# Syntactic Effectful Realizability in Higher-Order Logic


Liron Cohen
*Ben-Gurion University*
Be'er Sheva, Israel
cliron@bgu.ac.il

Ariel Grunfeld
*Ben-Gurion University*
Be'er Sheva, Israel
arielgru@post.bgu.ac.il

Dominik Kirst
*Ben-Gurion University, Israel*
*Inria Paris, France*
dominik.kirst@inria.fr

Étienne Miquey
*Aix Marseille Univ., CNRS, I2M*
Marseille, France
etienne.miquey@univ-amu.fr



***Abstract*—Realizability interprets propositions as specifications for computational entities in programming languages. Specifically, syntactic realizability is a powerful machinery that handles realizability as a syntactic translation of propositions into new propositions that describe what it means to realize the input proposition. This paper introduces EffHOL (Effectful Higher-Order Logic), a novel framework that expands syntactic realizability to uniformly support modern programming paradigms with side effects. EffHOL combines higher-kinded polymorphism, enabling typing of realizers for higher-order propositions, with a computational term language that uses monads to represent and reason about effectful computations. We craft a syntactic realizability translation from (intuitionistic) higher-order logic (HOL) to EffHOL, ensuring the extraction of computable realizers through a constructive soundness proof. EffHOL's parameterization by monads allows for the synthesis of effectful realizers for propositions unprovable in pure HOL, bridging the gap between traditional and effectful computational paradigms. Examples, including continuations and memoization, showcase EffHOL's capability to unify diverse computational models, with traditional ones as special cases. For a semantic connection, we show that any instance of EffHOL induces an evidenced frame, which, in turn, yields a tripos and a realizability topos.**


## I. Introduction

Realizability, rooted in the works of Kleene [1], aims to concretize the principle of constructivity by interpreting propositions as specifications for computational entities within a programming language. A key feature of realizability is its assurance that the evidence for every proposition is computable. Originally, realizability used natural numbers to interpret formulas, but to get a more general notion of realizability, its modern notion generally uses some complete code language for realizing formulas. In the constructive case, it is standardly based on the notion of Partial Combinatory Algebras (PCAs) [2], while Krivine classical realizability uses an abstract notion of stack machines [3].

However, both the constructive and classical approaches are based on semantic constructions to yield models defined as a tripos or a topos [4]. On the other hand, the syntactic approach to realizability, pioneered by Gödel [5] and further developed in Kreisel's modified realizability [6], can be seen as abstracting away many of the complex semantic machinery otherwise required for constructing realizability models for rich languages. By restricting to the syntax and abstracting away all the nifty details of any particular semantic structure, this also 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, it is based on handling realizability as a syntactic translation of formulas into new formulas 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., [7]–[12].

Yet, works on *syntactic* realizability tend to focus on the traditional notion of realizability or single computational effects. However, features of modern programming languages include multiple effects like non-determinism, stateful computation, probabilistic computation, etc.

To this end, much work has been devoted to the extension of the notion of realizability to allow for side effects, e.g., [9, 13]–[24]. Other than providing support for richer programming languages, it was also shown that effectful realizability provides new models, which, in turn, can implement new features. For instance, with the development of classical realizability continuing work of Griffin [25], Krivine evidences the fact that extending the $\lambda$-calculus with new programming instructions may result in new reasoning principles: `call/cc` to get classical logic [3], `quote` for dependent choice [26], etc.

Accordingly, this work provides a framework for syntactic effectful realizability. That is, we extend the syntactic realizability approach by considering a target language that supports effectful programs as realizers for a higher-order source language. Concretely, we present a framework, dubbed **EffHOL** for Effectful Higher-Order Logic, that combines two key features: higher-kinded polymorphism and a computational term language. The higher-kinded polymorphic type system, inspired by Girard's System $F_\omega$ [27], allows for typed realizers of higher-order propositions. The computational term language, which can be seen as a simplification of Pitts' Evaluation Logic [28], enables reasoning about effectful programs in the sense of a general program logic (cf. [29]). In


Cohen, Grunfeld and Kirst were partially supported by Grant No. 2020145 from the United States-Israel Binational Science Foundation (BSF). Kirst also received funding from the European Union's Horizon research and innovation programme under the Marie Skłodowska-Curie grant agreement No.101152583 and a Minerva Fellowship of the Minerva Stiftung Gesellschaft für die Forschung mbH.

particular, to support effectful realizability, we adhere to the standard approach for reasoning about effectful programs via *monads* [30]. This provides a uniform language parameterized intuitively by a monad that carries the computational behavior of the language. By providing internal support for standard program language features, for example by having typed realizers and effectful programs, **EffHOL** is applicable to a broad range of languages, and reasoning about realizers is done in a natural manner, similar to the way one would reason about programs. Traditional systems like System $F_\omega$ and computational $\lambda$-calculus are subsystems of **EffHOL**, illustrating the versatility and robustness of the system.

Concretely, we show how to model (intuitionistic) higher-order logic [31] (**HOL**) using **EffHOL** by providing a syntactic realizability translation from **HOL** into **EffHOL**. The realizability translation assigns 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 said proposition. A key feature of our syntactic realizability translation is that when translating a provable **HOL** sequent, it provides an algorithm for constructing an **EffHOL** proof of its translation, which, in turn, contains a realizer. Thus, our translation can be seen as synthesizing **HOL** realizers within **EffHOL**. Furthermore, as **EffHOL** is parameterized by a monad, the synthesized realizer obtained from the soundness proof is agnostic to the specifics of the monad. That is, the **HOL** realizers stemming from the soundness proof do not make actual use of the specific behavior of the monad. However, as we show via a few illustrative examples, we can take advantage of the monad to get effectful realizers for propositions that are not provable in pure **HOL**.

To obtain a semantic connection, we link our syntactic realizability translation to the framework of evidenced frames [18], which are abstract algebraic structures from which one can construct effectful realizability models (i.e., triposes) via a uniform construction. Here we show that for each instance of **EffHOL** with a concrete choice of monad, the realizability translation induces an evidenced frame, thereby associating our realizability translation with the tripos models.

*Outline and Main Contributions:*

- Sec. II provides a high-level, intuitive overview of the construction of **EffHOL** and the realizability translation, emphasizing the reasoning behind each component and the interplay between them.
- Sec. III fixes the formalism of higher-order logic used in this paper.
- Sec. IV presents **EffHOL**, a higher-order logic that combines higher-kinded polymorphism and computational types, thus allowing higher-order reasoning about effectful programs.
- Sec. V provides a realizability translation of **HOL** to **EffHOL**, yielding a realizability model of **HOL** from any instance of **EffHOL**, and gives a constructive soundness proof for the translation, extracting **EffHOL** programs from **HOL** proofs.
- Sec. VI demonstrates the utility and generality of **EffHOL** via illustrative examples, including continuations and memoization.
- Sec. VII relates our syntactic approach to realizability with the semantic ones by showing how the realizability translation induces a structure of an evidenced frame.
- Sec. VIII discusses related works and Sec. IX concludes.

We supplement our development with a Rocq mechanization (https://github.com/dominik-kirst/effhol), and mechanized results are indicated with clickable icons.

## II. Overview of the Realizability Framework

The realizability approach to semantics relates logic with computation by interpreting propositions as denoting specifications of computer programs, or equivalently, using computer programs to demonstrate the validity of propositions. While the approach is very general, with many variations, at its core it usually relates a specific logic with a specific programming language by translating each proposition in the logic to a specification for programs of this programming language. The wide literature on realizability interpretation introduces numerous very different such interpretations with different purposes. A recent work [18] aiming to pinpoint the common structure of these interpretations identified a structure, dubbed evidenced frame, mathematically defined as a triple $(E, \Phi, \Vdash)$ precisely capturing these three components : $E$ being the set of so-called evidences (the programs that serve as realizers), $\Phi$ the set of formulas (i.e. the logic that is being interpreted), while $\Vdash$ is the realizability relation connecting the former components. Many realizability interpretations usually define this realizability relation $\cdot \Vdash \cdot$ *externally*, in the meta-theory.

Following the idea that this relation can be understood as defining specifications for programs to be adequate realizers of the corresponding formulas, we make this slogan more concrete by relying on a program logic to define such specifications. In line with Kreisel's modified realizability [6], we formally consider a realizability interpretation as a translation: it then translates propositions, i.e. statements about mathematical structures, as specifications in our target logic, i.e. statements about computer programs.

$$\textit{Logical Proposition} \xrightarrow{\text{realizability translation}} \textit{Program Specification}$$

By carefully defining a general enough target language, which we call *Effectful Higher-Order Logic (***EffHOL***)* (introduced in Sec. IV), it turns out that not only does our approach encompass usual realizability interpretations, but it also provides us with an even more general framework, compatible with typed realizers, effectful programs, etc. To better grasp intuitions on how these features fit into the picture, let us first recall through a simple example how realizability works. Consider for instance a simple tautology $\Phi$ expressing that the conjunction is, from a logical point of view, commutative:

$$\Phi \triangleq (\Phi_1 \wedge \Phi_2) \supset (\Phi_2 \wedge \Phi_1)$$

Following the core intuition of realizability as depicted by the BHK interpretation [32], a realizer of a conjunction is intended to be a program providing a pair of realizers, one for

$\Phi_1$ and one for $\Phi_2$, while a realizer of an implication is meant to compute out of a realizer of the premise a realizer of the conclusion. Formally, this takes the form of a translation of any proposition $\Phi$ into a specification $[\![\Phi]\!]^{\mathrm{S}}$ mapping programs to propositions, $[\![\Phi]\!]^{\mathrm{S}}(r)$ expressing that $r$ is a realizer of $\Phi$. A realizer of this proposition is expected, out of a program computing a pair of realizers for $\Phi_1$ and $\Phi_2$, to provide a computation that produces a pair of realizers for $\Phi_1$ and $\Phi_2$:

$$[\![\varphi \supset \psi]\!]^{\mathrm{S}}(t) \triangleq \forall c.\left([\![\varphi]\!]^{\mathrm{S}}(c) \supset [\![\varphi]\!]^{\mathrm{S}}(t\,c)\right)$$
$$[\![\varphi_1 \wedge \varphi_2]\!]^{\mathrm{S}}(c) \triangleq [\![\varphi_1]\!]^{\mathrm{S}}(\pi_1\,c) \wedge [\![\varphi_2]\!]^{\mathrm{S}}(\pi_2\,c)$$

For instance, using a $\lambda$-calculus with primitive pairs and projections as the language of realizers, the term $r \triangleq \lambda x.\langle \pi_2\, x, \pi_1\, x\rangle$ would do the job: formally, with the definitions above, $[\![\Phi]\!]^{\mathrm{S}}(r)$ holds. In fact, since such a term does not rely on a specific choice of $\Phi_1$ and $\Phi_2$ (or of their interpretations), this term would be a realizer for the same proposition if $\Phi_1$ and $\Phi_2$ were universally quantified. Moreover, depending on the choice of the set of programs that may serve as realizers, many other programs may realize the same proposition. In particular, realizers need not be purely functional (they may use, e.g., printing instructions, increase some references, etc) and there are no rules as to how that ordered pair has to be computed (right-to-left, left-to-right, in parallel?), what representation it should have (are pairs primitive, encoded?), or even whether an actual value of an ordered pair is ever given (or only a program that may compute such a pair). Before introducing in detail our target logic **EffHOL**, we first give an overview of some of its features and how they allow our framework to account for a large spectrum of realizability interpretations.

### A. Typed realizers

Realizability is commonly based on an untyped notion of computer programs, where realizers of a given proposition $\Phi$ are characterized as being the computationally well-behaved programs (w.r.t. the specification $\Phi$ provides). Notably, when the languages for propositions and types coincide, typed programs are shown to be realizers of their types (but not the only ones) in what is usually called the *adequacy lemma* of realizability interpretation. However, certain settings necessitate a clear distinction between the language of propositions that are interpreted and the language of types. This is for instance the case in Blot's interpretation of second-order arithmetic using programs in an extension of PCF [33], or in Paulin-Mohring's realizability model used to prove the soundness of Rocq's original extraction mechanism, where formulas in the Calculus of Constructions are realized by programs in $F_\omega$ [34]. More generally, to enable our framework to provide specifications for actual programming languages, we need to (i) distinguish between the language used for logical propositions and that used for types, and (ii) allow the language of realizers to be typed. The latter is not a restrictive requirement since considering a unique type assigned to all programs reduces to an untyped setting. To further refine the separation, we consider an *a priori* distinct language for the propositions in our logic, which we call *specifications*. To address this challenge, we thus enhance the translation to further include an assignment of a type for each proposition:

$$\text{Proposition} \xrightarrow{\text{realizability translation}} \text{Type} \times \text{Specification}$$

Technically, if $\Phi$ is a proposition of the source logic, we define both its interpretations as a type $[\![\Phi]\!]^{\mathrm{T}}$ and as a specification $[\![\Phi]\!]^{\mathrm{S}}$. Going back to our example, assuming that $\Phi_1$ and $\Phi_2$ are respectively interpreted by some types $\tau_1$ and $\tau_2$, a realizer of $\Phi = (\Phi_1 \wedge \Phi_2) \supset (\Phi_2 \wedge \Phi_1)$ is required to be a program of type $[\![\Phi]\!]^{\mathrm{T}} = \tau_1 \times \tau_2 \to \tau_2 \times \tau_1$, while $[\![\Phi]\!]^{\mathrm{S}}$ now defines a propositions on programs of that type.

