talk

Funding This work was supported by Grant No. 2020145 from the United States-Israel Binational Science Foundation (BSF).

Acknowledgements The author is deeply grateful to her collaborators in the various parts of this exploration – Robert Constable, Ariel Grunfeld, Dominik Kirst, Étienne Miquey, Bruno da Rocha Paiva, Vincent Rahli, Sam Speight and Ross Tate – for an exceptionally insightful and truly enjoyable research experience.

1 Introduction

Constructive logic and type theory are grounded in a profound principle: that mathematical truth should be computationally meaningful. From Kleene's realizability interpretation [46] to Martin-Löf's type theory [55], constructive foundations have sought to align provability with computability, and logical propositions with programmatic constructions. This strongly manifests in the Curry–Howard correspondence [42,80] which has driven decades of innovation in both logic and programming languages, shaping the design of type theories and enabling the development of interactive theorem provers such as Rocq and Agda [44, 62], and more.

Yet despite the inherently computational nature, foundational constructive frameworks have traditionally recognized only a narrow notion of computation. That is, most constructive theories assume a *pure* model of computation, in which program behaviour can be described as simple input-output transformations (à la mathematical functions). This excludes programs that affect the computational process in ways that are not purely functional, such as state manipulation, exceptions, non-determinism, concurrency, interactions with external environments, etc. This limitation is apparent in both the semantic and syntactic pillars of constructivism. Semantically, realizability theory, which, at its core, concretizes the principle of constructivity by interpreting formulas as specifications for computational entities within a programming language, plays a central role in constructive foundations to provide models for various theories, including Higher-Order Logic (HOL) and Zermelo-Fraenkel set theory (ZF). However, realizability models have traditionally been built over effect-free (up-to non-termination) partial combinatory algebras (PCAs) [75, 78]. Syntactically, type theory, which serves as the syntactic and formal deductive framework for constructive reasoning, lie at the heart of most modern interactive theorem provers, providing not only a logic for mathematical reasoning but also a computational foundation for program verification. However, these systems are typically designed to guarantee strong metatheoretic properties, such as normalization, decidable type-checking, and canonicity of terms, making them inhospitable to effectful constructs like state, exceptions, concurrency, or control. And thus, these foundations generally prevent direct *internal* expression of computational effects, and instead, when effects are needed, they are usually modeled *externally* via monads or abstract interpretation layers, rather than being integrated into the logical core of the system.

But this restriction is not a foundational necessity. Rather, it is a methodological artifact, a reflection of historical preferences for purity due to the importance of the λ-calculus and metatheoretic tractability. With the evolution of programming languages toward effectful computation, this divergence between foundational logic and practical computation has become increasingly untenable. If the Curry–Howard correspondence is to remain meaningful, logic must be capable of expressing computation in all its complexity, not through encodings or external models, but through internal structure. Indeed, the logical structures that underpin constructivism should not offer an idealized abstraction of computation, but rather serve as its foundational account.

There are many potential exciting implications to uprooting the built-in computational limitations and enriching the theory of constructivism with effects. For one, effectful constructivism can be exploited to provide novel computational meaning and implement foundational principles. Recent works have begun to incorporate effects such as memory or nondeterminism, to obtain additional logical structure, shaping the resulting logical theory. For example, Krivine's classical realizability shows that extending the λ-calculus with new programming instructions results in new reasoning principles: `callcc` to obtain classical logic [52], `quote` for Dependent Choice [51], etc. In addition, the principle of Countable Choice (CC) was shown to be implementable using monotonic memory through a method called memoization [12]. The specifics of such computational interpretations can also carry meaningful practical implications for constructive systems. And so in effectful constructivism one can also compare different, logically equivalent, interpretations in terms of their computational behaviour, e.g., efficiency, etc.

Effectful constructivism also reveals a certain volatility of constructive foundations. When we say that a statement is "constructively valid" if it holds in all models, what do we really mean? Do we only refer to the logical substructure of the model or are we also looking at all possible computational models? Indeed, in effectful constructivism the validity of logical principles, such as CC or Markov's Principle, can depend sensitively on the computational substrate. For example, constructive type theories are often claimed to model CC (see, e.g., [1, 74]) simply since they model the relational variant of it due to the fact that the standard realizability interpretation entails that the proof of totality of the relation is in itself a choice function. However, CC can be invalidated by simply adding the computational capability of nondeterministic reallocation in the form of a coin flip [12], while, as noted, extending the computational system with memoization restores CC even in the presence of demonic non-determinism (see Sec. 2.1). Various other works, e.g., [7, 10, 40, 57], employed different methods such as coinduction, lazy evaluation, and infinite terms in a manner similar to memoization, suggesting that it is, somehow, a more robust model of CC. Similarly, Markov's Principle may hold under certain control effects but fail in strictly constructive contexts. These phenomena expose a fragility in the traditional notion of constructive validity: logical truth shifts when the underlying effects change. This calls for a reexamination of notions of constructivism, exploring which "truths" can be made robust with respect to the underlying effects considered and which cannot.

To address this, we need a refined perspective of *robust constructive validity*, according to which a principle is robustly valid across a class of realizability models that share essential computational features, such as monotonic memory, controlled nondeterminism, or continuations. This structural view shifts logical validity from a static property to an invariant across computational architectures. For example, CC may be robustly valid relative to models that support monotonic memory. Establishing this robustness requires additional computational analysis, such as developing a notion of morphism that preserves the underlying computational structure. Such computational morphisms of effectful models will provide more robust constructive foundations, where one can prove that not only does an axiom hold within a particular model, but it also holds in all extensions of that model regardless of what additional effects might be introduced. This, in turn, will lead to extensible constructive systems, in which effectful computations can be used as needed without concern for accidentally breaking prior foundations. This structural view of robust validity aligns logical consequence with computational invariance. Just as formal logic demands explicit axioms, constructive reasoning must make explicit the computational assumptions, effects included, that shape its theory.

These observations are not merely theoretical. In mechanized verification, the mismatch between effectful programs and effect-free logics limits what can be verified *internally*. Therefore, interactive theorem provers based on pure type theories generally require external modeling for any computation that involves effects. This dependency can introduce potential trust issues, as evidenced by several instances where incorrect models led to undetected bugs, e.g., in the verified compiler CompCert due to a flawed formalization of PowerPC semantics [21, 82], and an oversimplified model causing an overlooked unsoundness issue in Java's type system [2, 29, 61]. Alternatively, the fact that many provers are based on constructive theories with a built-in understanding of computation can be exploited to develop and reason about the software *internally*. But to support the internal development of mechanically verified real-world effectful software, the underlying constructive framework must itself support effects internally and natively.

This paper aims to put forward this vision of *effectful constructivism* and ground it in feasibility. For this, we first survey motivating examples from recent discourse where computational effects reshape the logical status of key principles, illustrating both the expressive power and the fragility of theorems under variation in effects (Sec. 2). Having hopefully illustrated to the reader the value and impact of internalizing effects as first-class citizens in constructivism, we then aim to demonstrate that a systematic uniform incorporation of effects is not only feasible, but also quite natural. For this, we outline three semantic realizability frameworks that structurally integrate effects, each in a different manner while all, in some sense, extending traditional realizability-related structures (Sec. 3). We further discuss constructive type theories, focusing on $\mathsf{TT}^{\square}$, a family of extensional type theories that internalize effects using choice operators and modalities (Sec. 4).

All in all, this survey advocates a unified thesis: computation, with all its effects, is not an obstacle to constructive logic, but its proper foundation. *Effects are not noise – they are structure.* Constructivism must be reshaped to make this structure explicit and to harness it for foundational insight.

2 Effects Reshape Logic: Motivating Examples

This section reviews a selection of compelling examples from the literature where computational effects have reshaped logical principles, illustrating how the model of computation directly impacts logical validity. This is by no means an exhaustive review, but rather a focused exploration of key instances where effects and logic intersect in insightful ways. Making these dependencies explicit is crucial for developing robust constructive systems that accurately reflect the computational realities they aim to model.

