A Relatively Complete Generic Hoare Logic for Order-Enriched Effects

Sergey Goncharov and Lutz Schröder
Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg
Email: {Sergey.Goncharov, Lutz.Schroeder}@cs.fau.de

Abstract—Monads are the basis of a well-established method of encapsulating side-effects in semantics and programming. There have been a number of proposals for monadic program logics in the setting of plain monads, while much of the recent work on monadic semantics is concerned with monads on enriched categories, in particular in domain-theoretic settings, which allow for recursive monadic programs. Here, we lay out a definition of order-enriched monad which imposes cpo structure on the monad itself rather than on base category. Starting from the observation that order-enrichment of a monad induces a weak truth-value object, we develop a generic Hoare calculus for monadic side-effecting programs. For this calculus, we prove relative completeness via a calculus of weakest preconditions, which we also relate to strongest postconditions.

I. INTRODUCTION

Side-effects in programming languages come in many shapes and sizes, including e.g. store and heap dependency, I/O, exceptions, and resumptions. As a means of organizing such effects in a uniform manner, thus increasing reusability of semantics, tools, verification logics, and meta-theory, monads have been proposed by Moggi [19].

Almost immediately following the discovery of the monadic programming paradigm, the first monad-based program logics appeared [23], [21]. One recurring question in this context is how to come by truth values and predicates. E.g., one may just assume a hyperdoctrine that determines the predicates, and then, e.g., explicitly impose that predicates in the state monad are state-dependent [23]. Alternatively, one may generate the predicates directly from the monadic structure. The latter approach is pursued in previous work on monad-based Hoare logic and dynamic logic [30], [32], [22], which induces from the underlying monad a canonical notion of predicate that in particular allows for a principled reconstruction of state dependency of predicates in the state monad. However, it still does assume that the truth values are provided by the underlying category, and hence do not relate to the structure of the monad. In the present work, we take this program one step further and generate the truth values from the monad itself.

To this end, we need to impose additional domain-theoretic structure on the monad, which however is needed anyway in order to support iteration; we call such monads order-enriched (to avoid terms such as ‘bounded-complete-dcpo-enriched’). We emphasize that this structure lives on the monad, not on the underlying category C, which makes our approach applicable, e.g., in the following principal cases.

– C is the category Set of sets. In this case, we cover largely the standard examples of effects as long as they account for non-termination (e.g. while the total state monad TX = S → (S × X) is not order-enriched, the partial state monad TX = S ↣ (S × X) is).

– C is a category of predomains (e.g. bottomless bounded-complete dcpo’s); then a monad on C is order-enriched if computational types accommodate bottom elements and binding respects existing finite joins on the left.

– C is a category of presheaves such as C = [I, Set] where I is the category of finite sets and injections; one monad of interest here is the local state monad [25], which we modify to allow for partiality (and hence order-enrichment):

\[(TX)n = Sn \rightarrow \int_{m \in n/I} Sm \times Xm\]

where \(S : I^{op} \rightarrow \textbf{Set}\) with \(Sn = V^n\) and the integral denotes a so-called coend (the intuitive meaning of the integral here is a sum over \(m \supseteq n\) of pairs representing the new state and the output parameterized by the set m of allocated locations; each component of this sum corresponds to allocating \(m - n\) new memory cells).

Our notion of predicate is then derived from the underlying order-enriched monad in a uniform manner using a generic notion of innocence of programs, a weakening of the generic notion of purity introduced in [31] — informally, an innocent program is a deterministic program that reads but does not write and, unlike a pure program, may fail to terminate. These notions are defined as (in)equational properties in the spirit of [8]; one of our core results characterizes the innocent monads on Set as submonads of reader monads. Truth values are then simply innocent programs of unit type, thus generalizing the tests of KAT (Kleene algebra with tests) [17]; as such, they have nothing to do with a truth value object that may be present in the base category, and e.g. may a priori be intuitionistic even on Set. In fact, the truth value object need not in general be a Heyting algebra, and our approach works also over domain-like base categories which (due to non-monotonicity of negation) do not support internal Heyting-algebra objects. Based on these concepts, we develop a monadic Hoare calculus which — unlike previous complete monadic program logics [21], [10], [22] — supports loops.

For this calculus, we prove a relative completeness theorem in spirit of Cook [5]. Like in the classical case our proof rests on the concept of weakest (liberal) preconditions. Since
these are related to dynamic logic, it is clear from previous work [31] that not all monads will provide sufficient support for them (one counterexample being the continuation monad). We show that the technique of weakest preconditions applies, in a precise sense, exactly to those monads that satisfy a mild technical requirement called sequential compatibility concerning compatibility of weakest preconditions with monadic binding.

In a closing observation, we relate weakest preconditions to strongest postconditions. It turns out that while weakest preconditions exist in general, strongest postconditions may fail to exist; in natural examples, the existence of strongest postconditions actually characterizes computational feasibility of a program such as allocating or writing to only finitely many locations in each execution.