### B. Polymorphism

Our candidate realizer $r$ for the proposition $\Phi$ does not rely on a particular choice for the propositions $\Phi_1$ and $\Phi_2$. At the typing level, this can be reflected using polymorphic types, in particular, in an expressive enough type-system we could type $r : \forall \tau_1, \tau_2.\tau_1 \times \tau_2 \to \tau_2 \times \tau_1$. The logical counterpart of this amounts to the universal quantification on propositions provided by second-order logic, allowing for a refinement of the proposition compatible with any choice of $\Phi_1$ and $\Phi_2$:

$$\Phi' \triangleq \forall \Phi_1, \Phi_2.\,(\Phi_1 \wedge \Phi_2) \supset (\Phi_2 \wedge \Phi_1)$$

While there exist numerous logical systems featuring such a quantification, e.g. Girard-Reynold's System F [35], it is well-known that this quantification introduces nuanced challenges when interpreting it within a model [36]. This work is no exception: since we want propositions in the source language to be interpreted both as a specification on their realizers and as the type of these realizers, it means that via a sound translation, the image of a universal quantification over any possible proposition should range over any possible specification over several possible types, while the corresponding realizers may be assigned polymorphic types. Hence, the type system for terms includes a quantification over types, while the specification language features both a quantification over types and over specifications. With these, we define the translations to types and specifications of a proposition $\forall X.\Phi(X)$ as follows

$$[\![\forall X.\Phi(X)]\!]^{\mathrm{T}} \triangleq \forall \tau : \star.\, [\![\Phi(\tau)]\!]^{\mathrm{T}}$$
$$[\![\forall X.\Phi(X)]\!]^{\mathrm{S}} \triangleq \forall \tau : \star. \forall p : \tau \to \mathsf{Prop}.\, [\![\Phi(p)]\!]^{\mathrm{S}}$$

where $\tau : \star$ (resp. $p : \mathsf{Prop}$) denotes that $\tau$ is a type (resp. $p$ a proposition). In particular, as specifications, propositional variables are interpreted as predicates on their realizer's type.

### C. Higher-Order Logic

The realizability interpretation we provide is, in fact, of stronger logical expressiveness in that it interprets higher-order logic (**HOL**), which adds a few more technicalities. In (multi-sorted) **HOL**, besides mere propositions, formulas also include predicates ranging over terms of a given *sort* $s$ (acting like sets of such terms), predicates ranging over such predicates (acting like sets of sets), etc. In the general case, **HOL** includes predicates of sorts $\mathsf{P}(s_1, \ldots, s_n)$, ranging over terms of sorts

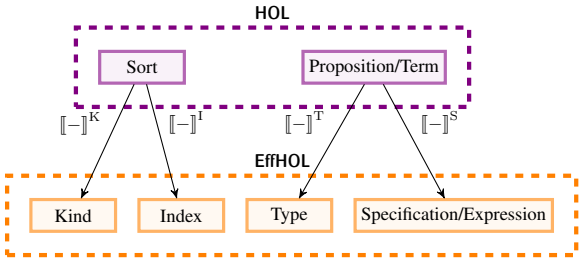

Fig. 1: The Realizability Translation

$s_1$, ... $s_n$. This makes the underlying theory expressive enough to internalize any (open) formula $\Phi(x_1, ...., x_n)$ ranging over variables $x_1, ...x_n$ respectively of sorts $s_1, ..., s_n$ through a comprehension predicate $\{x_1 : s_1, ..., x_n : s_n | \Phi(x_1, ...., x_n)\}$ of sort $\mathsf{P}(s_1, \ldots, s_n)$. The fact that this predicate holds for certain terms $t_1, ..., t_n$ can then be expressed using a membership proposition $t_1, ..., t_n \in \{x_1 : s_1, ..., x_n : s_n | \Phi(x_1, ...., x_n)\}$ that is logically equivalent to the formula $\Phi(t_1, ..., t_n)$.

The realizability interpretation requires the target language **EffHOL** to also encompass higher-order logic. Specifically, since any proposition of **HOL** is translated both as a type and as a specification, both components need to be equipped with a counterpart of the sort system: the translation to types uses a *kind system* for types while the translation to specifications uses something analogous to kinds for specifications which we call *indices*. To summarize, our target language **EffHOL** is a language of *specifications*, whose structures are reflected by *indices*, expressing properties of *terms*. In turn, these terms come with a *type*, whose structure is expressed by means of a *kind* system. The realizability interpretation, as illustrated by Fig. 1, consists of four translations, two translations $[\![\cdot]\!]^{\mathrm{K}}$ and $[\![\cdot]\!]^{\mathrm{I}}$ from sorts to kinds and indices, and two translations $[\![\cdot]\!]^{\mathrm{T}}$ and $[\![\cdot]\!]^{\mathrm{S}}$ to types and specifications respectively.

In particular, for **HOL** formulas that quantify over predicates of a given sort, e.g. $\forall X : s.\Phi(X)$, through the realizability interpretation, predicates (here the predicate variable $X : s$) are turned into specifications expressing which terms define adequate realizers. Such specifications therefore range over one extra variable for terms of the corresponding type, and as such their index should be refined to reflect this. We write $\mathsf{R}_\tau(\sigma_1, ..., \sigma_n)$ to denote the index of a specification (over specifications of indices $\sigma_1, ..., \sigma_n$) whose realizers are of type $\tau$. Hence, the translation of a sort $s$ to an index takes the corresponding type (say $\tau$) of the intended realizers as a parameter, written as $[\![s]\!]^{\mathrm{I}}_\tau$. Overall, the translations of a formula $\forall X : s.\Phi$ to types and specifications is given by:

$$[\![\forall X : s.\Phi(X)]\!]^{\mathrm{T}} \triangleq \forall \tau : [\![s]\!]^{\mathrm{K}} . [\![\Phi(\tau)]\!]^{\mathrm{T}}$$
$$[\![\forall X : s.\Phi(X)]\!]^{\mathrm{S}} \triangleq \forall \tau : [\![s]\!]^{\mathrm{K}} . \forall p : [\![s]\!]^{\mathrm{I}}_\tau . [\![\Phi(p)]\!]^{\mathrm{S}}$$

The full translation, with extra technicalities for handling open propositions and various contexts, is given in Sec. V.

### D. Spectrum

Following the modern tradition of realizability interpretations, we aim at a framework that allows for effectful realizers. In this paper, we choose to approach effects via monads, as is done in Pitts' Evaluation Logic [28] and Moggi's Computational $\lambda$-Calculus [37]. Specifically, our language features a type $M(\tau)$ denoting computations of type $\tau$, while terms account for the usual *return* and *bind* constructions of monads. This choice has the strong benefit that the resulting system is compatible with *any* choice of a monad, endorsing the subsequent language of realizers with the corresponding effects. However, this is only a choice made for the purpose of providing a ready-to-use generic framework. Indeed, we could also consider variants of **EffHOL** where terms, instead of monadic constructs, allow for effectful instructions in direct-style, as is done for instance in Krivine classical realizability with the control operator `call/cc` [3]. Nonetheless, such an approach would have the significant drawback that any choice of a particular effect would require to adapt the operational semantics (e.g. with stacks for control-operators, heaps for states, etc.), and therefore the target language **EffHOL** of our realizability translation and its semantics. While a monad-free approach to effects would be more general, ensuring soundness and usefulness for a specific instance would require significantly more effort. Thus, to provide a robust generic framework, we here focus on the monadic approach. This already allows us to cover a wide spectrum of realizability interpretations, as we shall demonstrate.

## III. Higher-Order Logic

To begin, we fix the specific version of higher-order logic employed in this paper. To generate realizability models, we focus on a constructive variant, namely, intuitionistic, many-sorted, monadic, higher-order logic, denoted by **HOL**. To keep our system as general as possible, we opt for a very minimalistic formalization of **HOL**. For one, we only consider a core of logical constructs, namely, implication and universal quantification. Furthermore, our language emulates propositional application and abstraction through comprehension terms and membership propositions. We do this instead of the alternative formalization using $\lambda$-terms to ensure we do not commit to any specific language construct that goes beyond the bare minimum required for **HOL**.

Fig. 2 presents the **HOL** framework, where all inference rules assume well-formedness. As a many-sorted logic, the syntax of **HOL** consists of *propositions* and *terms*, where each term has a *sort*. Terms are either variables or (base) comprehension terms. Propositions can make statements about properties of terms by having terms appear inside propositions. The comprehension terms provide a syntactic machinery for terms to refer to propositions, which, in turn, allows propositions to make statements about properties of other propositions. This is the core source of the higher-order structure of the logic. 'Extracting' the inner proposition from a comprehension term, is then done via the membership proposition. Concretely, for a term $t_1$ of sort $s$ and a comprehension term $t_2 = \{u : s \mid \psi\}$ of sort $s \rightarrowtail \star$ , the proposition $t_1 \in t_2$ states that $t_1$ is a member of $t_2$, which amounts to saying that $\psi$ with $t_1$ for $u$ holds . This mechanism may be seen as a representation of standard func-

$$
\begin{array}{lrcl}
\textbf{Sorts} & s & ::= & \star \mid s \rightarrowtail \star \\
\textbf{Terms} & t & ::= & u \mid \{u : s \mid \psi\} \mid \{\psi\} \\
\textbf{Propositions} & \psi & ::= & \overline{t} \mid t \in t \mid \psi \sqsupset \psi \mid \forall_{u:s}.\psi \\
\textbf{Sort context} & \mathbb{S} & ::= & \emptyset \mid \mathbb{S}, u : s
\end{array}
$$

$$
\frac{\mathbb{S} \vdash \Psi, \psi_1 \Rrightarrow \psi_2}{\mathbb{S} \vdash \Psi \Rrightarrow \psi_1 \sqsupset \psi_2}\,(\texttt{Imp-I}) \qquad \frac{\mathbb{S} \vdash \Psi \Rrightarrow \psi_1 \sqsupset \psi_2 \quad \mathbb{S} \vdash \Psi \Rrightarrow \psi_1}{\mathbb{S} \vdash \Psi \Rrightarrow \psi_2}\,(\texttt{Imp-E})
$$

$$
\frac{\mathbb{S}, u : s \vdash \Psi \Rrightarrow \psi}{\mathbb{S} \vdash \Psi \Rrightarrow \forall_{u:s}.\psi}\,(\texttt{Uni-I}) \qquad \frac{\mathbb{S} \vdash \Psi \Rrightarrow \forall_{u:s}.\psi}{\mathbb{S} \vdash \Psi \Rrightarrow \psi\,[u := t]}\,(\texttt{Uni-E})
$$

$$
\frac{\mathbb{S} \vdash \Psi \Rrightarrow \psi\,[u := t]}{\mathbb{S} \vdash \Psi \Rrightarrow t \in \{u : s \mid \psi\}}\,(\texttt{Mem-I}) \qquad \frac{\mathbb{S} \vdash \Psi \Rrightarrow t \in \{u : s \mid \psi\}}{\mathbb{S} \vdash \Psi \Rrightarrow \psi\,[u := t]}\,(\texttt{Mem-E})
$$

$$
\frac{\mathbb{S} \vdash \Psi \Rrightarrow \psi}{\mathbb{S} \vdash \Psi \Rrightarrow \overline{\{\psi\}}}\,(\texttt{Mem}_0\texttt{-I}) \qquad \frac{\mathbb{S} \vdash \Psi \Rrightarrow \overline{\{\psi\}}}{\mathbb{S} \vdash \Psi \Rrightarrow \psi}\,(\texttt{Mem}_0\texttt{-E}) \qquad \frac{\psi \in \Psi}{\mathbb{S} \vdash \Psi \Rrightarrow \psi}\,(\texttt{Id})
$$

Fig. 2: **HOL** Syntax and Theory

tion abstraction and application in programming languages, which allows higher-order functions. Comprehension binds a variable within a proposition and turns it into a comprehension term, just as abstraction binds a variable within an expression to turn it into a function. Membership applies a comprehension term to an argument, yielding a proposition, just like a function applied to an argument yields an expression.

Propositions include only implication, universal quantification, and membership, from which the other logical constructs are standardly derivable. But, while comprehension terms $\{u : s \mid \psi\}$ of composite sort $s \rightarrowtail \star$ are in principle enough to encode the other logical connectives and construct the higher-order hierarchy, our language also requires base terms and propositions. For this, we include comprehension terms $\{\psi\}$ of base sort $\star$ with corresponding embedding of terms $t : \star$ as formulas $\overline{t}$ to effectively simulate quantification over propositional variables. Thus, for instance, a falsity constant $\bot$ can be defined naturally by $\forall_{u:\star}.\overline{u}$.