2.1 Choice Principles via Memoization

The Axiom of Choice (AC), in one of its many formulations, states that every total relation has a corresponding function exhibiting that relation.

$$\big(\forall x : A.\ \exists y : B.\ R(x, y)\big) \Rightarrow \big(\exists f : B^A.\ \forall x : A.\ R(x, fx)\big)$$

While originating from set theory, AC also has various formulations in constructive theories. Whereas the intensional formulation of AC is trivially modelled by many constructive type theories through the standard interpretations of the quantifiers, the extensional formulation is essentially inconsistent with them. This is because the extensional AC implies the (non-constructive) Law of Excluded Middle (LEM) [26]. This is but one of many famous unexpected consequences of (extensional) AC. So while constructive theories cannot implement the full (extensional) AC, one can constructively approach AC. That is, we can provide computational models of variants of the axiom that are equivalent to AC only in the presence of LEM.

One important such variant is that of Countable Choice (CC), in which one restricts attention to relations over, say, the natural numbers (taking $A := \mathbb{N}$). CC is a fundamental principle in constructive real analysis as it is used to unify the different formalizations of the reals. Cauchy reals (of converging sequences of rational numbers) are *not*, in fact, Cauchy complete in constructive real analysis. The Dedekind reals are, but there is no way to convert a Dedekind real to a Cauchy real [54]: while for any given degree of precision one can demonstrate that there *exists* a rational number that approximates the Dedekind real up to that precision, there is no way to construct an infinite sequence of increasingly precise approximations. For this, one needs to use CC.[1]

Now, CC is generally accepted as constructively valid as it holds true in traditional realizability models (stemming from PCAs). But this fact relies on the determinacy of the internal computational system, and, in fact, the proof fails in the presence of non-determinism because the realizer for the totality of the relation no longer necessarily behaves like a deterministic function. Nonetheless, the addition of stateful computation can then restore CC [12]. The introduction of mutable (monotonic) shared state allows for the memoization of the non-deterministic realizer, thereby ensuring that repeated queries for the

[1] In fact, weak CC, in which choice is possible if there is at most one choice to be made across all the countable inputs, is sufficient.

same input would produce the same output. This memoization process is enabled due to the fact that for the natural numbers one can compare a given input with prior inputs for equivalence, a property which does not hold for more general types.

There are a number of systems implementing notions of choice for various *classical* settings, e.g., [7, 10, 40, 57]. Interestingly, despite the difference in goals, these systems use techniques like coinduction, lazy evaluation, and infinite terms in a manner similar to memoization, suggesting that it is, somehow, a more robust interpretation of CC. Whereas in the constructive setting the challenges lie in dealing with impredicativity and non-determinism, in all these systems the challenge lies in being compatible with the complex control constructs used to give constructive interpretations of classical logic [3, 25, 59]. In particular, [40] shows that a mixed-strength existential quantifier can simultaneously be compatible with classical logic while also providing a constructive interpretation of weak Dependent (and Countable) Choice. [57] refines this work and proves strong normalization (and therefore also soundness) of a sequent-style variant of [40]'s logic dPA$^\omega$. [7] implement a negative translation of AC by using a bar-recursion-like operator.

Beyond CC, Dependent Choice (DC), which allows for the construction of sequences where each element depends on the previous one, can also be realized using state.

$$\big(\forall x : A.\ \exists y : A.\ R(x, y)\big) \Rightarrow \big(\exists f : A^{\mathbb{N}}.\ \forall n : \mathbb{N}.\ R(f(n), f(n+1)\big)$$

As noted, Herbelin developed a calculus, dPA$^\omega$, in which constructive proofs of CC and DC can be derived via the memoization of choice functions (in a way that would be impossible in a stateless model [58]). The specifics of such computational interpretations have meaningful practical implications for constructive systems. To see why, consider again the example of the reals. Both *Countable* and *Dependent* Choice can be implemented using mutable state and are sufficient for constructing the infinite sequences needed for Cauchy reals to be Cauchy-complete (so, *logically*, one could choose either one); however, they have different implementation characteristics. The implementation of CC uses an (infinite) array to "remember" non-unique choices as they are made on demand. Each entry in the array is computed independently, resulting in a significant amount of redundant computation and requiring a less efficient choice policy to ensure that the sequence converges. With DC, each entry in the array is computed using the value of the previous entry, avoiding this redundancy and permitting a much more efficient choice policy (e.g. choose something at least twice as precise as the previous entry). In fact, [47] uses a stronger (effectfully computable) principle of *non-deterministic* DC to mechanically verify and extract efficient computations approximating real numbers up to arbitrary precision.

2.2 Classical Logic via Continuations

Classical logic, characterized by principles such as the Law of Excluded Middle (LEM) and Pierce's Law, has traditionally been considered incompatible with constructivism due to its reliance on non-constructive proof techniques. However, a rich line of research has demonstrated that classical logic can, in fact, be constructively interpreted by introducing *continuations*, a concept originating from programming languages, and the control operator `call/cc` (call with current continuation) [3, 25, 38, 59]. Roughly speaking, a continuation is a higher-order function that represents the "rest of the computation" at any given point, effectively capturing the control state of the program. When a continuation is captured (e.g., using `call/cc`), it can be stored, passed around, or invoked multiple times, providing direct control over the execution flow in a manner typically outside the scope of pure functional computation. This mechanism is extremely powerful because it allows for non-local exits from a computation, essentially allowing for the representations of many other effects [32].

The idea of using `call/cc` to interpret classical logic was developed into a robust and highly influential framework by Krivine [52], known as *classical* or *Krivine* realizability. Krivine's framework is based on the λ_c-calculus, an extension of the λ-calculus with a `call/cc` instruction that enables the manipulation of control flow. The Curry–Howard correspondence directly relates `call/cc` to Peirce's law, effectively transforming the computational model into a setting where classical logic can be directly realized. This ability of control operators like `call/cc` to dynamically explore and resolve cases is the computational essence of classical reasoning.

The original presentation of Krivine's realizability uses control operators in a direct-style fashion in which `call/cc` instructions can be compiled to the pure λ-calculus using a continuation-passing style (CPS) translation. Later, Oliva and Streicher demonstrated that Krivine realizability can be obtained by combining a standard intuitionistic realizability interpretation with a CPS translation [35, 56, 63]. Beyond `call/cc`, Krivine further expanded his framework by introducing the `quote` instruction, which computes a natural number associated with the code of a program in a way that allows comparing programs based on their codes [51]. This can be seen as enabling the dynamic generation of choices based on the evolving state of computation, which, in turn, permits the constructive interpretation of principles like DC, where the evolving state of computation can influence the generation of choices.

2.3 Markov's Principle via Control Operators, Exceptions and Monotonic Memory

Markov's Principle (MP) is a central principle in (Russian) constructive mathematics nowadays most commonly stated as follows:

$$\forall f : \mathbb{N} \to \mathbb{B}.\ \neg\neg(\exists n.\ f n = \mathsf{true}) \to \exists n.\ f n = \mathsf{true}.$$

Essentially, it states that Σ^0_1 propositions, i.e. existential quantifications over decidable predicates, are stable under double negation [24, 27, 68].

In Krivine realizability, where control operators such as `call/cc` are available, MP can be realized constructively. Indeed, since continuations provide a form of non-linear control flow, they allow the system to search for an element without requiring a fully constructive witness. Alternatively, Pédrot used exceptions to realize MP within the Calculus of Inductive Constructions (CIC) [65]. Concretely, by extending the type theory with a form of statically-bound exception mechanism, one can extract an existential witness from a proof of its double negation [39].