We recall the basics of monad-based side-effects in Section II, and introduce our notions of order-enriched monads and innocence in Sections III and IV. Next, we define an imperative metalanguage with loops (Section V) as the setting for our Hoare calculus (Section VI), which we prove relatively complete via weakest preconditions in Section VII. Finally, we study strongest postconditions in Section VIII.

II. MONADS FOR COMPUTATIONS

We briefly recall the definition of a strong monad over a (locally small) Cartesian category $C$ (i.e. category with finite products), which we fix throughout. When using monads as models of effects, it is customary to present a monad $T$ as a Kleisli triple $(T, \eta, \varepsilon)$ consisting of an endomap $T$ over $Ob C$ sending an object $A$ to an object $TA$ of $A$-valued computations, a family of morphisms $\eta_A : A \to TA$, and a Kleisli star operator that assigns to every morphism $f : A \to TB$ a morphism $f^\dagger : TA \to TB$ lifting $f$ from $A$ to computations over $A$. One typical example (of many) is the state monad, which has $TA = S \to (A \times S)$ for a fixed set $S$ of states, so that morphisms $A \to TB$ may be seen as abstract state-dependent programs with input from $A$ and output in $B$; we defer further examples to Section III. This data are subject to the equations

\[ \eta^\dagger = \text{id} \quad f^\dagger \eta = f \quad (f^\dagger g)^\dagger = f^\dagger g^\dagger \]

(for $g : C \to TA$), which ensure that the Kleisli category $C_T$ of $T$, i.e. the category that has the same objects as $C$ and $C$-morphisms $A \to TB$ as morphisms $A \to B$, is actually a category, with identities $\eta_A$ and composition $(f, g) \mapsto f^\dagger g$. It is easy to check that this presentation is equivalent to the otherwise more standard one via an endofunctor $T$ and natural transformations $\eta : \text{Id} \to T$ (unit) and $\mu : TT \to T$ (multiplication). We generally use blackboard capitals $T$... to refer to monads and the corresponding Romans $T$... to refer to their functorial parts.

A monad $T$ is strong if it comes with a natural transformation $\tau_{A,B} : A \times TB \to T(A \times B)$ called its strength, satisfying a number of coherence conditions [20]. Strong monads are precisely those which support programming in terms of more than one variable, and hence are arguably the only computationally relevant ones.

Reasoning about strong monads is considerably facilitated by using Moggi’s computational metalanguage [20]. Important language features are the ret operator, which just denotes the monad unit $\eta$, and (using Haskell-style do in place of Moggi’s let) a binding construct $do \ x \leftarrow p; q$ which denotes Kleisli composition of $\lambda x. q$ with $p$, with the context propagated using the strength. Intuitively, $\text{ret} x$ just returns $x$ as a value without causing side-effects, and $do \ x \leftarrow p; q$ executes $p$, binds the result of the computation to $x$, and then executes $q$ (which may depend on $x$). We will give a full definition of an extended metalanguage in Section V; for now, we only note that the language is multisorted and as such involves typed terms in contexts $\Gamma \triangleright p : A$ where $\Gamma$, the context, is a sequence of pairs $x_i : A_i$ presenting a variable that may appear in $p$ and its type, mentioning every variable at most once. As usual, contexts are concatenated using comma-separated juxtaposition. We will refer to $p$ as a program and to $A$ as its return type. The contexts and the return types are commonly omitted. In terms of the metalanguage the monad laws can be rewritten as

\[
\begin{align*}
d & \leftarrow (do \ y \leftarrow p; q); r = do \ y \leftarrow p; x \leftarrow q; r \\
d & \leftarrow do \ x \leftarrow \text{ret} \ a; p = p(a/x) \\
d & \leftarrow do \ x \leftarrow p; \text{ret} x = p.
\end{align*}
\]

A monad morphism is a natural transformation between the underlying monad functors satisfying obvious coherence conditions w.r.t. the unit, the Kleisli star (equivalently, multiplication) and the strength, see e.g. [2] for details. A submonad of a monad $T$ is a monad $S$ together with a componentwise monic monad morphism $S \to T$ called the inclusion morphism.

An important issue in this framework is the construction of effects. Well-behaved constructions tend to be given in terms of algebraic operations as identified by Plotkin and Power [26]:

**Definition 1** (Algebraic operation). Given $n \in \mathbb{N}$ and a monad $T$ over $C$, a natural transformation $\alpha_X : (TX)^n \to TX$ is an (n-ary) algebraic operation if

- for every $f \in C(A, TB)$, $\alpha_B(f^1)^n = f^1\alpha_A$ and
- for every $A, B \in Ob C$, $\tau(\text{id} \times \alpha_B) = \alpha_{A \times B}\tau^n \partial_n$

where $\partial_n : A \times (TB)^n \to (A \times TB)^n$ is the morphism whose $i$-th component is $\text{id} \times \pi_i : A \times (TB)^n \to A \times TB$.