We use $\mathbb{S}$ to denote a sort context, i.e., a list of unique sorted variables $u_1 : s_1, \ldots, u_n : s_n$. We have standard typing rules for the judgments that, under a sort context $\mathbb{S}$ a term is of a specific sort, written $\mathbb{S} \vdash t : s$, and that $\psi$ is a well-formed proposition, written $\mathbb{S} \vdash \psi$. We opt for a representation where well-formed propositions are a separate construct, equivalently, a unique sort could have been used to denote well-formed propositions.

For simplicity, we use a monadic presentation of **HOL**, i.e. using only unary sorts $s \rightarrowtail \star$ instead of $n$-ary sorts of the form $(s_1, \ldots s_n) \rightarrowtail \star$ . While in second-order logic, the monadic fragment is less expressive, with arbitrary orders, expressivity is equivalent since $n$-ary sorts can be encoded at higher orders [38]. Thus, our sort structure is, in fact, equivalent to the natural numbers but is presented syntactically to allow for potential extensions to more complex sort structures.

The main judgments of the theory of **HOL** are of the form $\mathbb{S} \vdash \Psi \Rrightarrow \psi$, where $\Psi$ is a finite set of propositions and $\mathbb{S}$ is an adequate sort context. A sequent represents *entailment*, that is, the judgment that within the specified context, assuming all the propositions in the left-hand-side hold entails the right-hand-side holds. The theory of **HOL** is inductively generated by the inference rules in Fig. 2. Standardly, in rule (`Uni-I`), $u$ is a fresh variable. The membership rules simply identify membership in comprehension terms $t \in \{u : s \mid \psi\}$ with substitutions $\psi\,[u := t]$ of the inner formula as expected. ($\psi\,[u := t]$ and $t'\,[u := t]$ denote standard substitutions of $t$ for $u$ in $\psi$ and $t'$, resp.) As noted, we omit the typing premises, but, for example, rule (`Mem-I`) has the additional premise $\mathbb{S} \vdash t : s$. Standard structural rules like Exchange, Weakening, Contraction, and Cut are derivable from those in Fig. 2, given that $\Psi$ is treated as a set in the (`Id`) rule.

## IV. Effectful Higher-Order Logic

To provide an abstract, general framework for handling a wide range of program languages, this section describes Effectful Higher-Order Logic, **EffHOL**. **EffHOL** is a form of higher-order logic based on two key features: higher-kinded polymorphism and computational term language. The higher-kinded polymorphic type system, inspired by Girard's System $F_\omega$ [27], is used to type higher-order specifications. The computational term language enables the treatment of effectful programs. It can be seen as a simplification of Pitts' Evaluation Logic [28] that only has a single monotonic modality with the values and composition laws holding only in the left-to-right direction. In particular, the effectful aspect of the language is captured through monads. The design of **EffHOL** invokes similar design choices as **HOL**, for example, opting for a monadic presentation and using only unary kinds and indices.

### A. The Language of EffHOL

The syntax of **EffHOL**, formally given in Fig. 3, consists of the following components: *kinds*, *types*, *programs*, *indices*, *expressions* and *specifications*. Kinds and types are used to provide the types of realizers associated with typed programs, while indices, expressions and specifications hold the logical counterpart of our realizability interpretation by describing the properties of these programs.

As is standard, *kinds* are used as types for *type constructors*. Well-formed type constructors are of kind $\kappa \rightarrowtail \star$, which denotes the kind of type constructors that take type constructors of kind $\kappa$ and return a type. Concrete types take no arguments and have kind $\star$. As for sorts in **HOL**, we take a minimalistic kind system, equivalent to the natural numbers.

The syntax of *types* is also minimal, and includes variables, application, abstraction, functions, universal, and computation types. As is standard, the type abstraction and universal bind a free variable in a type. The former returns a type constructor, while the latter returns a polymorphic type. Intuitively, the $M$ stands for the underlying monad of **EffHOL**. This approach ensures that the computational behavior of programs is made explicit in the type system, and allows for easy extensions with constructs that exhibit various computational effects. Since the type system has higher-order polymorphism it does not include base types. Each type variable has a kind. To keep track of the kinds of the free type variables in a type, all judgments in

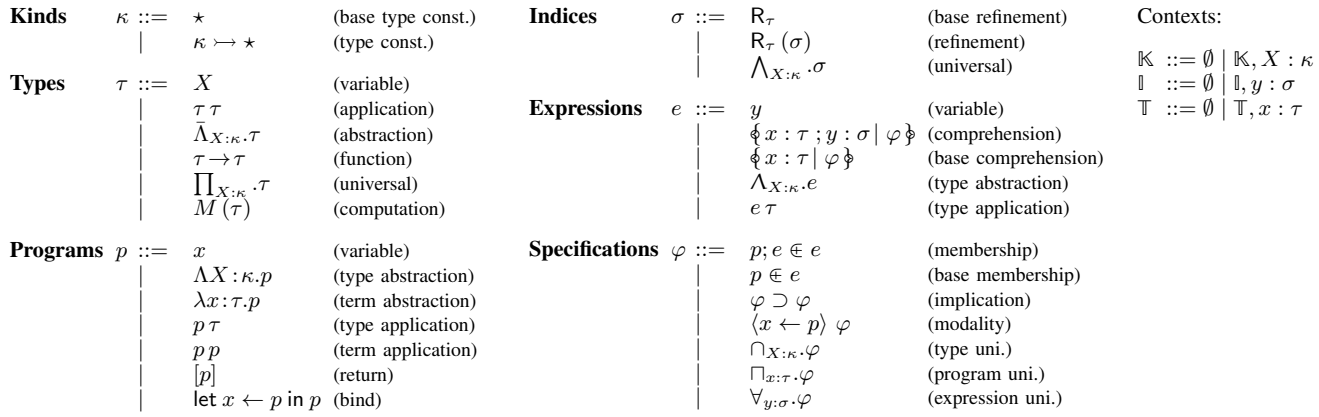


Fig. 3: **EffHOL** Syntax

**EffHOL** depend on a *kind context*, denoted by $\mathbb{K}$, containing declarations of the form $X : \kappa$.

The syntax of programs combines constructs for pure and effectful computations. It contains the standard constructs of pure programs, as in System $F_\omega$, such as variables, term abstraction and application, and type abstraction and application. In addition, it contains standard constructs for handling effectful computations through monads via return and bind, as in Pitts' Evaluation Logic [28] (or rather, Moggi's Computational $\lambda$-Calculus [37]). The variable $X$ is bound in type abstraction and the variable $x$ is bound in term abstraction and bind program. Programs depend on types, which, in turn, depend on kinds. Therefore, they further require a *type context*. A type context, denoted by $\mathbb{T}$, contains declarations of the form $x : \tau$. That is, it is a list of typed program variables, where each type may have free variables denoting type constructors, hence a type context depends on a kind context.

To reason about programs, the logical part of **EffHOL** relies on *specifications*, which in turn are defined using *expressions* which are typed with *indices*. Indices mark the logical order of statements. First-order statements about a program of type $\tau$ are indexed with the base refinement $\mathsf{R}_\tau$. Higher-order statements can also refer to some other statements of different types, in which case they are of the form $\mathsf{R}_\tau\,(\sigma)$. The syntax of indices also includes indices of the form $\bigwedge_{X:\kappa}.\sigma$, which bind $X$, that capture that the index is polymorphic over the type variable $X$ of kind $\kappa$, together with the corresponding expressions to abstract over types.

Expressions include variables, (base) comprehension, and type abstraction and application. The type abstraction (in which $X$ is bound) and type application reflect type polymorphism. To enable stating higher-order properties of programs, the comprehension expression binds a program variable along with an optional expression variable. The program variable specifies the program satisfying the property, while the optional expression variable describes a property that is related to the program in some manner via the inner specification. A comprehension expression that takes an expression argument is denoted by $\{\!|\, x : \tau\,;y : \sigma \mid \varphi \,|\!\}$, whereas, like in **HOL**, base comprehension expression with no arguments is denoted by $\{\!|\, x : \tau \mid \varphi \,|\!\}$. Each expression is typed with an index, and we use an *index context* $\mathbb{I}$, to keep track of such declarations $y : \sigma$, where $y$ is an expression variable.

The membership specification, $p; e_2 \Subset e_1$, intuitively states that $e_1$ (with argument $e_2$) holds on $p$. When $e_1$ takes no arguments, we use the base expression $p \Subset e_1$. As in **HOL**, a membership specification is a form of application in that stating that an expression is a member of a comprehension expression amounts to the inner comprehension specification applied to that expression. The language of specifications includes standard implication, and has three different forms of universal quantification: type quantification quantifies over programs, index quantification quantifies over properties thereof, and kind quantification allows the specifications to be polymorphic over arbitrary types. Last, to describe specifications of effectful programs, we use a modality $\langle x \leftarrow p \rangle\ \varphi$, which intuitively states that the property $\varphi$ holds when $x$ is the result of running the (effectful) program $p$. This is the core source of effectful computations in **EffHOL**. Specifically, if only considering pure programs, this would collapse into standard substitution.

**Example 1.** To illustrate the role of each of the logical components, consider, e.g., a specification $p \downarrow^\tau := (\langle x \leftarrow p \rangle\ \bot) \supset \bot$ stating that a program $p$ of type $\tau$ is not non-terminating. Using a comprehension term, one can build the first-order expression $\mathtt{norm}_\tau \triangleq \{\!|\, x : \tau \mid x \downarrow^\tau \,|\!\}$ which corresponds to the set of programs of type $\tau$ which are not non-terminating, or, in a more type-theoretic fashion, as a function that associates to each such program $p : \tau$ the specification $p \downarrow^\tau$. Such an expression is of index $\mathsf{R}_\tau$, while in the higher-order case, an index $\mathsf{R}_\tau\,(\sigma)$ denotes expressions that define a specification for programs of type $\tau$ using an expression of index $\sigma$.

For a program of type $\tau \to \tau$, a natural example of such a higher-order statement is to state that some property that holds for the input also holds for the output. Using comprehension, such property can be formalized using the expression $\mathtt{pres}_\tau \triangleq \{\!|\, x : \tau \to \tau\,; e : \mathsf{R}_\tau \mid \sqcap_{x':\tau}.(x' \Subset e \supset x\,x' \Subset e) \,|\!\}$ of index $\mathsf{R}_{\tau\to\tau}\,(\mathsf{R}_\tau)$. Then, $(\lambda x{:}\tau.x)\,;\mathtt{norm}_\tau \Subset \mathtt{pres}_\tau$ ex-

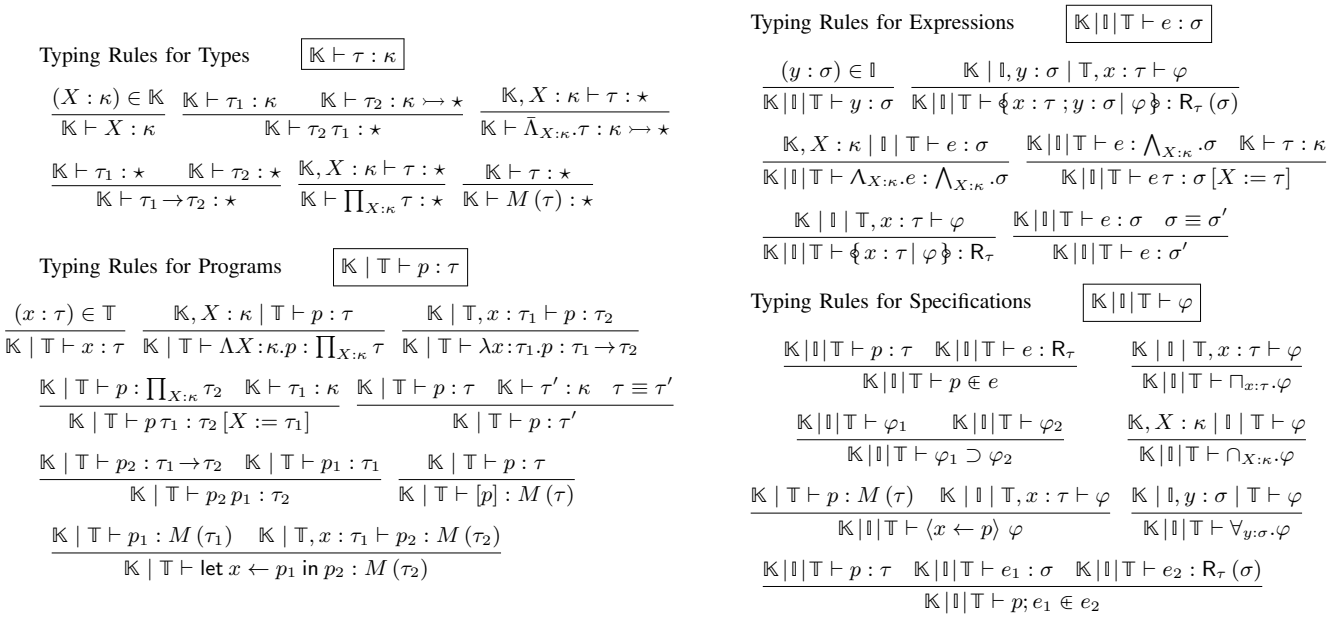


Fig. 4: Typing Rules for **EffHOL**

presses that the identity function preserves normalization.