The subtle additional structure offered by effectful computation opens another critical avenue for constructivism by allowing for differentiating variants of constructive principles. Many key principles, such as MP, have various formalizations that are classically equivalent, but not necessarily in constructive foundations. However, while this is well-known, it is often overlooked, leading to false claims in the constructive discourse. For example, the various non-constructively-equivalent definitions of decidable predicates lead to non-equivalent MPs, resulting in mistakes about the status of MP, e.g., in [65]. A recent work [14] clarified the status of three MP variants: $\mathsf{MP}_{\mathbb{P}}$ (for propositional decidability), $\mathsf{MP}_{\mathbb{B}}$ (for Boolean decidability), and $\mathsf{MP}_{\mathbb{PR}}$ (for primitive recursive functions). The separation of the variants, in the spirit of [24, 68], was done using $\mathsf{TT}^{\square}_{C}$ [18, 20], a generic family of effectful, extensional type theories with a forcing interpretation parameterized by modalities.

The separation of the MP variants is achieved by instantiating the choice operator C, primarily using *choice sequences* of Booleans or propositions (building on ideas going back to [48,49,76]). For example, to refute $\mathsf{MP}_{\mathbb{B}}$, the authors construct a model based on sequences of Boolean choices. In this model, a statement like $\neg\neg\exists n.\, f(n) = \mathtt{true}$ can hold, since it is always possible to make the choice "true" eventually, while $\exists n.\, f(n) = \mathtt{true}$ does not hold because one can construct a path where "true" is never chosen. Similarly, using propositional choices allows for the falsification of $\mathsf{MP}_{\mathbb{P}}$. Crucially, using the $\mathsf{TT}_C^{\square}$ framework one can show that in these same models, variants for effect-free functions or functions with limited effects *do* hold. This distinction reveals how effect-free or limited-effect computations maintain consistent behavior across different possible future worlds (states of choice sequences), unlike general effectful computations.

2.4 Continuity via Reference Cells and Dialogue Trees

Continuity is a fundamental concept in constructive mathematics, traditionally defined as the property that a function from $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ requires only a finite prefix of the input to determine its output.

$$\forall f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}\ \forall \alpha : \mathbb{N}^{\mathbb{N}}\ \exists m : \mathbb{N}\ \forall \beta : \mathbb{N}^{\mathbb{N}}.\ (\forall n \leq m.\ \alpha(n) = \beta(n)) \to f(\alpha) = f(\beta)$$

Continuity is typically justified through semantic arguments (often forcing-based), where functions are shown to respect this finite-dependence property in a model, e.g., [5, 22, 23, 30, 73, 81]. However, recent advancements have demonstrated that continuity can be directly internalized within constructive type theory through the use of various computational effects [13, 19].

One of the most direct methods for realizing continuity within type theory is through the use of *stateful computations*, particularly by leveraging reference cells. In this framework, continuity is not an external property but a constructively enforced behavior: as a function is evaluated, the type theory dynamically tracks the length of the input segment necessary to determine the output. Specifically, a reference cell is used to maintain a record of the smallest initial segment of the input sequence that has been read so far. This mechanism allows the system to efficiently compute the *modulus of continuity* of a function. Functions defined in this manner are constructively continuous by design, and their modulus of continuity is an internal component of their evaluation [19].

An alternative method of realizing continuity is achieved using *inductive dialogue trees*. In this approach, the continuity of a function is represented by the structure of a dynamically generated tree, where the paths of the tree encode information about how much input (an initial segment of a sequence in the Baire space) is needed to determine the output, and the leaves contain the computed values [13]. These dialogue trees are dynamic and evolve based on the function's interactions with its input, providing a general and flexible model for continuity. At each step, the tree structure captures the minimal interaction necessary for the function to produce a value.

The significance of these computational methods lies in their ability to directly enforce and witness continuity within the type theory itself. Rather than appealing to external semantic models, these systems constructively guarantee continuity through their internal effectful mechanisms. This means that the modulus of continuity (the m) is not merely a logical construct but an actively computed value, directly linked to the computational structure of the function. Moreover, the methods are efficient, as they avoid redundant recomputation by dynamically tracking input dependencies.

2.5 And Many More

Paul Cohen's method of forcing is traditionally used to construct models of set theory where specific statements hold or fail. Forcing constructs new models of set theory by adding "generic" elements, allowing for the exploration of various set-theoretic possibilities. One approach to adapting forcing in constructive settings involves simulating the structure of forcing using monotonic memory, i.e., a memory model where the state can only grow or remain the same over time, never decreasing. Avigad demonstrated that this monotonic accumulation of conditions can simulate the behavior of forcing, thereby constructing models where certain propositions hold [4].

Another use of monotonic memory arises in the context of nonstandard analysis, which introduces infinitesimals and infinite numbers to analysis. In constructive mathematics, Dinis and Miquey have shown how realizability interpretations using monotonic memory can provide a constructive framework for nonstandard principles [28]. By extending the λ-calculus with a memory cell that contains an integer (the state), they indicate in which slice of the ultrapower the computation is being done. This stateful realizability interpretation mimics the structure of the ultraproduct by allowing formulas to be interpreted as sets of terms with states, where each state corresponds to a slice. The effectful interpretation is then used to provide realizers for nonstandard reasoning principles, for instance, the idealization principle is realized by a program whose computational behavior involves a diagonalization process that increases the value stored in the state/reference.

The notion of parameterized realizability, introduced by Bauer and Hanson in [6], extends traditional pure realizability by building on a notion of computations that has access to external oracles. Using this framework, the authors constructed a topos in which the Dedekind reals are internally countable. The core idea involves using parameterized partial combinatory algebras where the application of a realizer depends on a parameter from a specified set $\mathbb{P}$. In the specific topos that yields countable reals, this parameter set $\mathbb{P}$ consists of oracles representing non-diagonalizable sequences. The realizers themselves are partial maps that are computable relative to these oracles. Thus, the dependence of computation on an external parameter or oracle allows the chosen non-diagonalizable sequence to serve as an internal enumeration of the reals within the constructed topos.

Computational reflection is another powerful effect used to internalize logical principles. Pédrot has explored the relationship between Church's Thesis (CT) and dependent type theory, particularly Martin-Löf Type Theory (MLTT) [66]. For this, MLTT was extended with quote primitives that relate to execution step counts and validating computation traces, essentially internalizing the runtime behavior of programs within the system. Using this, MLTT can internally realize CT because the new terms yield that for every function, there exists a code, and for every input to that function, there exists a step count and a proof that running the code for that many steps yields the correct result.

3 Effectful Realizability

The diverse examples surveyed so far vividly illustrate how computational effects can fundamentally reshape logical theory, revealing new principles, refining existing ones, and exposing subtle distinctions between related concepts. Yet, these examples have been examined in isolation, with each one highlighting a particular effect in a particular setting and its corresponding logical implications. This fragmented view leaves an important question unanswered: Is there a unified framework capable of systematically capturing the relationship between computational effects and logical principles?

This section explores how computational effects can be directly internalized within a semantic view of constructivism, and specifically the realizability setting. The realizability framework allows us to focus on the semantic models themselves, setting aside the technical complexities that arise when dealing with type theories. By extending realizability models to accommodate effects, we can systematically explore how computational effects impact logical principles, moving beyond isolated examples toward a coherent, unified understanding. We begin by briefly recalling the traditional structure of realizability models stemming from Partial Combinatory Algebras (PCAs) (Sec. 3.1), which serve as the foundation for effect-free realizability. We then briefly review three complementary approaches that generalize this framework to directly incorporate computational effects. First, we provide an algebraic foundation for effectful computation by extending the notion of PCAs to Monadic Combinatory Algebras (MCAs), incorporating monadic effects (Sec. 3.2). However, MCAs are still intrinsically tied to monadic structures and the traditional representation of computational models. Thus, we next consider the more abstract framework of evidenced frames, which generalizes the notion of realizability by abstracting away from any specific computational structure, capturing a broad spectrum of effects through general evidence (Sec. 3.3). Finally, we explore a syntactic perspective to realizability through Effectful Higher-Order Logic (EffHOL), an effectful program logic that can be soundly reduced to the well-established (pure) Higher-Order Logic (Sec. 3.4).