Thus, an algebraic operation combines several computations into one in a coherent way. Examples of algebraic operations include exception raising ($T^0 \to T$), binary nondeterministic choice ($T^2 \to T$), reading a value ranging over ${0, \ldots, n-1}$ ($T^n \to T$) and writing some value from/to a memory location ($T^1 \to T$).

III. ORDER-ENRICHMENT

The notion of monadic side-effect recalled in the previous section does not provide enough structure to cope with recursion or loops. Conditions ensuring definability of fixpoint operators that effectively impose additional domain-like structure on the base category have been discussed previously [6], [33], [31]. Here, we pursue the alternative approach to enrich only
the monad itself over suitable complete partial orders, thus broadening the range of applicability of our results as indicated in the introduction. We call this type of monads order-enriched. It will turn out that order-enrichment induces sufficient logical structure on tests for them to serve as truth values; we will base our generic Hoare logic on this observation (Section VI).

Definition 2 (Order-enriched monad). A strong monad \( \mathbb{T} \) over \( C \) is order-enriched if the following conditions are met.

- Every hom-set \( \text{Hom}(A, TB) \) carries a partial order, denoted \( \sqsubseteq \), that has a bottom element denoted by \( \perp_{A,B} \) or simply \( \perp \), joins of all directed subsets, and joins of all \( f, g \) such that \( f \sqsubseteq h, g \sqsubseteq h \) for some \( h \) (bounded completeness).
- For any \( h \in \text{Hom}(A', A) \) and any \( u \in \text{Hom}(B, TB') \), the maps
  \[
  f \mapsto f \circ h, \quad f \mapsto u \circ f, \quad f \mapsto \tau(\text{id}, f)
  \]
  preserve all existing joins (including the empty join \( \perp \)).

- Kleisli star is Scott continuous, i.e. if \( F \) is a nonempty directed subset of \( \text{Hom}(A, TB) \), then \( \bigcup_{f \in F} f^! = (\bigcup F)^! \).

Most of these conditions, including bounded completeness, descend from the standard definition of Scott domain. Bounded completeness turns out to be a critical property needed to introduce disjunction for our assertion logic in Section V. The continuity conditions above amount to requiring that expressions do \( x \leftarrow p; q \) preserve all existing joins in the left argument \( p \), and directed joins in the right argument \( q \). As usual, we denote the meet of \( f, g : A \to TB \) by \( f \sqcap g \), the join by \( f \cup g \), and the top element of \( \text{Hom}(A, TB) \) by \( \top \) if they exist. Next we summarize the most straightforward properties of order-enriched monads.

Proposition 3. Let \( \mathbb{T} \) be an order-enriched monad on \( C \), and let \( A, B \in \text{Ob}(C) \). Then

1) every nonempty subset of \( \text{Hom}(A, TB) \) having an upper bound has a least upper bound;

2) every nonempty subset of \( \text{Hom}(A, TB) \) has a greatest lower bound;

3) if \( \text{Hom}(A, TB) \) has a top element, then it is a complete lattice.

It is now apparent that the type of partial orders we involve is obtained from the definition of Scott domain by dropping the algebraicity condition, i.e., essentially, bounded-complete dcpo’s. Let \( \text{bdCpo}_{\perp} \) be the category of such partial orders with Scott continuous functions as morphisms. This category is monoidal w.r.t. the standard Cartesian structure induced by the evident forgetful functor to \( \text{Set} \). We would like to relate our definition of order-enrichment with the standard notion of enrichment [15]. To that end we first recall a characterization of algebraic operations essentially proved in [26].

Proposition 4. To give an algebraic operation \( T^n \to T \) is the same as to give a \( C^n \times C \)-indexed family of maps \( \gamma_{A,B} : \text{Hom}(A, TB)^n \to \text{Hom}(A, TB) \) satisfying the conditions

\[
\gamma_{A,B}(f_1, \ldots, f_n) \circ h = \gamma_{A',B}(f_1 \circ h, \ldots, f_n \circ h)
\quad (2)
\]

\[
u \circ \gamma_{A,B}(f_1, \ldots, f_n) = \gamma_{A,B}(u \circ f_1, \ldots, u \circ f_n)
\quad (3)
\]

\[
\tau(\text{id}, \gamma_{A,B}(f_1, \ldots, f_n)) = \gamma_{A,B}(\tau(\text{id}, f_1), \ldots, \tau(\text{id}, f_n))
\quad (4)
\]

for any \( h \in \text{Hom}(A', A) \) and any \( u \in \text{Hom}(B, TB') \).

This suggests the following definition of partial algebraic operation, intended to deal with the fact that join is a partial operation in bounded-complete dcpo’s.