**EffHOL** supports the following standardly defined capture-avoiding substitutions: $A\,[X := \tau]\,, B\,[x := p]\,, C\,[y := e]$ for $A \in \{\text{`}\tau\text{'}, \text{`}\sigma\text{'}, \text{`}p\text{'}, \text{`}e\text{'}, \text{`}\varphi\text{'}\}, B \in \{\text{`}p\text{'}, \text{`}e\text{'}, \text{`}\varphi\text{'}\}$, and $C \in \{\text{`}e\text{'}, \text{`}\varphi\text{'}\}$.

To summarize, the computational components of **EffHOL** are kinds, types and programs. Each type has a kind and each program has a type. The logical components of **EffHOL** are indices, expressions and specifications. Each expression has an index. Specifications and expressions depend on both programs and indices, which, in turn, both depend on types, which, in turn, depend on kinds.

### B. Operational Semantics and Type System

Since **EffHOL** is agnostic to the details of the computational effects, the operational semantics also has to be as generic and modular as possible, allowing for different computational effects to be plugged in. Therefore, we only invoke a minimal (one-step) $\beta$-reduction relation $\rightsquigarrow$ on programs:

$$\mathsf{let}\ x \leftarrow [p_1]\ \mathsf{in}\ p_2 \rightsquigarrow p_2\,[x := p_1]$$

$$(\Lambda X{:}\kappa.p)\,\tau \rightsquigarrow p\,[X := \tau] \qquad (\lambda x{:}\tau.p)\,\mathcal{V} \rightsquigarrow p\,[x := \mathcal{V}]$$

where $\mathcal{V} ::= x \mid \Lambda X{:}\kappa.p \mid \lambda x{:}\tau.p$. An abstraction applied to a type is reduced to type substitution (in types and expressions). We define a conversion relation $\equiv$ as the (reflexive, symmetric and transitive) contextual closure of $\equiv_\tau \cup \equiv_e$ for

$$(\bar{\Lambda}_{X:\kappa}.\tau_1)\,\tau_2 \equiv_\tau \tau_1\,[X := \tau_2] \qquad (\Lambda_{X:\kappa}.e)\,\tau \equiv_e e\,[X := \tau]$$

in all syntactic categories of the equivalence relation induced by these reductions.

The typing rules of **EffHOL** use four different judgments: $\mathbb{K} \vdash \tau : \kappa$ for $\tau$ being a well-formed type of kind $\kappa$ in the context, $\mathbb{K} \mid \mathbb{T} \vdash p : \tau$ for $p$ being a well-formed program of type $\tau$, $\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash e : \sigma$ for $e$ being a well-formed expression of index $\sigma$, and $\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \varphi$ for $\varphi$ being a well-formed specification. The type system of **EffHOL** is inductively defined by the rules in Fig. 4. For readability, we elide the typing for $\sigma$ being a well-formed index, as those are easily obtained by requiring the constituent types to be of base kind (for full details, see the Coq formalization). The typing rules are closed under context extension and under (well-formed) substitution. Moreover, they are closed under conversion and $\beta$-reduction, and type preservation for programs holds.

### C. Deductive Apparatus

The theory of **EffHOL** is obtained through inference rules that manipulate sequents of specifications. Judgments are of the form $\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi$, where $\Phi$ is a finite set of specifications. Fig. 5 presents the inference rules that inductively define the theory of **EffHOL** (omitting the typing premises). The rules closely resemble those of **HOL**, except for those handling modalities and polymorphism. Standardly, the theory includes an identity axiom, implication introduction and elimination, and universal introduction and elimination for expressions. Additionally, it includes introduction and elimination rules for program and type universal quantification. As is standard, in (`UniProg.-I`) and (`UniExp.-I`), $x$ and $y$, resp., are fresh.

In addition, **EffHOL** includes rules for modalities. The modality introduction rule, (`Mod-I`), states that whenever a specification holds for a value, it holds for the computation that does nothing except return that value. The modality elimination rule, (`Mod-E`), states that a nesting of two modalities, where the specification depends only on the last computation, can be collapsed into a single modality with the same specification over the nesting of the computations. The Monotonicity rule, (`Mon`), states that the modality respects deductions, so

$$\frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T}, x : \tau \vdash \Phi \Rightarrow \varphi}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \sqcap_{x:\tau}.\varphi}\text{(UniProg.-I)} \qquad \frac{\mathbb{K} \mid \mathbb{I}, y : \sigma \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \forall_{y:\sigma}.\varphi}\text{(UniExp.-I)} \qquad \frac{\mathbb{K}, X : \kappa \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \cap_{X:\kappa}.\varphi}\text{(UniType-I)}$$

$$\frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \sqcap_{x:\tau}.\varphi}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi\,[x := p]}\text{(UniProg.-E)} \qquad \frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \forall_{y:\sigma}.\varphi}{\mathbb{K} \mid \mathbb{T} \mid \mathbb{I} \vdash \Phi \Rightarrow \varphi\,[y := e]}\text{(UniExp.-E)} \qquad \frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \cap_{X:\kappa}.\varphi}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi\,[X := \tau]}\text{(UniType-E)}$$

$$\frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi, \varphi_1 \Rightarrow \varphi_2}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi_1 \supset \varphi_2}\text{(Imp-I)} \qquad \frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi\,[x := p]}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \langle x \leftarrow [p] \rangle\ \varphi}\text{(Mod-I)} \qquad \frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T}, x : \tau \vdash \Phi, \varphi_1 \Rightarrow \varphi_2 \quad \mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \langle x \leftarrow p \rangle\ \varphi_1}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \langle x \leftarrow p \rangle\ \varphi_2}\text{(Mon)}$$

$$\frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi_1 \supset \varphi_2 \quad \mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi_1}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi_2}\text{(Imp-E)} \qquad \frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \langle x_1 \leftarrow p_1 \rangle\ \langle x_2 \leftarrow p_2 \rangle\ \varphi}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \langle x_2 \leftarrow (\mathsf{let}\ x_1 \leftarrow p_1\ \mathsf{in}\ p_2) \rangle\ \varphi}\text{(Mod-E)} \qquad \frac{\varphi \in \Phi}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi}\text{(Id)}$$

$$\frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi\,[x := p]\,[y := e]}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \ \Rightarrow\ p; e \Subset \{\!|\, x : \tau\,; y : \sigma \mid \varphi \,|\!\}}\text{(Mem-I)} \qquad \frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi\,[x := p]}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \ \Rightarrow\ p \Subset \{\!|\, x : \tau \mid \varphi \,|\!\}}\text{(Mem}_0\text{-I)} \qquad \frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi' \quad \varphi \equiv \varphi'}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi}(\equiv)$$

$$\frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \ \Rightarrow\ p; e \Subset \{\!|\, x : \tau\,; y : \sigma \mid \varphi \,|\!\}}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi\,[x := p]\,[y := e]}\text{(Mem-E)} \qquad \frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \ \Rightarrow\ p \Subset \{\!|\, x : \sigma \mid \varphi \,|\!\}}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi\,[x := p]}\text{(Mem}_0\text{-E)} \qquad \frac{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi\,[x := p_2] \quad p_1 \rightsquigarrow p_2}{\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi\,[x := p_1]}(\rightsquigarrow)$$

Fig. 5: The Theory of **EffHOL**

whenever $\varphi_1$ entails $\varphi_2$, if $\varphi_1$ holds for the value of some computation, so does $\varphi_2$. The (base) membership introduction and elimination rules are completely dual and acount for the mutual dependency of expressions and specifications.

Lastly, **EffHOL** includes two computational rules that enable the use of the computational reductions and conversions in the logical side of **EffHOL**. The rule $(\equiv)$ allows for replacing specifications that are equivalent by conversions. The rule $(\rightsquigarrow)$ enforces a key property of the realizability interpretation, namely the closure of specifications under anti-reduction, by stating that if a program $p_1$ reduces to a program $p_2$, then any valid specification for $p_2$ is also valid for $p_1$. By the assumption that all judgments are well-typed, rule $(\rightsquigarrow)$ can only be applied when the reduction preserves typing. Standard structural rules are admissible in **EffHOL**, with the exception that the substitution rule for programs is only valid for values.

Since **HOL** is a mere fragment of **EffHOL**, there is a trivial forgetful translation function $[\![-]\!]$ from the latter to the former, erasing all program, type, and kind structure. Thus, all **EffHOL** deductions can be replayed in **HOL**. In particular, as any specification $\varphi$ expressing a contradiction in **EffHOL** is mapped to a formula $[\![\varphi]\!]$ expressing a contradiction in **HOL**. This reduces the consistency of **EffHOL** to that of **HOL**.

**Proposition 2 ().** **EffHOL** *is consistent.*

*Hoare Triples:* The theory of **EffHOL** allows expressing properties of programs in the familiar style of Hoare logic [39]. The known *triple form* is defined as an abbreviation of a sequent with a modality on the right-hand-side:

$$\{\Phi\}\, x \longleftarrow p\, \{\varphi\} \quad \triangleq \quad \Phi \Rightarrow \langle x \leftarrow p \rangle\ \varphi$$

Intuitively, this states that, assuming all the formulas in $\Phi$ hold, then after binding $x$ with the result obtained by computing $p$, the formula $\varphi$ holds. The triple form will be used later when discussing the realizability translation. That is, we show that for every provable theorem of **HOL** there is a corresponding program, called the *evidence* of the theorem, for which the translation of said theorem forms a provable *triple* in **EffHOL**.

Importantly, the consistency of Prop. 2 only concerns the core logical system of **EffHOL** and not the derived triples. That is, while $\bot$ is underivable, $\langle x \leftarrow p \rangle\ \bot$ may be derivable for some $p$s. Indeed, we allow for instantiations of the computational system, including ones realizing no meaningful logic.

### D. Instances of EffHOL

**EffHOL** can be seen as a generic framework relying on several parameters, that can be instantiated in different ways to capture a wide range of computational behaviors. The most direct family of instantiations is given by syntactic translations of **EffHOL** into itself, using its effect-free fragment **EffHOL**$^-$ in the target, i.e., the subsystem of **EffHOL** without the constructs $M\,(\tau)$, $[p]$, $\mathsf{let}\ x \leftarrow p_1\ \mathsf{in}\ p_2$, and $\langle x \leftarrow p \rangle\ \varphi$. In the target language, we allow any choice of a reduction relation $\rightsquigarrow$ on programs that extends the $\beta$-reduction of **EffHOL**. That is, different operational semantics for programs can be invoked by, e.g., extending the notion of values or adding congruence rules describing a particular evaluation strategy. Sec. VI-A provides an example that invokes a call-by-name evaluation strategy. As standard, meta-properties of **EffHOL**, such as type preservation for programs, might fail in such extensions.

**Definition 3** (Pure instance ). **A pure instance of EffHOL** is an interpretation of **EffHOL** in **EffHOL**$^-$ which interprets the non-pure constructs in **EffHOL** as pure. That is, a pure instance of **EffHOL**: (1) assigns to each $\tau, p, p_1, p_2, x, \varphi$, an interpretation in **EffHOL**$^-$ of $M\,(\tau)$ as a type, $[p]$ as a program, $\mathsf{let}\ x \leftarrow p_1\ \mathsf{in}\ p_2$ as a program, and $\langle x \leftarrow p \rangle\ \varphi$ as a specification, such that the typing and inference rules of $M\,(\tau)$, $[p]$, $\mathsf{let}\ x \leftarrow p_1\ \mathsf{in}\ p_2$, and $\langle x \leftarrow p \rangle\ \varphi$ are satisfied by their respective interpretations, and (2) picks a (potentially extended) evaluation strategy, such that the reduction and conversion rules are preserved.

**Lemma 4 ().** *If* $\mathbb{K} \mid \mathbb{I} \mid \mathbb{T} \vdash \Phi \Rightarrow \varphi$ *is derivable in* **EffHOL**, *then for every pure instance, the interpreted judgment is derivable in* **EffHOL**$^-$.

After introducing the realizability interpretation (Thm. 5), Sec. VI provides illustrative examples of pure instances and showcases how the effectful programs provided by the monad can be used to realize additional logical principles. In the context of Thm. 5 we will also discuss a way in which the computational behavior can be axiomatized within **EffHOL** itself, without an explicit syntactic instantiation.

Interestingly, the above restricted notion of pure instances is already sufficient for capturing all those examples. Nonetheless, one might invoke a more general notion of an instance, allowing for different target languages outside of **EffHOL** itself. To define such an instance, one has to choose a monad and an evaluation strategy for programs, and then to define the interpretation of the modality $\langle x \leftarrow p \rangle\ \varphi$. Any choice for these parameters can be taken, provided that substitution, reduction and conversion are respected, and the corresponding typing judgments and logical rules remain valid.