Each of these methods represents a natural and principled extension of traditional realizability, reflecting different aspects of how computational effects can be systematically internalized within a logical model. Our constructions are designed to be sufficiently flexible to uniformly support a diverse array of effectful computations, where realizers themselves can be inherently effectful, such as λ-terms that manipulate state, exhibit nondeterminism, or may fail entirely. Rather than treating effects as external modifications or secondary concerns, we show that they can be directly integrated into the logical structure itself, becoming first-class citizens in the model. This exploration reveals that the limitations of traditional realizability models are indeed not a foundational necessity, as indeed effects can be naturally and seamlessly internalized within realizability models. By systematically extending these models to support computational effects, we show that constructivism can naturally accommodate the full complexity of modern computation.

3.1 Traditional Realizability

Realizability originated from Kleene's interpretation of intuitionistic number theory in the 1940s [46] and has since developed into a large body of work in logic and theoretical computer science where it is used both as a model-theoretic tool and as a method for extracting computational insight. The key idea of realizability is to replace the standard Traskian truth values interpretation of formulas with an interpretation that assigns a set of computational elements called realizers (or evidence) to each formula, with the intuition that the formula dictates the computational behavior of these realizers. In essence, realizability consists of three components: formulas (or types) in some logical framework (e.g., HOL or ZF), a computational system such as the λ-calculus or some combinators algebra, and an interpretation of formulas that connects the former two elements by providing a "truth value" assignment to formulas based on the elements in the computational system. Thus, a realizability model specifies what it means for a proposition to be "realized" or made true by computational objects, and the salient feature of realizability is that these realizers are always computable and that truth values are saturated w.r.t. computational evaluation.

In the original *Kleene's realizability*, the realizers are natural numbers, and a number r realizes a formula φ if r can be interpreted as a program that computes a witness for φ. The realizability relation $r \Vdash \varphi$ is defined as follows:

- $r \Vdash (\varphi \wedge \psi)$ if r is a pair (r_1, r_2) where $r_1 \Vdash \varphi$ and $r_2 \Vdash \psi$.
- $r \Vdash (\varphi \vee \psi)$ if r is a pair (i, r_1), where $i = 0$ and $r_1 \Vdash \varphi$ or $i = 1$ and $r_1 \Vdash \psi$.
- $r \Vdash (\varphi \rightarrow \psi)$ if r is a program that, given any realizer $s \Vdash \varphi$, produces a realizer $r \cdot s \Vdash \psi$.
- $r \Vdash \forall x.\varphi(x)$ if r is a program such that for any $n \in \mathbb{N}$, $r \cdot n \Vdash \varphi(n)$.
- $r \Vdash \exists x.\varphi(x)$ if r is a pair (n, r_1) where $n \in \mathbb{N}$ and $r_1 \Vdash \varphi(n)$.

In this realizability model, logical connectives are interpreted in terms of computational processes: conjunction is realized by a pair of realizers for each conjunct, disjunction is realized by a pair where the first element indicates which disjunct is true, implication is realized by a function that transforms realizers, universal quantification is realized by a program that produces realizers for each instance, and existential quantification is realized by a pair of a witness and a realizer that the witness satisfies the formula.

But while Kleene's original realizability was based on arithmetical realizers, the modern realizability models rely on a computational system of a *Partial Combinatory Algebra (PCA)* [31, 34, 41, 78]. PCAs offer an algebraic model for the untyped λ-calculus, thereby providing an abstract interface that ensures computability (i.e., Turing completeness) without committing to the details of any specific programming language. Concretely, given a set of codes $\mathbb{A}$, a PCA is defined via a (partial) application function $\cdot : \mathbb{A} \times \mathbb{A} \rightharpoonup \mathbb{A}$ satisfying some completeness conditions that ensure they can support general computation, i.e., be at least as computationally powerful as the λ-calculus(often formalized via the definability of the S and K combinators). From a PCA one can obtain a realizability tripos (i.e., a model of HOL, or more generally a higher-order fibration over a cartesian-closed category) by modeling predicates as indexed subsets of codes (i.e. a predicate on a set I specifies for each element $i \in I$ which codes (if any) "realize" that the predicate holds for i) and defining entailment between two predicates to hold whenever there is a uniform code that, for any index, can convert any realizer of the input predicate into a realizer of the output predicate. In turn, via the tripos-to-topos construction [72], from a realizability tripos one can construct a realizability topos, which form the general realizability model of a highly expressive (extensional, impredicative) dependent type theory (and set theory) [43].

However, PCAs are themselves limited in their computational capabilities. In particular, the only computational effect they support *internally* is non-termination through the partiality of the application. PCAs do offer ways in which various effects can be modeled *indirectly*, usually by incorporating additional structures or operations into the underlying algebraic framework. Notable examples include PCAs with errors, choice, exceptions, and parameters, e.g., [6, 12, 17]. However, the ability to *uniformly and internally* support a wide range of computational effects within the algebraic structure is crucial for the development of robust and expressive constructive theories and for reasoning about them collectively.

3.2 Algebraic Approach: Monadic Combinatory Algebras

An immediate manner in which to enrich realizability models with effects is simply to generalize the PCA structure in a way that allows capturing arbitrary effects. For this, the notion of Monadic Combinatory Algebras (MCAs) was introduced, generalizing PCAs but encapsulating a broader spectrum of computational effects by being constructed over some underlying monad [15]. Monads are a common categorical device for analyzing computational effects in the study of programming languages [60, 79]. Concretely, monads are a special kind of functors that allow the composition of Kleisli morphisms, i.e., morphisms where the target is an object in the image of the functor. Using monads, a procedure that takes a value of type A to a value of type B, while possibly invoking some computational effect, can be modeled by a morphism from A to MB, where M is some monad that embodies the effect.

Given a (Set) monad M, a *Monadic Combinatory Algebra (MCA)* over M is a set of "codes" $\mathbb{A}$ with an application Kleisli function: $(-) \cdot (-) : \mathbb{A} \times \mathbb{A} \to M\mathbb{A}$ satisfying some completeness conditions. The key to MCAs lies in that whereas the PCA application function can only return a code (if anything at all), the MCA application Kleisli function returns a *computation*, and so it can produce any computational effect describable by a monad. Indeed, PCA is a special case of an MCA by instantiating the monad with the subsingleton monad, $M\mathbb{A} = \mathcal{P}_1(\mathbb{A})$, i.e. $M\mathbb{A}$ is the set of subsets of $\mathbb{A}$ in which all elements are equal.

Leveraging the monadic structure enables MCAs to internalize various computational effects. For example, non-deterministic computation corresponds to an MCA with M being the *powerset* monad, $M\mathbb{A} = \mathcal{P}(A)$, which considers the subset of possible results. Stateful computation corresponds to an MCA with M being the *powerset state* monad, $M\mathbb{A} = \Sigma \to \mathcal{P}(\Sigma \times \mathbb{A})$, which takes a code in a given state and returns a set of all possible pairs of results in new states. CPS continuations correspond to a monad that represents a computation with direct access to the call stack, allowing the manipulation of the control flow of the programs in non-trivial ways $M\mathbb{A} = (\mathbb{A} \to R) \to R$. Other, more "logical" effects, such as parametric realizability wherein computation has access to an external oracle [6], can be captured via the *subsingleton reader* monad, $M\mathbb{A} = \mathbb{P} \to \mathcal{P}_1(\mathbb{A})$, where $\mathbb{P}$ is a set of some external parameters.