Definition 5 (Partial algebraic operation). A partial algebraic operation is given by a collection of partial maps of the form \( \gamma_{A,B} : \text{Hom}(A, TB)^n \to \text{Hom}(A, TB) \) satisfying the equations (2)–(4) in the sense that whenever the left-hand side of an equation exists then so does the right-hand side, and both sides are equal.

Proposition 6. A monad \( \mathbb{T} \) is order-enriched iff its Kleisli category \( C_{\mathbb{T}} \) is a \( \text{bdCpo}_{\perp} \)-category, strength is Scott continuous in the second argument, and the finite joins (including \( \perp \)) induced by the order-enrichment are partial algebraic.

There are two ways of adapting the standard examples of computational monads to the enriched settings: by shifting to an order-enriched base category \( C \) or by modifying the monad. The first approach is embodied in the following proposition. Let \( \text{bdCpo} \) be the category of bottomless bounded-complete dcpo’s and Scott continuous maps. Recall that a \( \text{bdCpo} \)-monad is a monad over a \( \text{bdCpo} \)-category whose underlying functor is \( \text{bdCpo} \)-enriched (see e.g. [35]; since \( \text{bdCpo} \) is concrete, \( \text{bdCpo} \)-naturality of the involved natural transforms is automatic). Combining Proposition 6 with [35, Theorem 15(b)], we obtain

Proposition 7. A \( \text{bdCpo} \)-monad is order-enriched if the finite joins on its Kleisli hom-sets induced by the \( \text{bdCpo} \)-enrichment are partial algebraic and have algebraic bottom elements.

Alternatively one can enrich only the monad functor and not the underlying category, hence allowing, in particular, for \( C = \text{Set} \), which is not \( \text{bdCpo} \)-enriched.

Example 8 (Order-enriched monad). Most of the standard examples of computational monads on \( \text{Set} \) (e.g. exceptions, state, I/O, see [19]) become order-enriched as soon as we allow for explicit non-termination. We look explicitly at variants of the partial state monad, all of which are order-enriched:

1. The partial state monad on \( \text{Set} \), with functorial part \( TA = S \twoheadrightarrow (A \times S) \) where \( S \) is a fixed set of states and \( \twoheadrightarrow \) denotes partial function space, is order-enriched when equipped with the extension ordering. (Contrastingly, the total state monad \( A \twoheadrightarrow (S \to (A \times S)) \) fails to be order-enriched, as it does not have a bottom element.)

2. The non-deterministic state monad, with functorial part \( TA = S \twoheadrightarrow \mathcal{P}(A \times S) \), is order-enriched.

3. Applying the exception monad transformer, which maps a monad \( T \) to the monad \( T'(- + E) \) for a fixed set \( E \) of exception, to the partial state monad yields the so-called Java monad [14]. This monad is order-enriched and as such yields an example of binding not preserving \( \perp \) on the right (do \( x \leftarrow \text{raise } e; \perp = \ldots \))
raise e ≠ ⊥ where raise e stands for raising an exception e ∈ E).

4. Similarly, preservation of binary joins in q by do x ← p; q fails for non-deterministic monads featuring input or resumptions [13], [3], [11], essentially for the same reason that causes the well-known failure of non-deterministic choice to commute with sequential composition from the left for parallel processes modulo bisimulation. Again, these monads do admit order-enrichment.

5. We call the partial state monad TA = S ‑ (A × S)⊥ on bdCpo the domain-theoretic state monad. A typical case is S = L → V⊥ where L and V are (possibly infinite) discrete domains of values and locations, respectively. In this case, Scott continuity of a state transformer f means essentially that the output and the value of f(s) at any given location depend on the values of only finitely many locations under s : L → V⊥.

6. The topological state monad on Set (!) has a topological state space S, and TA consists of the continuous partial maps S → (A × S)⊥ with open domain, where A carries the discrete topology. Equivalently, TA = S → (A × S)⊥ where for any space Y, the space Y⊥ = Y ∪ {⊥} carries the topology generated by the topology of Y (i.e. its opens are Y⊥ and the opens of Y). One example of interest is S = L → V (total function space!) for sets L and V — this set does not carry any non-trivial domain structure, but it does carry an interesting topology, the product topology of L copies of the discrete space V. Under this topology, continuity of f ∈ TA means, again, that the value of a location under f(s) and the output depend only on the values of finitely many locations under s.

7. The partial local state monad from the introduction is order-enriched under the obvious extension ordering.

IV. INNOCENCE

Following [29] we next introduce an appropriate notion of (comparatively) well-behaved computations. To begin, we recall the notion of commutative monad [16] as based on a requirement of commutation of programs.

**Definition 9** (Commutation). Two programs p and q commute if the equation

\[ \text{do } x ← p; y ← q; \text{ret}(x, y) = \text{do } y ← q; x ← p; \text{ret}(x, y) \]

holds, where x, y /∈ Vars(p) ∪ Vars(q). If this is true for all p, q then the monad at hand is commutative.

The core notion of innocence is then defined as follows.