## V. The Realizability Translation

This section established how the **EffHOL** framework can be used to model **HOL**. To this end, we provide a syntactic realizability translation of **HOL** judgments into **EffHOL** judgments, which ensures that the programming language described by **EffHOL** is a realizability model of **HOL**. As mentioned in the overview, the general translation mainly consists of the following four syntactic translations.

$$\llbracket - \rrbracket^{\mathrm{K}} \ : \ \mathtt{sort} \to \mathtt{kind} \qquad \llbracket - \rrbracket^{\mathrm{I}}_{-} \ : \ \mathtt{sort} \to \mathtt{type} \to \mathtt{index}$$
$$\llbracket - \rrbracket^{\mathrm{T}} \ : \ \mathtt{prop} \to \mathtt{type} \qquad \llbracket - \rrbracket^{\mathrm{S}}_{-} \ : \ \mathtt{prop} \to \mathtt{prog} \to \mathtt{spec}$$

To accommodate the term language with comprehension terms, there are two additional (canonical) translations:

$$\llbracket - \rrbracket^{\mathrm{t}} \ : \ \mathtt{term} \to \mathtt{type} \qquad \llbracket - \rrbracket^{\mathrm{e}} \ : \ \mathtt{term} \to \mathtt{expr}$$

The recursive translations are given in Fig. 6. The proposition-to-specification translation, $\llbracket \psi \rrbracket^{\mathrm{S}}_{p}$, is the main realizability translation, converting any **HOL** proposition $\psi$ to a specification describing what it means for a program to realize the proposition. The realizer is internalized as an extra input $p$ of the appropriate type $\llbracket \psi \rrbracket^{\mathrm{T}}$ of realizers. The sort-to-index translation, $\llbracket s \rrbracket^{\mathrm{I}}_{\tau}$ reflects the proposition-to-specification translation at the sort level, and the sort-to-kind translation, $\llbracket s \rrbracket^{\mathrm{K}}$, reflects the proposition-to-types translation at the sort level. In fact, as the types of sorts and kinds are isomorphic, the last translation is essentially the identity mapping.

The translations of terms are required to translate variables. Since we translate higher-order propositions to higher-order specifications, a variable $u$ that appears in a proposition gets translated into a type variable, $X_u$, that fills the same role variables serve in higher-order logic, namely, to fill in for arbitrary specifications. Similarly, when translating $u$ to an expression we use an expression variable, $y_u$.

The translation of implication states that a realizer of implication takes a realizer of the assumption and computes a realizer of the conclusion, and the type of the realizer reflects this fact as a type of functions from the type of realizers of the assumption to the type of computations of realizers of the conclusion. The translation of universal quantification says that a realizer of a universally quantified proposition computes a realizer of the proposition for any possible instantiation of the quantified variable, reinterpreted as an assertion variable ranging over type variables of the appropriate kind. The translation of membership is simply membership in **EffHOL**, with the realizer bundled with the element translation and the predicate translation instantiated to the element's type.

We lift the sort translations $\llbracket - \rrbracket^{\mathrm{K}}$ and $\llbracket - \rrbracket^{\mathrm{I}}_{-}$ to contexts:

$$\llbracket \emptyset \rrbracket^{\mathrm{K}} := \emptyset \qquad \llbracket \mathbb{S}, u : s \rrbracket^{\mathrm{K}} := \llbracket \mathbb{S} \rrbracket^{\mathrm{K}}, X_u : \llbracket s \rrbracket^{\mathrm{K}}$$
$$\llbracket \emptyset \rrbracket^{\mathrm{I}} := \emptyset \qquad \llbracket \mathbb{S}, u : s \rrbracket^{\mathrm{I}} := \llbracket \mathbb{S} \rrbracket^{\mathrm{I}}, y_u : \llbracket s \rrbracket^{\mathrm{I}}_{X_u}$$

This allows us to prove that our translation preserves well-formedness judgments. With this we next state and prove the soundness of our translation. That is, we show that for every sequent $\Psi \Rrightarrow \psi$ in **HOL**, our translation produces a computation program $p : M \llbracket \psi \rrbracket^{\mathrm{T}}$, such that the triple $\left\{ \llbracket \Psi \rrbracket^{\mathrm{S}} \right\} x_r \longleftarrow p \left\{ \llbracket \psi \rrbracket^{\mathrm{S}}_{x_r} \right\}$ is provable in **EffHOL**.

**Theorem 5** (Soundness 🦫)**.** *For each* **HOL** *theorem*

$$\mathbb{S} \vdash \psi_1, \ldots, \psi_n \Rrightarrow \psi$$

*there is a program $p$ with*

$$\llbracket \mathbb{S} \rrbracket^{\mathrm{K}} \mid \llbracket \Psi \rrbracket^{\mathrm{T}} \vdash p : M \llbracket \psi \rrbracket^{\mathrm{T}}$$

*such that for any collection of specifications $\mathbb{K} \,|\, \mathbb{I} \,|\, \mathbb{T} \vdash \Phi$:*

$$\llbracket \mathbb{S} \rrbracket^{\mathrm{K}}, \mathbb{K} \mid \llbracket \mathbb{S} \rrbracket^{\mathrm{I}}, \mathbb{I} \mid \llbracket \Psi \rrbracket^{\mathrm{T}}, \mathbb{T} \vdash \left\{ \llbracket \Psi \rrbracket^{\mathrm{S}}, \Phi \right\} x_r \longleftarrow p \left\{ \llbracket \psi \rrbracket^{\mathrm{S}}_{x_r} \right\}.$$

*where we use $\Psi := \psi_1, ..., \psi_n$, $\llbracket \Psi \rrbracket^{\mathrm{S}} := \llbracket \psi_1 \rrbracket^{\mathrm{S}}_{x_1}, \ldots, \llbracket \psi_n \rrbracket^{\mathrm{S}}_{x_n}$, and $\llbracket \Psi \rrbracket^{\mathrm{T}} := x_1 : \llbracket \psi_1 \rrbracket^{\mathrm{T}}, \ldots, x_n : \llbracket \psi_n \rrbracket^{\mathrm{T}}$.*

In this setting, we can then view the translation as defining a realizability model for **HOL** by saying that a theorem of **HOL** is valid whenever there exists a program $p$ satisfying the statements of the soundness theorem. Importantly, the proof is constructive, in the sense that it actually constructs the realizer $p$ from the derivation in **HOL**. In fact, the proof shows how each rule in **HOL** corresponds to a program construct that is sound for the realizability translation, i.e. which builds a realizer of the conclusion from realizers of the premises. As a consequence, the programs extracted from the soundness proof are modular with respect to **HOL** derivations. In particular, if a specific instance of **EffHOL** allows us to provide an effectful realizer for some proposition of **HOL** through the translation, then this proposition could be taken as an extra axiom in **HOL**. Indeed, since it has a realizer this proposition would be valid in the induced realizability model, as would any theorem derived using this theorem, by modularity of the extracted realizer. We shall illustrate this in Sec. VI-A, using an instance of **EffHOL** corresponding to Krivine realizability, relying on control operators to validate classical reasoning principles.

Importantly, the program $p$ obtained in the translation is uniform in that it does not rely on the specifics of the monad and the modality. Hence, in particular, it must also work for

$$
\begin{array}{l|l|l}
[\![-]\!]^{\mathbf{K}} : \mathtt{sort} \to \mathtt{kind} & [\![-]\!]^{\mathbf{t}} : \mathtt{term} \to \mathtt{type} & [\![-]\!]^{\mathbf{e}} : \mathtt{term} \to \mathtt{expr} \\
[\![\star]\!]^{\mathrm{K}} \triangleq \star & [\![u]\!]^{\mathrm{t}} \triangleq X_u & [\![u]\!]^{\mathrm{e}} \triangleq y_u \\
[\![s \rightarrowtail \star]\!]^{\mathrm{K}} \triangleq [\![s]\!]^{\mathrm{K}} \rightarrowtail \star & [\![\{u : s \mid \psi\}]\!]^{\mathrm{t}} \triangleq \bar{\Lambda}_{X_u : [\![s]\!]^{\mathrm{K}}} . [\![\psi]\!]^{\mathrm{T}} & [\![\{u : s \mid \psi\}]\!]^{\mathrm{e}} \triangleq \Lambda_{X_u : [\![s]\!]^{\mathrm{K}}} . \{\!\{ x : [\![\psi]\!]^{\mathrm{T}} \; ; y_u : [\![s]\!]^{\mathrm{I}}_{X_u} \mid [\![\psi]\!]^{\mathrm{S}}_x \}\!\} \\
 & [\![\{\psi\}]\!]^{\mathrm{t}} \triangleq [\![\psi]\!]^{\mathrm{T}} & [\![\{\psi\}]\!]^{\mathrm{e}} \triangleq \{\!\{ x : [\![\psi]\!]^{\mathrm{T}} \mid [\![\psi]\!]^{\mathrm{S}}_x \}\!\}
\end{array}
$$

$$
\begin{array}{l|l|l}
[\![-]\!]^{\mathbf{I}}_{-} : \mathtt{sort} \to \mathtt{type} \to \mathtt{index} & [\![-]\!]^{\mathbf{T}} : \mathtt{prop} \to \mathtt{type} & [\![-]\!]^{\mathbf{S}}_{-} : \mathtt{prop} \to \mathtt{prog} \to \mathtt{spec} \\
[\![\star]\!]^{\mathrm{I}}_{\tau} \triangleq \mathsf{R}_{\tau} & [\![\psi_1 \sqsupset \psi_2]\!]^{\mathrm{T}} \triangleq [\![\psi_1]\!]^{\mathrm{T}} \to M\left([\![\psi_2]\!]^{\mathrm{T}}\right) & [\![\psi_1 \sqsupset \psi_2]\!]^{\mathrm{S}}_p \triangleq \sqcap_{x_1 : [\![\psi_1]\!]^{\mathrm{T}}} . [\![\psi_1]\!]^{\mathrm{S}}_{x_1} \supset \langle x_2 \leftarrow p\, x_1 \rangle\, [\![\psi_2]\!]^{\mathrm{S}}_{x_2} \\
[\![s \rightarrowtail \star]\!]^{\mathrm{I}}_{\tau} \triangleq \bigwedge_{X_0 : [\![s]\!]^{\mathrm{K}}} . \mathsf{R}_{\tau}\, X_0 \left([\![s]\!]^{\mathrm{I}}_{X_0}\right) & [\![\forall_{u:s} \psi]\!]^{\mathrm{T}} \triangleq \prod_{X_u : [\![s]\!]^{\mathrm{K}}} . M\left([\![\psi]\!]^{\mathrm{T}}\right) & [\![\forall_{u:s} \psi]\!]^{\mathrm{S}}_p \triangleq \sqcap_{X_u : [\![s]\!]^{\mathrm{K}}} . \forall_{y_u : [\![s]\!]^{\mathrm{I}}_{X_u}} . \langle x_0 \leftarrow p\, X_u \rangle\, [\![\psi]\!]^{\mathrm{S}}_{x_0} \\
 & [\![t_1 \in t_2]\!]^{\mathrm{T}} \triangleq [\![t_2]\!]^{\mathrm{t}}\, [\![t_1]\!]^{\mathrm{t}} & [\![t_1 \in t_2]\!]^{\mathrm{S}}_p \triangleq p ; [\![t_1]\!]^{\mathrm{e}} \Subset [\![t_2]\!]^{\mathrm{e}}\, [\![t_1]\!]^{\mathrm{t}} \\
 & [\![\overline{t}]\!]^{\mathrm{T}} \triangleq [\![t]\!]^{\mathrm{t}} & [\![\overline{t}]\!]^{\mathrm{S}}_p \triangleq p \Subset [\![t]\!]^{\mathrm{e}}
\end{array}
$$

Fig. 6: The Realizability Translation

consistent modalities that do not permit evidence of falsity, i.e., where $p : M(\tau) \vdash \langle x \leftarrow p \rangle \perp \Rightarrow \perp$ is provable. Therefore, if the modality itself is consistent, the only way to give evidence for $\top \Rightarrow \perp$ is if it holds in the meta-theory, thus ensuring the object theory is consistent as long as the meta-theory is.

By the general formulation of the soundness theorem, the realizability translation of **HOL** sequents is valid in the presence of arbitrary well-typed assumptions $\Phi$. This in itself is not surprising, due to weakening, but it offers additional applicativity in that the additional set of assumptions can be used, for example, to introduce new constants in computation types with axiomatized behavior. This generality enables treating the translation of **HOL** as a generic realizability translation into particular computational settings, enabling consistency and independence proofs. This can be done via, e.g., pure instances of **EffHOL**, as the instance interpretation can be composed with the realizability translation to obtain a sound realizability translation into the pure instance of **EffHOL** ().

## VI. Illustrative Applications

This section illustrates the versatility of the **EffHOL** framework through various applications. First, we highlight its uniformity by seamlessly reproducing classical realizability results. Next, we demonstrate its use in a simple (relative) consistency proof of Markov's Principle. Finally, we showcase its generality, enabling robust computational interpretations across diverse effectful instances. Due to space limitations, we focus on the first application and briefly outline the other two.