However, while the generalization to MCAs smoothly internalizes effects into the algebra, it also introduces a complexity in the standard pipeline for generating realizability models. This is because, to define a tripos from an MCA one has to define entailment. In traditional realizability, evidence of entailment between two formulas, $\varphi \vdash \psi$, is some code e that, when applied to any realizer of φ produces a realizer for ψ. That is, by brutal abuse of notation: $\forall c_a.\ c_a \Vdash \varphi \;\Rightarrow\; e \cdot c_a \Vdash \psi$. This is well-defined in traditional realizability since when an application is defined, it is deterministic, but this is no longer the case in MCA-based realizability. Thus, for an *arbitrary* monad M, $e \cdot c_a \in M\mathbb{A}$ and it is unclear how to relate it to ψ because $e \cdot c_a$ is now a *computation* and realizability is defined for *codes*, not computations. Thus, one needs to provide an additional structure that converts formulas on codes $\mathbb{A}$ to formulas on computation $M\mathbb{A}$ in a way that respects the monad's structure. This is given by the notion of a M-modality, which intuitively describes a post-condition over the result of a (possibly effectful) computation. Roughly speaking, an M-modality $\diamond$ is a natural transformation that takes predicates on a set X (defined by functions from X to some Heyting prealgebras Ω) and extends them to a function in $M(X) \to \Omega$, intuitively thought of as predicates on $M(X)$. Using the modality we can write $\langle x \leftarrow m \rangle_{\diamond}\, \phi(x)$ stating that after the computation m yields a value x (in case it does), then $\phi(x)$ holds. To obtain a sound logical framework, an M-modality has to satisfy certain properties to ensure it is well-behaved with respect to the computational operators of the monad and the logical operators of the underlying complete Heyting prealgebra. For example, ensuring that properties held before a computation continue to hold afterwards.

As shown in [15], from a "monadic core", which essentially combines a monad with a non-trivial modality, one can construct a realizability tripos by defining entailment between two predicates $\varphi \vdash \psi$ whenever there is a uniform code e (in some combinatory complete subset of the codes) that for every realizer c for φ, the result of the possibly effectful computation $e \cdot c$

realizes ψ, that is, $\langle\!\diamond\, r \leftarrow e \cdot c \,\diamond\!\rangle\, \psi\,(r)$ holds. This shows that the generalized computational system of an MCA still allows for the standard pipeline construction of realizability models while accommodating a wider range of computational effects.

3.3 Semantic Approach: Evidenced Frames

Despite the extensive discourse on realizability, even in its traditional setting, the literature does not offer a universally agreed-upon definition of the general notion of realizability structures. This absence leaves open the question of what exactly constitutes a realizability framework in the most general sense. This section reviews a general and highly abstract structure that pinpoints the common structure of realizability interpretations called Evidenced Frames. Evidenced frames aim to capture the common structural elements of realizability without being restricted to a specific algebraic or computational model. They offer a minimal realizability framework that is flexible enough to allow for internalization of computational effects, even beyond monadic ones [17]. Rather than being a complete framework, an evidenced frame can be understood as a minimal specification that various realizability structures can instantiate, providing a unifying perspective on the relationship between propositions, computation, and logical validity.

Roughly speaking, an evidenced frame is a triple $(\Phi, E, \phi_1 \overset{e}{\to} \phi_2)$ that captures precisely the three key components of realizability: Φ is a set of of propositions, E is a set of evidence, and $\phi_1 \overset{e}{\to} \phi_2$ is an (evidence) relation. To form an evidenced frame, these three components need to satisfy the requirements described in Fig. 1, guaranteeing the existence of basic computational and logical constructs.

While the standard pipeline toward realizability models traditionally stems from a PCA, there is a uniform construction generating a realizability tripos from any given evidenced frame. Moreover, evidenced frames are complete with respect to Set-based triposes. One way to look at this is that realizability triposes are evidenced frames that have forgotten their evidence. Given the fact that a PCA is a simple instance of an evidenced frame, evidenced frames provide an alternative, powerful and flexible foundation for realizability theory. What is more, evidenced frames can uniformly encompass a broad spectrum of computational effects since their level of abstraction generalizes beyond the specific details of any particular model of computation. That is, the evidence can be any computational entities, functions, stateful computations, or non-deterministic procedures, depending on the computational model under consideration.

In particular, the simple and unified nature of evidenced frames makes them a powerful tool for systematic exploration of general metatheorems that link computational capabilities and logical principles. For instance, one can ask, following the examples discussed in Sec. 2.1, whether all realizability structures that support memoization also validate CC. This question becomes concrete within the framework of evidenced frames, where one can identify the minimal set of properties required to support memoization and investigate whether they universally satisfy CC in the resulting tripos. Similarly, evidenced frames provide a natural setting for comparing different methods that validate CC across various computational settings. By examining whether all known approaches, despite their technical differences, can be uniformly captured by an evidenced frame structure, one can uncover a deeper structural connection between them. This connects to the notion of "robust validity", where one aims to show that a logical principle holds across a wide range of computational models that preserve a specific computational structure. Evidenced frames make this notion precise, allowing one to rigorously identify the conditions under which a principle like CC is robustly valid.

Reflexivity An evidence $e_{\mathtt{id}} \in E$ s.t. $\forall \phi.\ \phi \xrightarrow{e_{\mathtt{id}}} \phi$

Transitivity An operator $; \in E \times E \to E$ s.t. $\forall \phi_1, \phi_2, \phi_3, e, e'.\ \phi_1 \xrightarrow{e} \phi_2 \wedge \phi_2 \xrightarrow{e'} \phi_3 \Rightarrow \phi_1 \xrightarrow{e\,;\,e'} \phi_3$

Top A proposition $\top \in \Phi$ and an evidence $e_\top \in E$ s.t. $\forall \phi.\ \phi \xrightarrow{e_\top} \top$

Conjunction Operators $\wedge \in \Phi \times \Phi \to \Phi$ and $\langle\!| \cdot, \cdot |\!\rangle \in E \times E \to E$ and evidence $e_{\mathtt{fst}}, e_{\mathtt{snd}} \in E$ s.t.

- $\forall \phi_1, \phi_2.\ \phi_1 \wedge \phi_2 \xrightarrow{e_{\mathtt{fst}}} \phi_1$
- $\forall \phi_1, \phi_2.\ \phi_1 \wedge \phi_2 \xrightarrow{e_{\mathtt{snd}}} \phi_2$
- $\forall \phi_1, \phi_2, \phi', e_1, e_2.\ \phi' \xrightarrow{e_1} \phi_1 \wedge \phi' \xrightarrow{e_2} \phi_2 \Rightarrow \phi' \xrightarrow{\langle\!| e_1, e_2 |\!\rangle} \phi_1 \wedge \phi_2$

Universal Quantification An operator $\supset \in \Phi \times \mathcal{P}(\Phi) \to \Phi$ such that there exists an operator $\lambda \in E \to E$ and evidence $e_{\mathtt{eval}} \in E$ s.t.:

- $\forall \phi_1, \phi_2, \vec{\phi}, e.\ (\forall \phi \in \vec{\phi}.\ \phi_1 \wedge \phi_2 \xrightarrow{e} \phi) \Rightarrow \phi_1 \xrightarrow{\lambda e} \phi_2 \supset \vec{\phi}$
- $\forall \phi_1, \vec{\phi}, \phi \in \vec{\phi}.\ (\phi_1 \supset \vec{\phi}) \wedge \phi_1 \xrightarrow{e_{\mathtt{eval}}} \phi$

Figure 1 The Properties of an Evidenced Frame.

3.4 Syntactic Approach: Effectful Higher-Order Logic

The syntactic approach to realizability, pioneered by Gödel [37] and further developed in Kreisel's modified realizability [50], abstracts away many of the complex semantic machinery otherwise required for constructing realizability models for rich languages. By restricting to the syntax and abstracting away the details of any particular semantic structure, it allows for a broader spectrum of possible interpretations, each yielding its own realizability interpretation by virtue of being a model of the target language, without having to tailor the realizability construction to some particular structure. Roughly speaking, the syntactic approach is based on handling realizability as a syntactic translation of logical propositions, i.e., statements about mathematical structures, into program specifications, i.e., statements about computational behavior, in a *target program logic* describing what it means to realize the input formula. This can be seen as an internalization of the notion of realizability of the source language into the target language. Recent works adopting the syntactic approach include, e.g., [8, 33, 53, 77]. However, here again, works on syntactic realizability focus on the traditional notion of realizability, which does not offer support for computational effects.