**Definition 10** (Innocence). Given an order-enriched monad T,

- a program p is copyable if it satisfies the equation
  \[ \text{do } x ← p; y ← p; \text{ret}(x, y) = \text{do } x ← p; \text{ret}(x, x) \]
- a program p is weakly discardable if it satisfies the inequality \[ \text{do } y ← p; \text{ret}∗ ⊆ \text{ret}∗ \]
- T is innocent if it is commutative and all programs in it are copyable and weakly discardable.

The conditions defining innocent monads are slightly less restrictive than those defining pure computations, used for similar purposes in [29]. Specifically, a monad is pure if it is innocent and satisfies discardability, i.e. for every p, do y ← p; \text{ret}∗ = \text{ret}∗.

Intuitively, weakly discardable programs may read but not write the state, and innocent programs are additionally deterministic (due to copyability). Computationally interesting monads typically fail to be innocent but have natural innocent submonads; this situation is illustrated in Example 12. For brevity, we fix the following terminology.

**Definition 11** (Predicated Monad). A predicated monad is a pair (T, P) consisting of an order-enriched monad T and an innocent submonad P of T.

**Example 12** (Innocent/Predicated Monads).

- The partiality monad P (where PA = A + 1) can be mapped into any order-enriched monad T by \[ α_A(\text{ln}(x)) = η_T(x), α_A(\text{in} ∗) = ⊥ ; \] if η is componentwise monic, then this makes P an innocent submonad of T. If C = Set and T is free, i.e. TA = μγ. F(γ + A) for some functor F, then the partiality monad is the only innocent submonad of T. This covers the case of exceptions, input, output and resumptions [19].
- If T is a partial or non-deterministic state monad (Example 8) or, e.g., the Java monad of [14] (TA = S → S × A + E × A), then the partial reader monad (PA = S → A) is an innocent submonad of T.
- If T is the partial local state monad from the introduction, then the corresponding partial local reader monad whose functorial part is given as

\[ (PA)n = Sn → An \]

is an innocent submonad of T.

Next, we establish that an innocent monad P yields a truth value object in a weak sense, P1, on which we will base our assertion language. Recall that a frame is a complete lattice in which finite meets distribute over all joins, with frame homomorphisms preserving finite meets and all joins.

**Lemma 13.** Let P be an innocent monad. Then p ∩ q = do p; q = do q; p for Γ ⊢ p, q : P1.

**Theorem 14.** Let P be an innocent monad. Then P1 is a distributive lattice object in C under the monad ordering, i.e. its hom-functor Hom(−, P1) factors through distributive lattices. In fact, Hom(−, P1) factors through frames, i.e. P1 has external joins, and finite meets in P1 distribute over these.

We emphasize that P1 need neither be internally complete nor residuated, i.e. does not in general support implication and quantifiers — e.g. in categories of domains, there will be no sensible objects supporting implication. It is one of the contributions of this work to show that despite its weakness, the arising logic does support a relatively complete Hoare logic, i.e. a weakest precondition calculus. When P1 has additional logical structure, the weakest precondition calculus will support this structure as well. We do have
Corollary 15. Let $\mathcal{P}$ be an innocent monad on $\text{Set}$. Then $P1$ is a complete Heyting algebra under the monad ordering.

Example 16. In case $\mathcal{P}$ is the partial local reader monad, it is easy to check that $P1$ is a Boolean algebra (although the ambient internal logic of the presheaf topos $[J, \text{Set}]$ is intuitionistic).

If $P1$ is even a Boolean algebra, we say that $(T, \mathcal{P})$ is classical. However, even on $\text{Set}$, not all monads are classical:

Example 17. The partial state monad has a natural innocent submonad (in fact, the largest such), the partial reader monad given by the expression $PA = S \rightarrow A$, so that $\Omega = P1$ can be identified with the powerset of $S$, a Boolean algebra. A simple example where $\Omega$ is, in general, non-classical is the topological state monad. Here, $P1$ consists essentially of the continuous functions $S \rightarrow 1_\perp$. These are in bijection with the open subsets of $S$, and hence in general form a proper Heyting algebra.

Finally, the largest innocent submonad of the domain-theoretic state monad maps $A$ to the continuous function space $S \rightarrow A_{\perp}$, so that in this case, $\Omega$ essentially consists of the Scott open subsets of $S$, in particular is again in general non-classical (not even a Heyting algebra). These notions of predicate are in accordance with the suggestions of [34].

Examples 17 and 12 intimate that an innocent monad might always be a submonad of some partial reader monad. As the main result of this section, we establish this for ranked monads on $\text{Set}$.

Theorem 18. Let $\mathcal{P}$ be a ranked innocent monad on $\text{Set}$. Then $\mathcal{P}$ is a submonad of a partial reader monad.