### A. Krivine Classical Realizability

This section illustrates how Krivine classical realizability can be seen as a particular pure instance of **EffHOL**. Krivine realizability was introduced as a complete reformulation of standard intuitionistic realizability, which was inherently incompatible with classical logic, by building on Griffin's seminal observation that the control operator `call/cc` can be typed with Peirce's law [3, 25]. Its original presentation uses control operators in a direct-style fashion, based on the $\lambda_c$-calculus, an extension of the $\lambda$-calculus with `call/cc`. To that end, the corresponding operational semantics is then expressed using processes $\langle p \parallel \pi \rangle$ of an abstract machine, where $p$ is a program and $\pi$ is a stack. The instruction `call/cc` allows programs to backtrack, as shown by its operational semantics:

$$\langle \mathtt{call/cc} \parallel p \cdot \pi \rangle \triangleright \langle p \parallel \mathtt{throw}_{\pi} \cdot \pi \rangle \quad \langle \mathtt{throw}_{\pi} \parallel p \cdot \pi' \rangle \triangleright \langle p \parallel \pi \rangle$$

When applied to a stack $\pi$, `call/cc` provides its first argument with a program $\mathtt{throw}_{\pi}$, which, at any future point, can drop the current stack to restore $\pi$. Such instructions can be compiled to the pure $\lambda$-calculus using a continuation-passing style (CPS) translation. At the type level, this translation corresponds to a negative translation embedding classical logic into intuitionistic logic [25]. Oliva and Streicher later demonstrated that Krivine realizability can be obtained by combining a standard intuitionistic realizability interpretation with a CPS translation [40]. More generally, this suggests that effectful interpretations can be defined via an indirect presentation of effects via a well-chosen monad, rather than relying on a direct-style representation. To illustrate this, we show how Oliva and Streicher's formulation of Krivine realizability can be naturally expressed in our framework using the continuation monad. We here only sketch certain salient features of Krivine realizability, and for more details, we refer the readers to papers focusing on Oliva and Streicher's approach [40]–[42].

We define **EffHOL**$^{\neg\neg}$ as a pure instance of **EffHOL** based on the continuation monad. Namely, we first define the monad $M(\tau) \triangleq \neg\neg\tau$ at the level of types where $\neg\tau$ denotes the type $\tau \to \perp$ (using the standard encoding $\perp \triangleq \prod_{X:\star} . X$). Intuitively, one can think of a computation of type $\neg\neg\tau$ as the CPS translation of a $\lambda_c$-term, therefore waiting for a continuation of type $\neg\tau$ which accounts for the translation of a stack. As such, via the continuation monad a process $\langle p \parallel \pi \rangle$ formed by the interaction of a computation $p$ and a continuation $\pi$ simply corresponds to the application $p\,\pi$.

We assume programs are evaluated in a call-by-name fashion[1], taking advantage of the fact that **EffHOL** is parametric with respect to the choice of an evaluation strategy. We define the return and bind of the monad as expected by:

$$[p] \triangleq \lambda k : \neg\tau . k\, p \quad \mathsf{let}\; x \leftarrow p_1 \,\mathsf{in}\, p_2 \triangleq \lambda k : \neg\tau_2 . p_1\, (\lambda x : \tau_1 . p_2\, k)$$

[1]This will have the benefit of easing some definitions and proofs that rely on $\beta$-reductions or substitutions, which are easier to handle in call-by-name.

where $p$ (resp. $p_1$, $p_2$) is of type $\tau$ (resp. $\neg\neg\tau_1$, $\neg\neg\tau_2$). It is then easy to verify that they satisfy the associated typing rules.

Lastly, we need to instantiate the modality $\langle p \leftarrow x\rangle\ \varphi$. For this, we follow the intuitions of Krivine's classical realizability, where realizers are defined using an orthogonality relation with respect to a set of stacks acting as opponents [3]. In a call-by-value setting, the latter is itself defined by orthogonality to a set of values acting as realizers [43]. The orthogonality relation is typically parameterized by a set of processes $\perp\!\!\!\perp$, which intuitively encompasses the set of intended correct computations in the chosen realizability model. Then, the orthogonal of a set of stacks $A$ is defined as the sets of programs that successfully compute in front of any stack in $A$, i.e. $A^\perp = \{p \mid \forall \pi \in A. \langle p \parallel \pi\rangle \in \perp\!\!\!\perp\}$. The realizability translation of an **HOL** proposition $\psi$ can be seen as the expression $\{\!|\, x : [\![\psi]\!]^{\mathrm{T}} \mid [\![\psi]\!]^{\mathrm{S}}_x \,|\!\}$ of index $\mathsf{R}_{[\![\psi]\!]^{\mathrm{T}}}$. We thus define an orthogonality relation on expressions $e$ of index $\mathsf{R}_\tau$

$$e^\perp \triangleq \{\!|\, x : \neg\tau \mid \sqcap_{x' : \tau}. (x' \Vdash e) \supset (x\, x' \Vdash \perp\!\!\!\perp) \,|\!\}$$

where, for simplicity, we take $\perp\!\!\!\perp \triangleq \{\!|\, x : \bot \mid \bot \,|\!\}$. Observe that if $e$ is an expression of index $\mathsf{R}_\tau$, then $e^\perp$ is of index $\mathsf{R}_{\neg\tau}$, and therefore $e^{\perp\perp}$ has index $\mathsf{R}_{\neg\neg\tau}$. The biorthogonality relation thus allows us to lift expressions on a given type $\tau$ to expressions on the corresponding computational type $\neg\neg\tau$. Viewing a well-formed specification $\varphi$ for $x : \tau$ as a proposition defining a set of values, for a program $p$ of type $\neg\neg\tau$, the modality $\langle x \leftarrow p\rangle\ \varphi$ can be viewed as expressing that $p$ is a valid computation with respect to $\varphi$, i.e. a realizer. Following Krivine realizability, we can define it by biorthogonality:

$$\langle x \leftarrow p\rangle\ \varphi \triangleq p \Vdash \{\!|\, x : \tau \mid \varphi \,|\!\}^{\perp\perp}$$

**Proposition 6.** **EffHOL**$^{\neg\neg}$ *is a valid pure instance of* **EffHOL**.

Since **EffHOL**$^{\neg\neg}$ is a valid pure instance of **EffHOL**, we can take advantage of the continuation monad to obtain effectful realizers for classical reasoning principles. Recall that Peirce's law is defined by $\mathrm{Peirce} \triangleq \forall_{a:\star}.\forall_{b:\star}.((a \supset b) \supset a) \supset a$, abusing notations to write $a$ for the proposition $\overline{a}$. Through the translations, we get:

$$[\![\mathrm{Peirce}]\!]^{\mathrm{T}} = \bar{\Lambda}_{X:\star}.\neg\neg\big(\bar{\Lambda}_{Y:\star}.\neg\neg((X \to \neg\neg Y) \to \neg\neg X) \to \neg\neg X\big)$$

To obtain a realizer of Pierce's law via the realizability interpretation of **HOL** in **EffHOL**$^{\neg\neg}$, we define the programs:

$$\begin{aligned}
\mathtt{call/cc} &\triangleq \Lambda X:\star.\ \Big[\Lambda Y:\star.\ \Big[\mathtt{call/cc}^{X,Y}\Big]\Big]\\
\mathtt{call/cc}^{\tau,\tau'} &\triangleq \lambda z : (\tau \to \neg\neg\tau') \to \neg\neg\tau.\lambda k : \neg\tau.\, z\, \mathtt{throw}_k^{\tau,\tau'}\, k\\
\mathtt{throw}_k^{\tau,\tau'} &\triangleq \lambda x : \tau.\lambda k' : \neg\tau'.\, k\, x
\end{aligned}$$

Observe that $\mathtt{call/cc}^{\tau,\tau'}$ and $\mathtt{throw}_k^{\tau,\tau'}$ are essentially the CPS translations of the corresponding instructions in Krivine's $\lambda_c$-calculus presented earlier (in particular, they have the same computational behavior up to the CPS), while `call/cc` handles the polymorphic aspect. Thus, it is no surprise that `call/cc` serves as a realizer for Peirce's law.

**Theorem 7.** *The following hold:*
*1)* $\vdash \mathtt{call/cc} : [\![\mathrm{Peirce}]\!]^{\mathrm{T}}$
*2)* $\vdash \top \Rightarrow \langle x \leftarrow [\mathtt{call/cc}]\rangle\ [\![\mathrm{Peirce}]\!]^{\mathrm{S}}_x$

### B. Consistency of Markov's Principle

In Sec. VI-A, since we defined a realizability model that validates Peirce's law, we obtain that **HOL** is consistent with it and all its fragments, for instance Markov's principle (MP) allowing double negation elimination of $\Sigma_1$ formulas. In fact, using partiality as a computational effect in **EffHOL**, this (relative) consistency result can be obtained in a more direct way. Note that it is well-known that Kreisel's modified realizability provides a model for the negation of Markov's Principle [44], so the principle is in fact independent.

Arguing informally, we stipulate a type variable $\mathbb{N}$ of base kind together with constants $O : \mathbb{N}$, $S : \mathbb{N} \to \mathbb{N}$ and $\mathsf{find} : (\mathbb{N} \to \mathbb{N}) \to M(\mathbb{N})$. To specify their intended behavior, we assume a set $\Phi$ expressing the usual axioms of (higher-order) arithmetic and including a specification of the form

$$\{\exists n : \mathbb{N}.\, f\, n = O\}\, x \longleftarrow \mathsf{find}\, f\, \{f\, x = O\}$$

expressing that find implements some sort of linear search. These few ingredients now allow one to quickly explore which consequences the presence of linear search has on the realized logic. For instance, one could set out to verify that under certain circumstances (for instance assuming MP on the meta-level) a realizer for the translation of MP, stating that $\neg\neg(\exists n : \mathbb{N}.\, f\, n = O)$ implies $\exists n : \mathbb{N}.\, f\, n = O$, can be constructed from find. So if indeed there is a $p$ such that

$$\{\Phi\}\, x_r \longleftarrow p \left\{ [\![\mathsf{MP}]\!]^{\mathrm{S}}_{x_r} \right\}$$

we immediately obtain that **HOL** cannot derive $\neg\mathsf{MP}$, as otherwise by Thm. 5 there would be evidence $p'$ with

$$\{\Phi\}\, x_r \longleftarrow p' \left\{ [\![\neg\mathsf{MP}]\!]^{\mathrm{S}}_{x_r} \right\}$$

yielding contradicting realizability triples. To obtain formal certainty that such a contradiction cannot follow from the axiomatic extension itsef, one can routinely verify that a consistent instance of **EffHOL** based on the monad $M(\tau) := \mathbb{N} \to \tau$ implemented by step-indexing allows to define a deterministic search function find as axiomatized (cf. [45]), and that assuming MP on the meta-level is consistent, as common in Kleene realizability.

### C. Memoizing Countable Choice

This section demonstrates an application of **EffHOL** that relies on its uniformity to identify core computational capabilities needed for realizing specifications. We illustrate this using the principle of Countable Choice (CC), whose validity status drastically changes w.r.t. the underlying effectful capabilities [19]. While deterministic computation guarantees CC, non-deterministic computation can negate CC, and stateful computation can validate CC via memoization. In fact, different forms of memoization techniques have been used in various settings to prove CC (e.g., [46]–[50]), suggesting it is somehow a more robust model of CC. The generality of **EffHOL** enables the formalization of such a robustness property, proving that memoization techniques indeed guarantee the validity of CC even in the presence of other effects.

To simplify the formalization of CC we again take $\mathbb{N}$ to be a standard encoding of the natural numbers in **HOL**, and use a predicate $\mathbb{n}_n$ verifying that $n$ is a natural number[2]For readability, we also use pairs, which can be standardly encoded in **HOL**. CC over a type $\tau$ can be axiomatized in **HOL** as:

$$\begin{aligned}\mathsf{CC} := \forall_{u_1:\mathbb{N}\times\tau\rightarrowtail\star}\ .\\ \quad \mathsf{Tot}\,(u_1) \sqsupset \exists_{u_2:\mathbb{N}\times\tau\rightarrowtail\star}\ .u_2 \subseteq u_1 \sqcap \mathsf{Det}\,(u_2) \sqcap \mathsf{Tot}\,(u_2)\end{aligned}$$

where $\mathsf{Tot}(u) = \forall n : \mathbb{N}.\ \exists i : \tau.\ \mathbb{n}_n \supset (n,i) \in u$ expresses that $u$ is a total relation, $\mathsf{Det}(u)$ that $u$ is deterministic and $u_1 \subseteq u_2$ that $u_1$ a sub-relation of $u_2$.

Crucially, in the statement of $\mathsf{Tot}(u)$ the universal quantification over natural numbers is relativized via the $\mathbb{n}_n$ predicate. Through the translation, when provided with a value $n$, a realizer $e_{u_1}$ of $\mathsf{Tot}(u_1)$ will compute a realizer, say $p_n^i$ of $(n,i) \in u_1$ for some index $i \in I$ (without specifying what $i$ is). Yet, in the presence of effectful computations, $e_{u_1}$ may not behave as a function (i.e., different applications of $e_u$ to $n$ may result in different $p_n^i$), and the relation may indeed need not be functional. To recover a functional sub-relation $u_2 \subseteq u_1$ with a realizer of $\mathsf{Tot}(u_2)$, one can use memoization techniques to store the $p_n^i$ at a given location, making sure that for each $n$ only one $p_n^i$ is ever computed. Those serve as realizers for a subrelation $u_2 \subseteq u_1$ which is now functional.