To develop a framework for *syntactic effectful realizability* one can consider a target language that supports effectful programs as realizers for a higher-order source language. *Effectful Higher-Order Logic* (**EffHOL**) provides a concrete, syntactic framework for effectful realizability, where propositions in higher-order logic (HOL) are translated into an effectful target language [16]. Within **EffHOL**, propositions can be interpreted as specifications for effectful programs, and realizers can be typed computational entities capable of interacting with various effects.

To provide internal support for standard programming language features, **EffHOL** combines components of three languages: Higher-Order Logic to model higher-order structure, Girard's System F_ω [36] to model higher-kinded polymorphism, and Evaluation Logic [71] to model computational types and modalities. The computational term language enables reasoning about effectful programs, where here too, as is the case for MCAs, the effectful aspect of the language is captured through monads. This provides a uniform language parameterized intuitively by a monad that carries the computational behavior of the language. Concretely, the type system explicitly includes a type $M(\tau)$ denoting computations of type τ. To describe specifications of effectful programs, **EffHOL** features a modality $\langle_{\diamond} x \leftarrow p \rangle_{\diamond} \varphi$, which intuitively states that property φ holds when x is the result of running the program p. This modality, just like in MCAs, is the core source of effectful computations within **EffHOL**. The higher-kinded polymorphic type system allows for typed realizers of higher-order propositions.

The syntax of **EffHOL** is divided into six distinct components, reflecting its dual nature of handling both computation (programs) and logic (specifications). On the computational side, *kinds* and *types* are used to provide the types of realizers associated with typed *programs*. On the logical side, *indices*, *expressions* and *specifications* hold the logical counterpart of the realizability interpretation by describing the properties of these programs. This clear division between computational and logical components allows **EffHOL** to maintain a robust, expressive syntax that can seamlessly integrate rich computational effects within a logically sound framework.

The syntactic realizability translation of **HOL** to **EffHOL** yields a realizability model of **HOL** from any instance of **EffHOL** for a concrete choice of underlying monad (and an evaluation strategy). Since our target language includes typed realizers, the realizability translation will assign to an **HOL** proposition the type of its realizers in **EffHOL**, along with a specification describing which programs of the corresponding type are realizers of that proposition. The soundness of this translation implies that there is an algorithm that converts any provable **HOL** sequent to an **EffHOL** proof of its translation, which, in turn, contains a computable realizer. These extracted realizers are effectful programs, allowing for the synthesis of such programs from proofs of propositions potentially unprovable in pure **HOL**.

4 Effectful Constructive Type Theories

In the previous section, we explored ways to integrate computational effects into the semantic pillar of constructivism, namely, realizability. We now turn our attention to the type-theoretical pillar of constructivism, where the situation becomes significantly more challenging because, unlike realizability structures which can be generalized with relative ease, type theories must carefully balance two conflicting objectives: expressiveness and meta-theoretical properties. Type theories are not merely abstract models but formal systems with strict syntactic rules, proof mechanisms, and desirable metaproperties such as normalization, canonicity, decidable type checking, and consistency, that can be easily affected by the introduction of effects.

Indeed, a considerable body of work has been devoted to investigating this delicate trade-off, seeking to identify how much effectful computation a type theory can support without undermining its metatheoretical guarantees. In [45] the authors introduce a forcing translation for the (intensional) type theory CIC [64] extended with effects, which relies on storage operators to translate induction principles, and crucially preserves definitional equality. A line of work, starting from [67], involves building syntactic models of CIC by translating CIC extended with logical principles and effects into itself. Using the monadic translation of dependent type theory that allows effects presented in [67], in [11] the authors present syntactic models through which properties can be added to negative types, allowing them to prove independent results, e.g., the independence of function extensionality in intentional type theory. In [68], the authors present a translation of CIC into itself, where the resulting type theory features exceptions, which is consistent if the target theory is *and* exceptions are required to be caught locally. The authors use this translation to exhibit syntactic models of CIC which validate the Independence of Premise axiom, but not Markov's Principle. In [70], the authors solve this problem of the restriction on exceptions in [68] by introducing a layered type theory with exceptions, which separates the consistency and effectful programming concerns. In [65] the authors present a syntactic presheaf model of CIC, which solves issues with dependent elimination present in [45], and allows extending CIC with MP. In [69], the authors go back to these dependent elimination issues and present a new version of call-by-push-value which allows combining effects and dependent types, and show why dependent elimination must be restricted in call-by-name, and substitution in call-by-value, when the calculus features effects.

Here, rather than attempting to construct a fully general and uniform framework for effectful type theory, we focus on an instructive example: $\mathsf{TT}_C^\square$, which is a generic family of effectful, extensional type theories [18, 20]. In a nutshell, $\mathsf{TT}_C^\square$ provides a unified, modular framework for capturing a wide range of computational effects through the concept of time-progressing choice operators C, a general mechanism for representing computational effects such as reference cells (modeling mutable state), choice sequences (modeling non-determinism), or exceptions. To govern the structure of the effect, $\mathsf{TT}_C^\square$ uses a general possible-worlds forcing interpretation parameterized by an abstract modality $\square$, which, in turn, can be instantiated with simple covering relations, leading to a general sheaf model. Intuitively, each world represents a state of computation, and the transition between worlds models the evolution of computational state over time. The forcing conditions are the worlds, and the interpretation of types and terms (equality in a type) depends on the current world and the modality. This structure allows the meaning of a term or type to vary depending on the state of the world, which can change due to effects.

Unlike traditional type theories, where purity is the default, types in $\mathsf{TT}_C^\square$ are impure by default. Thus, computational effects are the norm for terms inhabiting types, rather than the exception. But $\mathsf{TT}_C^\square$ includes specific constructs for controlling the scope of effects. Thus, for example, while a basic type like $\mathbb{N}$ represents potentially "read & write" numbers where terms can modify the world and compute to different values based on reads, types can also be restricted to "write-only", where terms must compute to the same value across world extensions, "read-only", where computations must start and end in the same world, or completely pure.

The controlled effects, together with the flexibility in the instantiations of the modality and the choice operator, allow for a fined-grained assessment of the resulting system in terms of expressivness and metatheoretic properties. Moreover, the uniformity and versatility of $\mathsf{TT}_C^\square$ has enabled a series of results demonstrating how the system can constructively interpret various logical principles via effectful instantiations of the system. For example, LEM [9, 18], Continuity (see Sec. 2.4) etc, and to separate various principles such as MP (see Sec. 2.3).

5 Conclusion

This paper explored the vision of *effectful constructivism*, where computational effects are recognized as an integral part of logical systems. Rather than viewing computational effects as external complications to be modeled or mitigated, effectful constructivism recognizes them as an intrinsic aspect of logical reasoning. As modern programming languages have progressed to support rich and varied computational effects, the logical structures underpinning constructivism must also reflect this reality. Our exploration has shown how effectful constructivism naturally extends traditional frameworks, revealing that logical principles are inherently sensitive to the underlying computational capabilities. By internalizing effects, effectful constructivism deepens the Curry–Howard correspondence and opens the door to a broader spectrum of logical systems, semantic models, and verified applications. Computation, with all its effects, is not an obstacle, but the true foundation of constructivism.

References

1 Peter Aczel. The Type Theoretic Interpretation of Constructive Set Theory: Choice Principles. In *Studies in Logic and the Foundations of Mathematics*, volume 110, pages 1–40. Elsevier, 1982.

2 Nada Amin and Ross Tate. Java and Scala's Type Systems Are Unsound: The Existential Crisis of Null Pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 838–848, New York, NY, USA, 2016. ACM. doi:10.1145/2983990.2984004.