V. A SIMPLE IMPERATIVE METALANGUAGE

Here we consider a first order version of a simple computational metalanguage for side-effecting programs [20], which is also reminiscent of the simple imperative language $\text{Imp}$ from [27], [36] and therefore called the imperative meta-language. Our decision to drop high-order types, analogously to the previous decision to drop the Scott's algebraicity condition for dcpo's is entirely due to their irrelevance for any of our results — as such they would only amount to unnecessary restriction of generality.

Like Moggi’s original meta-language, our language is generic in the underlying side-effect and in the choice of basic statements. In a significant step beyond this, it features unbounded loops.

Let $W$ be a set of basic types. The sets of value types $A$ and types $C$ are defined by the grammar

$$A ::= W \mid 1 \mid 2 \mid A \times A \quad C ::= A \mid \Omega \mid TA \mid PA$$

where $W$ ranges over $W$. The intended reading is that $TA$ and $PA$ are types of computations and innocent computations, respectively, over $A$; $\Omega$ abbreviates $P1$ and serves as a type of truth values. Contrastingly, 2 is the type $1 + 1$ of Booleans.

Finally, a predicate type has the form $A \rightarrow \Omega$ where $A$ is a value type; predicates may be thought of as innocently side-effecting (intuitively: state-dependent) truth-valued functions.

Term formation rules for simple programs are given in the top section of Figure 1. They derive typed terms in context $\Gamma \vdash t : A$ as explained in Section II; however, we restrict $\Gamma$ to contain only variables of value types and predicate types. Application of $\text{ret}$ is restricted to terms of value types to ensure well-formedness of the resulting type. We fix a signature $\Sigma$ of typed functional symbols of the form $f : A \rightarrow C$ where $A$ is a value type and $C$ is a type. When $C$ is not a value type, then $f$ is a basic program. Typical examples are read and write operations in a store-based state monad as in Example 8, or non-deterministic assignment in a non-deterministic state monad.

We assume that $\Sigma$ contains the usual Boolean operations on $2$.

At the same time, we introduce the language of assertions to be used in our verification logics in the bottom section of Figure 1. An assertion is a program of return type $\Omega$. Note that the identification of $\Omega$ with $P1$ entails that the $\text{do}$ rule can be applied to show that when $\Gamma \vdash p : PA$ and $\Gamma, x : A \rightarrow \phi : \Omega$, then $\Gamma \vdash \text{do } x \leftarrow p; \phi : \Omega$. We include quantifiers and implication in the language but emphasize these are supported only when $P1$ has sufficient structure, e.g. lives over $\text{Set}$. The intuitive meaning of the iteration constructor $\text{init } x \leftarrow p$ while $\phi$ do $q$ is to initialize the variable $x$ by $p$ and then pass it iteratively though the loop updating it by $q$ (which can itself depend on $x$) at every iteration.

Our language offers conjunction and disjunction (but not in general implication) as well as universal and existential quantification. Moreover, we allow for $\text{fixpoint predicate constructors } \mu X. \phi$ and $\nu X. \phi$ where $X$ has a predicate type $A \rightarrow \Omega$, omitted in the notation for brevity.

We parametrize the semantics of the simple imperative metalanguage over a predicated monad $(T, \mathcal{P})$, as well as over an interpretation of basic types and operations. From now on, we assume the underlying category $\text{C}$ to be distributive [4] (but not necessarily Cartesian closed); i.e. $\text{C}$ has binary coproducts, and finite products distribute over finite coproducts. Every basic type $W$ is interpreted as an object $[W]$ in $\text{C}$; this interpretation is inductively extended to all types by $[1] = 1$, $[2] = 1 + 1$, $[A \times B] = [A] \times [B]$, $[\Omega] = P1$, $[TA] = T[A]$, $[PA] = P1[A]$. Signature forms $f : A \rightarrow C \in \Sigma$ are interpreted as morphisms $[f] : [A] \rightarrow [B]$.

From now on, we fix a predicated monad $(T, \mathcal{P})$ and an interpretation of $\Sigma$ as above.

Since we do not assume function objects in the underlying category, the interpretation of predicate variables in contexts requires some care: a context $\Gamma \vdash t : A$ as explained in Section II; however, we restrict $\Gamma$ to contain only variables of value types and predicate types. Application of $\text{ret}$ is restricted to terms of value types to ensure well-formedness of the resulting type. We fix a signature $\Sigma$ of typed functional symbols of the form $f : A \rightarrow C$ where $A$ is a value type and $C$ is a type. When $C$ is not a value type, then $f$ is a basic program. Typical examples are read and write operations in a store-based state monad as in Example 8, or non-deterministic assignment in a non-deterministic state monad.

We assume that $\Sigma$ contains the usual Boolean operations on $2$.