To define a realizer $e_{\mathsf{CC}}$ of CC along these lines, one essentially needs to be provided with computational features that are axiomatized by a monad with state and location, e.g., [51, 52]. Again, assume that $\Phi_{\mathsf{mem}}$ is a set of specifications which axiomatizes the structure necessary for the standard computational constructs of lookup, alloc and update. Then, using memoization as in [19] (and assuming CC on the meta-level as done with MP in the previous example) we can define a program $e_{\mathsf{CC}} : [\![\mathsf{CC}]\!]^{\mathrm{T}}$ such that:

$$\{\Phi_{\mathsf{mem}}\}\ x \longleftarrow e_{\mathsf{CC}}\left\{[\![\mathsf{CC}]\!]^{\mathrm{S}}_x\right\}$$

The interesting by-products of such an axiomatic presentation is that any instance $\textbf{EffHOL}^{\mathsf{mem}}$ of **EffHOL** providing an instantiation of the additional computational features such that the axioms of $\Phi_{\mathsf{mem}}$ hold will then have the corresponding $e_{\mathsf{CC}}$ also realizing of CC. In other words, this captures the robustness of the interpretation of CC based on memoization techniques. A concrete instance $\textbf{EffHOL}^{\mathsf{mem}}$ can be obtained with the state monad and its standard demonic (i.e., necessity) modality, mimicking the mem-SCA framework in [19]. However, this suggests that a realizer of CC can be obtained in a larger class of instantiations, and it remains to be seen if works realizing CC in different settings, such as [46, 47, 49, 50, 53] can be obtained as such instances.

## VII. From syntax to semantics : the induced evidenced frame

An approach to unify several established forms of realizability, including effectful ones, was suggested by Cohen, Miquey, and Tate [18]. That work abstracts away the core of various constructions of realizability models through a single structure, called *evidenced frame*, which focuses solely on the relationship between propositions and their evidence, leaving out the computational specifics of particular models. This is done semantically, by considering any model as an evidenced frame and providing a uniform construction of a realizability tripos from an evidenced frame. Similarly, **EffHOL** unifies several established forms of realizability, but through a syntactic translation rather than a semantic abstraction, while focusing on effectful programming languages.

Next, we show how evidenced frames can be constructed from our **EffHOL** framework. More concretely, we provide a method that takes a pure instance of **EffHOL** and defines an evidenced frame. Roughly speaking, an evidenced frame is a triple $(\Phi, E, \cdot \dot{\rightarrow} \cdot)$, where $\Phi$ is a set of propositions, $E$ is a set of evidence, and $\phi_1 \xrightarrow{e} \phi_2$ is a evidence relation on $\Phi \times E \times \Phi$, with some required relationships between the operations on propositions and the operations on evidence.

The definition of evidenced frames is at its core very semantical, and in particular, the universal implication requires to define a proposition that acts like an intersection on any set of propositions, which the syntax of **HOL** cannot account for: we can not directly define propositions of an evidenced frame as mere **HOL** propositions. In turn, as is usual, the set of (semantics evidenced frame) propositions $\Phi_{\mathrm{ef}}$ we are looking for should reflect the structure of the set of realizers we obtain through the realizability translation. For a (closed) **HOL** proposition $\psi$, the set of its realizers is given by a set of closed programs $\vdash p : [\![\psi]\!]^{\mathrm{T}}$ such that $[\![\psi]\!]^{\mathrm{S}}_p$ holds. The evidencing relation $\varphi \xrightarrow{e} \psi$ should reflect the triple $\left\{[\![\varphi]\!]^{\mathrm{S}}_p\right\}\ x \longleftarrow e\,p\left\{[\![\psi]\!]^{\mathrm{S}}_x\right\}$. Nonetheless, as was observed in earlier work, in typed settings such as those coming from modified realizability, one obtains a tripos only if the type system admits a universal type, i.e. if the setting is essentially untyped [54]. The present work faces similar issues, and to circumvent this, we consider a type erasure function $\lfloor\cdot\rfloor$ and define propositions as erasures of set of programs.

Consider a pure instance of **EffHOL**, with $\mathcal{P}$ its set of programs (which we assume to be extended with pairs and projections to handle the conjunction). We denote by $\lfloor\cdot\rfloor : \mathcal{P} \to \Lambda$ the type erasure function from $\mathcal{P}$ to the untyped computational $\lambda$-calculus, i.e. the function erasing type annotations on variables, type abstractions and type applications. We write $\Lambda$ for the set of $\lambda$-terms equipped with the same reduction (up to erasure) as $\mathcal{P}$, and we write $\mathcal{V}$ for its set of values. Then, propositions are defined as sets of values obtained as erasures of programs, while evidence are (untyped) $\lambda$-terms:

$$\Phi_{\mathrm{ef}} \triangleq \{\lfloor \mathrm{P} \rfloor \mid \mathrm{P} \subseteq \mathcal{P} \ \wedge\ \lfloor \mathrm{P} \rfloor \subseteq \mathcal{V}\} \qquad \mathrm{E}_{\mathrm{ef}} \triangleq \Lambda$$

As highlighted in Sec. VI-A, defining the evidence relation requires lifting a proposition $\varphi \in \Phi_{\mathrm{ef}}$ (defined in terms of values) to a set $\overline{\varphi}$ intuitively comprising of computations returning values in $\varphi$. For this we use the modality, which transforms an expression $e$ of index $\mathsf{R}_\tau$ into an expression $\overline{e}$

[2]This enables relativized quantifiers over $\mathbb{N}$, i.e. formulas of the shape $A \equiv \forall n : \mathbb{N}.\mathbb{n}_n \supset A'$ where $\mathbb{n}_n$ acts as a formula realized by the encoding of $n$.

of index $\mathsf{R}_{M(\tau)}$ by $\overline{e} \triangleq \{\!|\, x' : M\,(\tau) \,|\, \langle x \leftarrow x' \rangle\ x \Subset e \,|\!\}$. For a pure instance of **EffHOL**, this modality is defined internally as a specification. By considering the (meta) set-theoretic counterparts of **EffHOL**'s logical constructs (replacing comprehension terms by comprehension, logical membership by membership, and quantification by meta-quantification), the definition of the modality carries over to sets of (erased) programs, thus we define, for any $A \subseteq \mathcal{P}$, the set $\overline{A} \triangleq \{p \in \mathcal{P} \mid \langle x \leftarrow p \rangle\ x \in A\}$. We extend this definition along the erasure function by simply taking $\overline{\lfloor A \rfloor} \triangleq \lfloor \overline{A} \rfloor$. With this, we can define the evidence relation to mimic the syntactic realizability translation:

$$\phi_1 \xrightarrow{e} \phi_2 \triangleq \forall p_1 \in \phi_1 . e\, p_1 \in \overline{\phi_2}$$

**Theorem 8.** $(\Phi_{\mathrm{ef}}, \mathrm{E}_{\mathrm{ef}}, \cdot \dot{\to} \cdot)$ *is an evidenced frame.*

In particular, since any evidenced frame defines a tripos [18], connecting the dots we obtain:

**Corollary 9.** *Any pure instance of* **EffHOL** *induces a tripos and a realizability topos.*

In fact, this construction applies to any (not necessarily pure) instance for which the operation $\overline{A}$ to lift sets of values to sets of programs using the modality satisfies the meta-theoretic counterpart of the rules (`Mod-I`), (`Mod-E`), (`Mon`), $(\rightsquigarrow)$ for sets of programs. Interestingly, the resulting evidenced frame and tripos neither match the usual definitions coming from a fully typed realizability setting such as modified realizability (where types would be accounted for in all the components of the evidenced frame), nor the ones in an untyped setting [55]. In a sense, by taking types into account while erasing them, this definition lies somewhere in between these two situations, and the precise connection between them is left for future work.

## VIII. Related Work

*Evaluation Logic:* Evaluation logic [28] is a deductive system for reasoning about program evaluation, extending [30] with modalities. It resembles **EffHOL** but with a simplified polymorphic and kind structure and it uses two modalities. Its higher-order extension [56] still lacks polymorphism and a separation of logical and computational components, limiting its ability to support *typed* higher-order realizability.

*Syntactic Realizability:* Syntactic realizability, pioneered by Gödel [5] and Kreisel [6], translates Heyting Arithmetic into to a simply typed purely functional programming language (System T). Modified realizability was extended by Paulin-Mohring [34] to verify Rocq's extraction mechanism, with further advancements in [10, 11]. Realizability as a translation between type systems was also studied in [57]. In the context of interactive realizability, Birolo [58] introduces monadic realizability which focuses on the exception-state monad and is restricted to first-order Heyting arithmetic. Our work extends these frameworks to higher-order logic using effectful programming languages.

*Effectful Curry-Howard:* Many recent works extend formal systems with logical principles and computational effects, e.g. [8, 9, 21]–[24, 59]–[68]. In particular, Pédrot and Tabareau [24] prove that any *observably* effectful type theory (with other standard properties) is inconsistent. While these extensions are usually guided by specific principles or computational capabilities such as exceptions, our work defines a generic syntactic translation for **HOL**, without dependencies on or direct references to the effectful realizers at play. Recently, PCAs were extended to Monadic Combinatory Algebras (MCAs) and used to derive an alternative formulation of effectful realizability [55]. However, while **EffHOL** yields a typed syntactic notion of realizability, MCAs lead to an untyped semantic notion of realizability.

## IX. Conclusion and Future Work

In this paper, we presented a purely syntactic account of effectful realizability within a higher-order setting. **EffHOL** is a *highly expressive*, *unified* system for *internally* reasoning about *effectful* program logics in a *natural* manner. The reasoning is done internally as the language of **EffHOL** includes programs that act as realizers for **HOL** theorems via the syntactic realizability translation. **EffHOL**'s strength lies in its parameterization by a monad, capturing effectful behavior, and its natural integration of standard programming language features, such as typed realizers and effectful programs, enabling reasoning about realizers in a way akin to reasoning about programs.

Since this paper focuses on syntactic realizability, due to space constraints, we leave the presentation of a categorical semantics for **EffHOL** to future paper. This involves combining the semantics of Higher-Order Logic, System $F_\omega$, and Evaluation Logic, using indexed categories for polymorphism [69], a strong monad for computational types, along with a tripos and a T-modality for the logical components, as in [28, 30].

Future work includes direct effect handling beyond monads in **EffHOL**. This can be done by replacing the dependency on the monad by restricting some language constructs to variables in the spirit of dynamic logic [70]. Additionally, applying **EffHOL** to a case study in a complex programming language with effects is planned, though this requires addressing auxiliary details, which we reserve for future efforts. We also note that while **EffHOL** follows traditional PCA-based realizability by invoking a CbV evaluation strategy [2, 13, 45, 69], we plan to explore alternative strategies like call-by-push-value [71], which may reveal new computational behaviors.

A particularly interesting example is traditional realizability interpretations *à la* Kleene, relying on partial combinatory algebras (that is a partial variant of the untyped $\lambda$-calculus). However, since the very core of such models is untyped, obtaining them as an instance of **EffHOL** in a natural way requires further structure and is left for future work.


## Acknowledgment

We are deeply grateful to Ross Tate for his valuable ideas and insights, which significantly shaped the direction of this work. We also thank the anonymous reviewers for their constructive feedback and thoughtful suggestions, which greatly improved the clarity and quality of this paper.

## References


[1] S. C. Kleene, "On the Interpretation of Intuitionistic Number Theory," *The journal of symbolic logic*, vol. 10, no. 4, pp. 109–124, 1945.

[2] P. J. Hofstra, "Partial Combinatory Algebras and Realizability Toposes," *University of Ottowa*, 2004. [Online]. Available: https://cspages.ucalgary.ca/~robin/FMCS/FMCS_04/material/hofstra.pdf

[3] J.-L. Krivine, "Realizability in Classical Logic. In Interactive Models of Computation and Program Behaviour," *Panoramas et synthèses*, vol. 27, 2009.

[4] J. M. E. Hyland, P. T. Johnstone, and A. M. Pitts, "Tripos Theory," in *Mathematical Proceedings of the Cambridge philosophical society*, vol. 88, no. 2. Cambridge University Press, 1980, pp. 205–232.

[5] V. K. Gödel, "Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes," *dialectica*, vol. 12, no. 3-4, pp. 280–287, 1958.

[6] G. Kreisel, "Interpretation of Analysis by Means of Constructive Functionals of Finite Types," in *Constructivity in mathematics*, A. Heyting, Ed. North-Holland Pub. Co., 1959, pp. 101–128.

[7] J. van Oosten, "The Modified Realizability Topos," *Journal of pure and applied algebra*, vol. 116, no. 1-3, pp. 273–289, 1997.

[8] G. Jaber, G. Lewertowski, P.-M. Pédrot, M. Sozeau, and N. Tabareau, "The Definitional Side of the Forcing," in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, 2016, pp. 367–376.

[9] P. Pédrot and N. Tabareau, "Failure is Not an Option - An Exceptional Type Theory," in *27th European Symposium on Programming*, ser. LNCS, vol. 10801. Thessaloniki, Greece: Springer, Apr. 2018, pp. 245–271. [Online]. Available: https://hal.inria.fr/hal-01840643

[10] P. Letouzey, "A New Extraction for Coq," in *International Workshop on Types for Proofs and Programs*. Springer, 2002, pp. 200–219.