3 Zena M. Ariola, Paul Downen, Hugo Herbelin, Keiko Nakata, and Alexis Saurin. Classical Call-by-Need Sequent Calculi: The Unity of Semantic Artifacts. In Tom Schrijvers and Peter Thiemann, editors, *Functional and Logic Programming - 11th International Symposium, FLOPS*, Lecture Notes in Computer Science, pages 32–46. Springer, 2012. doi:10.1007/978-3-642-29822-6.

4 Jeremy Avigad. Forcing in Proof Theory. *The Bulletin of Symbolic Logic*, 10(3):305–333, 2004. doi:10.2178/BSL/1102022660.

5 Martin Baillon, Assia Mahboubi, and Pierre-Marie Pédrot. Gardening with the Pythia A Model of Continuity in a Dependent Setting. In Florin Manea and Alex Simpson, editors, *CSL*, volume 216 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CSL.2022.5.

6 Andrej Bauer and James E. Hanson. The Countable Reals, 2024. arXiv:2404.01256.

7 Stefano Berardi, Marc Bezem, and Thierry Coquand. On the Computational Content of the Axiom of Choice. *J. Symb. Log.*, 63(2):600–622, 1998. doi:10.2307/2586854.

8 Jean-Philippe Bernardy and Marc Lasson. Realizability and Parametricity in Pure Type Systems. In *Foundations of Software Science and Computational Structures: 14th International Conference, FOSSACS*, pages 108–122. Springer, 2011. doi:10.1007/978-3-642-19805-2_8.

9 Mark Bickford, Liron Cohen, Robert L. Constable, and Vincent Rahli. Open Bar—a Brouwerian Intuitionistic Logic with a Pinch of Excluded Middle. In Christel Baier and Jean Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic (CSL 2021)*, volume 183, pages 11:1–11:23, 2021. doi:10.4230/LIPIcs.CSL.2021.11.

10 Valentin Blot and Colin Riba. On Bar Recursion and Choice in a Classical Setting. In *Programming Languages and Systems - 11th Asian Symposium, APLAS*, pages 349–364, December 2013. doi:10.1007/978-3-319-03542-0_25.

11 Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The Next 700 Syntactical Models of Type Theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 182–194, 2017. doi:10.1145/3018610.3018620.

12 Liron Cohen, Sofia Abreu Faro, and Ross Tate. The Effects of Effects on Constructivism. In *Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2019, London, UK, June 4-7, 2019*, volume 347, pages 87–120, 2019. Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics. doi:10.1016/j.entcs.2019.09.006.

13 Liron Cohen, Bruno da Rocha Paiva, Vincent Rahli, and Ayberk Tosun. Inductive Continuity via Brouwer Trees. In Jérôme Leroux, Sylvain Lombardy, and David Peleg, editors, *48th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 272, pages 37:1–37:16, 2023. doi:10.4230/LIPIcs.MFCS.2023.37.

14 Liron Cohen, Yannick Forster, Dominik Kirst, Bruno Da Rocha Paiva, and Vincent Rahli. Separating Markov's Principles. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '24, 2024. doi:10.1145/3661814.3662104.

15 Liron Cohen, Ariel Grunfeld, Dominik Kirst, and Étienne Miquey. From Partial to Monadic: Combinatory Algebra with Effects. In Maribel Fernández, editor, *10th International Conference on Formal Structures for Computation and Deduction (FSCD)*, volume 337 of *LIPIcs*, 2025.

16 Liron Cohen, Ariel Grunfeld, Dominik Kirst, and Étienne Miquey. Syntactic Effectful Realizability in Higher-Order Logic. In *40th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2025.

17 Liron Cohen, Étienne Miquey, and Ross Tate. Evidenced Frames: A Unifying Framework Broadening Realizability Models. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2021. `doi:10.1109/LICS52264.2021.9470514`.

18 Liron Cohen and Vincent Rahli. Constructing Unprejudiced Extensional Type Theories with Choices via Modalities. In *FSCD*, volume 228 of *LIPIcs*, pages 10:1–10:23, 2022. `doi:10.4230/LIPIcs.FSCD.2022.10`.

19 Liron Cohen and Vincent Rahli. Realizing Continuity Using Stateful Computations. In Bartek Klin and Elaine Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic (CSL)*, volume 252, pages 15:1–15:18, 2023. `doi:10.4230/LIPIcs.CSL.2023.15`.

20 Liron Cohen and Vincent Rahli. $\mathrm{TT}^{\square}_{\mathcal{C}}$: a Family of Extensional Type Theories with Effectful Realizers of Continuity. *Logical Methods in Computer Science*, Volume 20, Issue 2, June 2024. `doi:10.46298/lmcs-20(2:18)2024`.

21 The CompCert Project. URL: `http://compcert.inria.fr/`.

22 Thierry Coquand and Guilhem Jaber. A Note on Forcing and Type Theory. *Fundam. Inform.*, 100(1-4):43–52, 2010. `doi:10.3233/FI-2010-262`.

23 Thierry Coquand and Guilhem Jaber. A Computational Interpretation of Forcing in Type Theory. In P. Dybjer, Sten Lindström, Erik Palmgren, and G. Sundholm, editors, *Epistemology versus Ontology: Essays on the Philosophy and Foundations of Mathematics in Honour of Per Martin-Löf*, pages 203–213. Springer Netherlands, Dordrecht, 2012. `doi:10.1007/978-94-007-4435-6_10`.

24 Thierry Coquand and Bassel Mannaa. The Independence of Markov's Principle in Type Theory. *Log. Methods Comput. Sci.*, 13(3), 2017. `doi:10.23638/LMCS-13(3:10)2017`.

25 Pierre-Louis Curien and Hugo Herbelin. The Duality of Computation. In *ICFP*, 2000. `doi:10.1145/351240.351262`.

26 Radu Diaconescu. Axiom of Choice and Complementation. *Proceedings of the American Mathematical Society*, 51(1):176–178, 1975. URL: `http://www.jstor.org/stable/2039868`.

27 Hannes Diener. Constructive Reverse Mathematics, 2020. `arXiv:1804.05495`.

28 Bruno Dinis and Étienne Miquey. Realizability with Stateful Computations for Nonstandard Analysis. In *Computer Science Logic*, January 2021. URL: `https://hal.science/hal-03002239`.

29 Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java Type System Sound? *Theory and practice of object systems*, 5(1):3–24, 1999.

30 Martín Hötzel Escardó. Continuity of Gödel's System T Definable Functionals via Effectful Forcing. *Electr. Notes Theor. Comput. Sci.*, 298:119–141, 2013. `doi:10.1016/j.entcs.2013.09.010`.

31 Solomon Feferman. A Language and Axioms for Explicit Mathematics. In *Algebra and logic*, pages 87–139. Springer, 1975.

32 Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1996.

33 Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. Verified Extraction from Coq to OCaml. *Proceedings of the ACM on Programming Languages*, 8(PLDI):52–75, 2024. `doi:10.1145/3656379`.

34 Jonas Frey. Characterizing Partitioned Assemblies and Realizability Toposes. *Journal of Pure and Applied Algebra*, 223(5):2000–2014, 2019. `doi:10.1016/j.jpaa.2018.08.012`.

35 Samuel Gardelle and Étienne Miquey. Do CPS Translations Also Translate Realizers? In Timothy Bourke and Delphine Demange, editors, *JFLA 2023 - 34èmes Journées Francophones des Langages Applicatifs*, pages 103–120, January 2023. URL: `https://hal.inria.fr/hal-03910311`.

36 Jean-Yves Girard. *Interprétation Fonctionnelle Et élimination Des Coupures De L'arithmétique D'ordre Supérieur.* PhD thesis, Université Paris Diderot - Paris 7, 1972.

37 Von Kurt Gödel. Über eine bisher noch nicht benützte erweiterung des finiten standpunktes. *dialectica*, 12(3-4):280–287, 1958.

38 Timothy Griffin. A Formulae-as-type Notion of Control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 47–58, New York, NY, USA, 1990. ACM. `doi:10.1145/96709.96714`.