At the same time, we introduce the language of assertions to be used in our verification logics in the bottom section of Figure 1. An assertion is a program of return type $\Omega$. Note that the identification of $\Omega$ with $P1$ entails that the $\text{do}$ rule can be applied to show that when $\Gamma \vdash p : PA$ and $\Gamma, x : A \rightarrow \phi : \Omega$, then $\Gamma \vdash \text{do } x \leftarrow p; \phi : \Omega$. We include quantifiers and implication in the language but emphasize these are supported only when $P1$ has sufficient structure, e.g. lives over $\text{Set}$. The intuitive meaning of the iteration constructor $\text{init } x \leftarrow p$ while $\phi$ do $q$ is to initialize the variable $x$ by $p$ and then pass it iteratively though the loop updating it by $q$ (which can itself depend on $x$) at every iteration.

Our language offers conjunction and disjunction (but not in general implication) as well as universal and existential quantification. Moreover, we allow for $\text{fixpoint predicate constructors } \mu X. \phi$ and $\nu X. \phi$ where $X$ has a predicate type $A \rightarrow \Omega$, omitted in the notation for brevity.

We parametrize the semantics of the simple imperative metalanguage over a predicated monad $(T, \mathcal{P})$, as well as over an interpretation of basic types and operations. From now on, we assume the underlying category $\text{C}$ to be distributive [4] (but not necessarily Cartesian closed); i.e. $\text{C}$ has binary coproducts, and finite products distribute over finite coproducts. Every basic type $W$ is interpreted as an object $[W]$ in $\text{C}$; this interpretation is inductively extended to all types by $[1] = 1$, $[2] = 1 + 1$, $[A \times B] = [A] \times [B]$, $[\Omega] = P1$, $[TA] = T[A]$, $[PA] = P1[A]$. Signature forms $f : A \rightarrow C \in \Sigma$ are interpreted as morphisms $[f] : [A] \rightarrow [B]$.

From now on, we fix a predicated monad $(T, \mathcal{P})$ and an interpretation of $\Sigma$ as above.

Since we do not assume function objects in the underlying category, the interpretation of predicate variables in contexts requires some care: a context $\Gamma \vdash t : A$ as explained in Section II; however, we restrict $\Gamma$ to contain only variables of value types and predicate types. Application of $\text{ret}$ is restricted to terms of value types to ensure well-formedness of the resulting type. We fix a signature $\Sigma$ of typed functional symbols of the form $f : A \rightarrow C$ where $A$ is a value type and $C$ is a type. When $C$ is not a value type, then $f$ is a basic program. Typical examples are read and write operations in a store-based state monad as in Example 8, or non-deterministic assignment in a non-deterministic state monad.

We assume that $\Sigma$ contains the usual Boolean operations on $2$.
case distinction over the summands of $2 = 1 + 1$, interpreted using the distributive structure of $\mathbb{C}$ [4];

- do and ret are interpreted using Kleisli star, strength, and unit of $T$ and $\mathbb{P}$, respectively [19];

- Boolean connectives and, if present, quantifiers are interpreted using the lattice structure of $[\Omega] = P1$ (Theorem 14, Corollary 15).

We treat the semantics of predicate terms and fixpoints in more detail. Let $\Gamma = (C, H)$ as above.

- A variable $\Gamma \vdash X : A \rightarrow \Omega$ is interpreted as a projection map $H \rightarrow \text{Hom}(C \times X, \Omega)$.

- Let $\Gamma \vdash t : A$. Then $\Gamma \vdash s : A \rightarrow \Omega$. Then $s(t)$ is interpreted as the map $[s(t)] : H \rightarrow \text{Hom}(C, \Omega)$ obtained from $[s] : H \rightarrow \text{Hom}(C \times X, \Omega)$ and $\Gamma \vdash t : C$, by putting $[s(t)](h) = [s](h) \circ (\text{id}_C, [t](h))$ where $(-,-)$ denotes pairing in $C$.

- Since we do not assume function objects in $\mathbb{C}$ and hence represent predicates $\Gamma \vdash \phi : A \rightarrow \Omega$ in uncurred form, i.e. as maps $H \rightarrow \text{Hom}(C \times X, \Omega)$, $\lambda$-abstraction semantically does nothing: the interpretation of $\Gamma, x : A$ is $(C \times X, H)$, so when $\Gamma, x, : A \rightarrow \Omega$, then $[x]$ is a map $H \rightarrow \text{Hom}(C \times X, \Omega)$; we take $[\lambda x. t]$ to be the same map.

- The interpretation of $\Gamma, X : A \rightarrow \Omega \vdash \phi : A \rightarrow \Omega$ is a map $[\phi] : H \rightarrow \text{Hom}(C \times X, \Omega)$, and $[\phi]$ is interpreted as $H \rightarrow \text{Hom}(C \times X, \Omega)$ over $\Gamma$, $\lambda$-abstraction semantically does nothing: the interpretation of $\Gamma, x : A$ is $(C \times X, H)$, so when $\Gamma, x, : A \rightarrow \Omega$, then $[x]$ is a map $H \rightarrow \text{Hom}(C \times X, \Omega)$; we take $[\lambda x. t]$ to be the same map.