[11] Y. Forster, M. Sozeau, and N. Tabareau, "Verified Extraction from Coq to OCaml," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 52–75, 2024.

[12] J.-P. Bernardy and M. Lasson, "Realizability and Parametricity in Pure Type Systems," in *Foundations of Software Science and Computational Structures: 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 14*. Springer, 2011, pp. 108–122.

[13] R. Lepigre, "A Classical Realizability Model for a Semantical Value Restriction," in *ESOP*, ser. Lecture Notes in Computer Science, vol. 9632. Springer, 2016, pp. 476–502.

[14] E. Miquey, "A Sequent Calculus with Dependent Types for Classical Arithmetic," in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 720–729. [Online]. Available: https://doi.org/10.1145/3209108.3209199

[15] G. Geoffroy, "Classical Realizability as a Classifier for Nondeterminism," in *ACM/IEEE Symposium on Logic in Computer Science*, Oxford, United Kingdom, Jul. 2018. [Online]. Available: https://hal.science/hal-01802215

[16] D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy, "Dijkstra Monads for Free," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 515–529.

[17] K. Maillard, D. Ahman, R. Atkey, G. Martínez, C. Hriţcu, E. Rivas, and E. Tanter, "Dijkstra Monads for All," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3341708

[18] L. Cohen, É. Miquey, and R. Tate, "Evidenced Frames: A Unifying Framework Broadening Realizability Models," in *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2021, pp. 1–13.

[19] L. Cohen, S. A. Faro, and R. Tate, "The Effects of Effects on Constructivism," *Electronic Notes in Theoretical Computer Science*, vol. 347, pp. 87–120, 2019.

[20] P. Pédrot and N. Tabareau, "An Effectful Way to Eliminate Addiction to Dependence," in *Logic in Computer Science (LICS), 2017 32nd Annual ACM/IEEE Symposium on*, Reykjavik, Iceland, Jun. 2017, p. 12. [Online]. Available: https://hal.inria.fr/hal-01441829

[21] S. Boulier, P.-M. Pédrot, and N. Tabareau, "The Next 700 Syntactical Models of Type Theory," in *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, ser. CPP 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 182–194. [Online]. Available: https://doi.org/10.1145/3018610.3018620

[22] P. Pédrot, N. Tabareau, H. J. Fehrmann, and É. Tanter, "A Reasonably Exceptional Type Theory," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, pp. 108:1–108:29, 2019. [Online]. Available: https://doi.org/10.1145/3341712

[23] P. Pédrot, "Russian Constructivism in a Prefascist Theory," in *35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany*, H. Hermanns, L. Zhang, N. Kobayashi, and D. Miller, Eds. ACM, 2020, pp. 782–794. [Online]. Available: https://doi.org/10.1145/3373718.3394740

[24] P. Pédrot and N. Tabareau, "The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 58:1–58:28, 2020. [Online]. Available: https://doi.org/10.1145/3371126

[25] T. Griffin, "A Formulae-as-type Notion of Control," in *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '90. New York, NY, USA: ACM, 1990, pp. 47–58.

[26] J.-L. Krivine, "Dependent Choice, 'quote' and the Clock," *Th. Comp. Sc.*, vol. 308, pp. 259–276, 2003.

[27] J.-Y. Girard, "Interprétation Fonctionnelle Et élimination Des Coupures De L'arithmétique D'ordre Supérieur," Ph.D. dissertation, Université Paris Diderot - Paris 7, 1972.

[28] A. M. Pitts, "Evaluation Logic," in *IV Higher Order Workshop, Banff 1990: Proceedings of the IV Higher Order Workshop, Banff, Alberta, Canada 10–14 September 1990*. Springer, 1991, pp. 162–189.

[29] M. Vistrup, M. Sammler, and R. Jung, "Program Logics à la Carte," *Proc. ACM Program. Lang.*, vol. 9, no. POPL, Jan. 2025. [Online]. Available: https://doi.org/10.1145/3704847

[30] E. Moggi, "Notions of Computation and Monads," *Information and computation*, vol. 93, no. 1, pp. 55–92, 1991.

[31] B. Jacobs, *Categorical Logic and Type Theory*, ser. Studies in Logic and the Foundations of Mathematics. Amsterdam: North Holland, 1999, no. 141.

[32] J. L. Bates and R. L. Constable, "Proofs as Programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 113–136, 1985.

[33] V. Blot, "A Direct Computational Interpretation of Second-Order Arithmetic via Update Recursion," in *LICS 2022 - 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, Haïfa, Israel, Aug. 2022. [Online]. Available: https://inria.hal.science/hal-03698879

[34] C. Paulin-Mohring, "Extracting $F_\omega$'s Programs from Proofs in the Calculus of Constructions," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 89–104. [Online]. Available: https://doi.org/10.1145/75277.75285

[35] J. Girard, Y. Lafont, and P. Taylor, *Proofs and Types*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989. [Online]. Available: https://books.google.fr/books?id=6JOEQgAACAAJ

[36] J. C. Reynolds, "Polymorphism is Not Set-Theoretic," in *International Symposium on Semantics of Data Types*. Springer, 1984, pp. 145–156.

[37] E. Moggi, "Computational Lambda-Calculus and Monads," *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pp. 14–23, 1989. [Online]. Available: https://api.semanticscholar.org/CorpusID:5411355

[38] D. Scott and D. McCarty, "Reconsidering Ordered Pairs," *The Bulletin of Symbolic Logic*, vol. 14, no. 3, pp. 379–397, 2008. [Online]. Available: http://www.jstor.org/stable/20059989

[39] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM*, vol. 12, no. 10, p. 576–580, oct 1969. [Online]. Available: https://doi.org/10.1145/363235.363259

[40] P. Oliva and T. Streicher, "On Krivine's Realizability Interpretation of Classical Second-Order Arithmetic," *Fundam. Inform.*, vol. 84, no. 2, pp. 207–220, 2008.

[41] A. Miquel, "Existential Witness Extraction in Classical Realizability and via a Negative Translation," *Logical Methods in Computer Science*, vol. Volume 7, Issue 2, Apr. 2011. [Online]. Available: https://lmcs.episciences.org/1068

[42] S. Gardelle and É. Miquey, "Do CPS Translations Also Translate Realizers?" in *JFLA 2023 - 34èmes Journées Francophones des Langages Applicatifs*, T. Bourke and D. Demange, Eds., Praz-

sur-Arly, France, Jan. 2023, pp. 103–120. [Online]. Available: https://hal.inria.fr/hal-03910311

[43] G. Munch-Maccagnoni, "Focalisation and Classical Realisability," in *Computer Science Logic*, E. Grädel and R. Kahle, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 409–423.

[44] G. Kreisel, "The Non-Derivability of $\neg xA(x) \rightarrow \exists x \neg A(x)$, $A(x)$ Primitive Recursive, in Intuitionistic Formal Systems," *The Journal of Symbolic Logic*, vol. 23, no. 4, pp. 456–457, 1958.

[45] F. Richman, "Church's Thesis Without Tears," *Journal of Symbolic Logic*, vol. 48, no. 3, p. 797–803, 1983.

[46] S. Berardi, M. Bezem, and T. Coquand, "On the Computational Content of the Axiom of Choice," *Journal of Symbolic Logic*, vol. 63, no. 2, pp. 600–622, 1998. [Online]. Available: http://dx.doi.org/10.2307/2586854

[47] H. Herbelin, "A Constructive Proof of Dependent Choice, Compatible with Classical Logic," in *LICS*, 2012. [Online]. Available: https://hal.inria.fr/hal-00697240

[48] V. Blot and C. Riba, "On Bar Recursion and Choice in a Classical Setting," in *Programming Languages and Systems*, 2013.

[49] E. Miquey, "A Sequent Calculus with Dependent Types for Classical Arithmetic," in *LICS*, 2018. [Online]. Available: https://doi.org/10.1145/3209108.3209199

[50] J.-L. Krivine, "Bar Recursion in Classical Realisability: Dependent Choice and Continuum Hypothesis," in *Computer Science Logic*, vol. 62, 2016. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2016/6565

[51] A. Bauer, "What is Algebraic about Algebraic Effects and Handlers?" 2019. [Online]. Available: https://arxiv.org/abs/1807.05923

[52] G. Plotkin and J. Power, "Notions of Computation Determine Monads," in *Foundations of Software Science and Computation Structures*, M. Nielsen and U. Engberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 342–356.

[53] V. Blot, "A Direct Computational Interpretation of Second-Order Arithmetic via Update Recursion," in *LICS 2022 - 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, Haïfa, Israel, Aug. 2022.

[54] P. LIETZ and T. STREICHER, "Impredicativity entails untypedness," *Mathematical Structures in Computer Science*, vol. 12, no. 3, p. 335–347, 2002.

[55] L. Cohen, A. Grunfeld, D. Kirst, and E. Miquey, "From Partial to Monadic: Combinatory Algebra with Effects," in *10th International Conference on Formal Structures for Computation and Deduction*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 337, 2025.

[56] E. Moggi, "A Semantics for Evaluation Logic," *Fundam. Inf.*, vol. 22, no. 1,2, p. 117–152, apr 1995.

[57] J.-P. Bernardy and M. Lasson, "Realizability and Parametricity in Pure Type Systems," in *Proceedings of the 14th International Conference on Foundations of Software Science and Computational Structures: Part of the Joint European Conferences on Theory and Practice of Software*, ser. FOSSACS'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 108–122.

[58] G. Birolo, "Interactive Realizability, Monads and Witness Extraction," *arXiv preprint arXiv:1304.4091*, 2013.

[59] M. Baillon, A. Mahboubi, and P.-M. Pédrot, "Gardening with the Pythia A Model of Continuity in a Dependent Setting," in *30th EACSL Annual Conference on Computer Science Logic (CSL 2022)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), F. Manea and A. Simpson, Eds., vol. 216. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 5:1–5:18. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2022/15725

[60] M. H. Escardó, "Continuity of Gödel's System T Definable Functionals via Effectful Forcing," *Electr. Notes Theor. Comput. Sci.*, vol. 298, pp. 119–141, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2013.09.010

[61] T. Coquand and G. Jaber, "A Note on Forcing and Type Theory," *Fundam. Inform.*, vol. 100, no. 1-4, pp. 43–52, 2010. [Online]. Available: http://dx.doi.org/10.3233/FI-2010-262

[62] ——, "A Computational Interpretation of Forcing in Type Theory," in *Epistemology versus Ontology: Essays on the Philosophy and Foundations of Mathematics in Honour of Per Martin-Löf*, P. Dybjer, S. Lindström, E. Palmgren, and G. Sundholm, Eds. Dordrecht: Springer Netherlands, 2012, pp. 203–213. [Online]. Available: https://doi.org/10.1007/978-94-007-4435-6_10

[63] M. Bickford, L. Cohen, R. L. Constable, and V. Rahli, "Computability Beyond Church-Turing via Choice Sequences," in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '18. New York, NY, USA: ACM, 2018, pp. 245–254. [Online]. Available: http://doi.acm.org/10.1145/3209108.3209200

[64] L. Cohen and V. Rahli, "Constructing Unprejudiced Extensional Type Theories with Choices via Modalities," in *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel*, ser. LIPIcs, A. P. Felty, Ed., vol. 228. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 10:1–10:23. [Online]. Available: https://doi.org/10.4230/LIPIcs.FSCD.2022.10

[65] M. Bickford, L. Cohen, R. L. Constable, and V. Rahli, "Open Bar - a Brouwerian Intuitionistic Logic with a Pinch of Excluded Middle," in *CSL*, ser. LIPIcs, C. Baier and J. Goubault-Larrecq, Eds., vol. 183. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 11:1–11:23. [Online]. Available: https://doi.org/10.4230/LIPIcs.CSL.2021.11

[66] L. Cohen and V. Rahli, "Realizing Continuity Using Stateful Computations," in *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), B. Klin and E. Pimentel, Eds., vol. 252. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 15:1–15:18. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2023/17476

[67] L. Cohen, B. da Rocha Paiva, V. Rahli, and A. Tosun, "Inductive Continuity via Brouwer Trees," in *48th International Symposium on Mathematical Foundations of Computer Science (MFCS 2023)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), J. Leroux, S. Lombardy, and D. Peleg, Eds., vol. 272. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, pp. 37:1–37:16. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2023/18571

[68] A. Bauer and J. E. Hanson, "The Countable Reals," 2024. [Online]. Available: https://arxiv.org/abs/2404.01256

[69] R. A. Seely, "Categorical Semantics for Higher Order Polymorphic Lambda Calculus," *The Journal of Symbolic Logic*, vol. 52, no. 4, pp. 969–989, 1987.

[70] D. Harel, D. Kozen, and J. Tiuryn, *Dynamic Logic*. MIT Press, 2000.

[71] P. B. Levy, "Call-by-push-value: A subsuming paradigm," in *International Conference on Typed Lambda Calculi and Applications*. Springer, 1999, pp. 228–243.