39 Hugo Herbelin. An Intuitionistic Logic that Proves Markov's Principle. In *2010 25th Annual IEEE Symposium on Logic in Computer Science.* IEEE, 2010. `doi:10.1109/lics.2010.49`.

40 Hugo Herbelin. A Constructive Proof of Dependent Choice, Compatible with Classical Logic. In *LICS*, 2012. URL: `https://hal.inria.fr/hal-00697240`.

41 Pieter JW Hofstra. Partial Combinatory Algebras and Realizability Toposes. *University of Ottowa*, 2004.

42 William Alvin Howard. The Formulae-as-Types Notion of Construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, 1980.

43 J. Martin E. Hyland, Peter T. Johnstone, and Andrew M Pitts. Tripos Theory. In *Mathematical Proceedings of the Cambridge philosophical society*, volume 88, pages 205–232, 1980.

44 INRIA. *The Coq Proof Assistant Reference Manual*, 2023.

45 Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot, Matthieu Sozeau, and Nicolas Tabareau. The Definitional Side of the Forcing. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 367–376. Association for Computing Machinery, 2016. `doi:10.1145/2933575.2935320`.

46 Stephen C. Kleene. On the Interpretation of Intuitionistic Number Theory. *Journal of Symbolic Logic*, 10(4):109–124, 1945. `doi:10.2307/2269016`.

47 Michal Konečný, Sewon Park, and Holger Thies. Axiomatic Reals and Certified Efficient Exact Real Computation. In *Logic, Language, Information, and Computation: 27th International Workshop, WoLLIC*, pages 252–268, Berlin, Heidelberg, 2021. `doi:10.1007/978-3-030-88853-4_16`.

48 Georg Kreisel. A Remark on Free Choice Sequences and the Topological Completeness Proofs. *The Journal of Symbolic Logic*, 23(4):369–388, 1958. `doi:10.2307/2964012`.

49 Georg Kreisel. The Non-derivability of $\neg(x)A(x) \to (\exists x)\neg(Ax)$, $A(x)$ primitive recursive, in intuitionistic formal systems (abstract). *Jour. Symb. Logic, 23(4):456–457*, 1958.

50 Georg Kreisel. Interpretation of Analysis by Means of Constructive Functionals of Finite Types. In A. Heyting, editor, *Constructivity in mathematics*, pages 101–128. North-Holland Pub. Co., 1959.

51 Jean-Louis Krivine. Dependent choice, 'Quote' and the Clock. *Th. Comp. Sc.*, 308:259–276, 2003. `doi:10.1016/S0304-3975(02)00776-4`.

52 Jean-Louis Krivine. Realizability in Classical Logic. *Panoramas et synthèses*, 27:197–229, 2009. URL: `https://hal.science/hal-00154500`.

53 Pierre Letouzey. A New Extraction for Coq. In *International Workshop on Types for Proofs and Programs*, pages 200–219. Springer, 2002. `doi:10.1007/3-540-39185-1_12`.

54 Robert S. Lubarsky. On the Cauchy Completeness of the Constructive Cauchy Reals. *Electron. Notes Theor. Comput. Sci.*, 167:225–254, January 2007. `doi:10.1016/j.entcs.2006.09.012`.

55 Per Martin-Löf. *Constructive Mathematics and Computer Programming.* Oxford University Press, 1982.

56 Alexandre Miquel. Existential Witness Extraction in Classical Realizability and via a Negative Translation. *Logical Methods in Computer Science*, Volume 7, Issue 2, April 2011. `doi:10.2168/LMCS-7(2:2)2011`.

57 Étienne Miquey. A Sequent Calculus with Dependent Types for Classical Arithmetic. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 720–729, 2018. `doi:10.1145/3209108.3209199`.

58 Étienne Miquey. A Constructive Proof of Dependent Choice in Classical Arithmetic via Memoization, 2019. `arXiv:1903.07616`.

59 Étienne Miquey and Hugo Herbelin. Realizability Interpretation and Normalization of Typed Call-by-Need λ-Calculus with Control. In *FoSSaCS*, 2018. `doi:10.1007/978-3-319-89366-2_1`.

60 Eugenio Moggi. Notions of Computation and Monads. *Information and computation*, 93(1):55–92, 1991. `doi:10.1016/0890-5401(91)90052-4`.

61 Tobias Nipkow and David von Oheimb. Javalight is Type-Safe—Definitely. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 161–170, 1998. `doi:10.1145/268946.268960`.

62 Ulf Norell. Towards a Practical Programming Language Based on Dependent Type Theory. *PhD thesis, Chalmers University of Technology*, 2007.

63 P. Oliva and T. Streicher. On Krivine's Realizability Interpretation of Classical Second-Order Arithmetic. *Fundam. Inform.*, 84(2):207–220, 2008. URL: `http://iospress.metapress.com/content/f51774wm73404583/`.

64 Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015. URL: `https://hal.inria.fr/hal-01094195`.

65 Pierre-Marie Pédrot. Russian Constructivism in a Prefascist Theory. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany*, pages 782–794. ACM, 2020. `doi:10.1145/3373718.3394740`.

66 Pierre-Marie Pédrot. "Upon This Quote I Will Build My Church Thesis". In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '24, 2024. `doi:10.1145/3661814.3662070`.

67 Pierre-Marie Pédrot and Nicolas Tabareau. An Effectful Way to Eliminate Addiction to Dependence. In *Logic in Computer Science (LICS), 2017 32nd Annual ACM/IEEE Symposium on*, page 12, 2017. `doi:10.1109/LICS.2017.8005113`.

68 Pierre-Marie Pédrot and Nicolas Tabareau. Failure is Not an Option - An Exceptional Type Theory. In *27th European Symposium on Programming*, volume 10801 of *LNCS*, pages 245–271. Springer, April 2018. `doi:10.1007/978-3-319-89884-1_9`.

69 Pierre-Marie Pédrot and Nicolas Tabareau. The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects. *Proc. ACM Program. Lang.*, 4(POPL):58:1–58:28, 2020. `doi:10.1145/3371126`.

70 Pierre-Marie Pédrot, Nicolas Tabareau, Hans Jacob Fehrmann, and Éric Tanter. A Reasonably Exceptional Type Theory. *Proc. ACM Program. Lang.*, 3(ICFP):108:1–108:29, 2019. `doi:10.1145/3341712`.

71 Andrew M Pitts. Evaluation Logic. In *IV Higher Order Workshop, Banff 1990: Proceedings of the IV Higher Order Workshop*, pages 162–189. Springer, 1991.

72 Andrew M Pitts. Tripos Theory in Retrospect. *Mathematical structures in computer science*, 12(3):265–279, 2002. `doi:10.1017/S096012950200364X`.

73 Vincent Rahli and Mark Bickford. A Nominal Exploration of Intuitionism. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2016, pages 130–141, 2016. `doi:10.1145/2854065.2854077`.

74 Michael Rathjen. Choice Principles in Constructive and Classical Set Theories. In *Logic Colloquium*, volume 2, pages 299–326, 2002.

75 Anne S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer-Verlag, 1973.

76 Mark van Atten. *On Brouwer*. Wadsworth Philosophers. Cengage Learning, 2004.

77 Jaap van Oosten. The Modified Realizability Topos. *Journal of pure and applied algebra*, 116(1-3):273–289, 1997.

78 Jaap Van Oosten. *Realizability: an Introduction to its Categorical Side*, volume 152. Elsevier, Amsterdam, 2008.

79 Philip Wadler. Monads for Functional Programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995. doi:10.1007/3-540-59451-5_2.

80 Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015. doi:10.1145/2699407.

81 Chuangjie Xu. *A Continuous Computational Interpretation of Type Theories*. PhD thesis, University of Birmingham, UK, 2015. URL: http://etheses.bham.ac.uk/5967/.

82 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294. ACM, 2011. doi:10.1145/1993498.1993532.