Definition 19. Let $[\Gamma] = (C, H)$. An assertion $\Gamma \vdash \phi : \Omega$ is valid (in $(T, \mathbb{P})$) if $[\phi](h) = \top$ for all $h \in H$.

Remark 20. Predicate variables have the context as an implicit argument in the semantics. This is necessitated by the absence of function objects in $\mathbb{C}$. Somewhat informally, it is justified by the equality $\mu X : C \times A \rightarrow \Omega. F(c, x) = (\mu X : C \times A \rightarrow \Omega. \lambda d : C. F(d, \lambda x : A. X(d, x)))(c)$ for $c : C$ representing the context (using full higher order syntax for brevity), correspondingly for $\nu$.

The definition of the semantics of the while loop requires some preliminaries. First, we introduce an operator $? : 2 \rightarrow \Omega$ by the equation $\phi ? = \text{if } \phi \text{ then } \top \text{ else } \bot$. It is easy to see that $2$ carries a natural Boolean algebra structure, with the complement of $b : 2$ denoted $\overline{b}$.


Lemma 22. Let $p, q : TA$. Then the join $(\text{do } b?: p) \sqcup (\text{do } \overline{b}?: q)$ exists and equals $(\text{if } b \text{ then } p \text{ else } q)$.

One consequence of this lemma is that if $b$ then $p$ else $q$ is Scott continuous in $p$ and $q$. We can therefore define the semantics of init $x \leftarrow ret x$ while $b do q$ (which should be read as an informal way to denote init $y \leftarrow ret x$ while $b[y/x] do q[y/x]$ for some fresh $y$) as expected, namely as the least fixpoint of the map $p \mapsto (\text{if } b \text{ then } do x \leftarrow ret x \text{ while } b do q \text{ else } p)$.

Fig. 1. Term formation rules for the simple imperative metalanguage ($\rightarrow, \forall, \exists$ supported only over $\mathbb{S}$).

6
only drawback of the short syntax is that it is not stable under variable substitution, but this will play no particular role below.

**Remark 23.** Unlike in the classical case where fixpoints are eliminated using Gödel’s β-function, we include them in the language. Fixpoints naturally arise in standard verification scenarios, e.g. when reasoning over data structures. For instance, a monad for singly linked lists can be thought of in terms of the abstract list monad, i.e. it holds for a fixed (but, up to mild side conditions, parametrized) predicated monad, i.e. it holds for a fixed interpretation. Like in [5] the core idea of the our proof is based on the notion of a weakest (liberal) precondition, which we introduce semantically by

\[ \text{wp}(x \leftarrow p, \psi) = \bigwedge \{ \phi \mid \{ \phi \} x \leftarrow p \{ \psi \} \} \]

for a program \( p \) and an assertion \( \psi \). By construction, \( \phi \subseteq \text{wp}(x \leftarrow p, \psi) \) whenever \( \{ \phi \} x \leftarrow p \{ \psi \} \). Moreover, since \( \text{do} \) preserves existing joins in the left argument, \( \text{wp} \) really yields a precondition, i.e. we have

**Lemma 26.** For all \( p, \psi \), \( \{ \text{wp}(x \leftarrow p, \psi) \} x \leftarrow p \{ \psi \} \).

A crucial ingredient of Cook’s proof in the classical case is the fact that \( \text{wp}(x \leftarrow p, \psi) \) is syntactically definable by induction over \( p \). Of course, syntactic weakest preconditions for basic programs \( f \) need to be assumed in our setting; that is, we assume that for every \( \psi \),

\[ \text{wp}(x \leftarrow f(z), \psi) \]

is expressible as an assertion.
We denote the set of axioms \( wp_f \), \( f \in \Sigma \), by \( \Delta_{\Sigma} \). We then define a syntactic version of \( wp_{\Sigma} \) of \( wp \) by

\[
\begin{align*}
wp_{\Sigma}(x & \leftarrow f(t), \psi) = wp(x \leftarrow f(z), \psi)[t/z] \quad (f \in \Sigma, z \text{ fresh}); \\
wp_{\Sigma}(x & \leftarrow \text{ret}(t), \psi) = \psi[t/x]; \\
wp_{\Sigma}(x & \leftarrow (\text{do } y \leftarrow p; q), \psi) = wp_{\Sigma}(y \leftarrow p, \wp_{\Sigma}(x \leftarrow q, \psi)); \\
wp_{\Sigma}(x & \leftarrow (\text{if } b \text{ then } p \text{ else } q), \psi) = \\
& \quad \text{if } b \text{ then } wp_{\Sigma}(x \leftarrow p, \psi) \text{ else } wp_{\Sigma}(x \leftarrow q, \psi); \\
wp_{\Sigma}(x & \leftarrow (\text{while } b \text{ do } x \leftarrow p), \psi) = \\
& \quad (\nu X. \lambda x. \text{if } b \text{ then } wp_{\Sigma}(x \leftarrow p, X(x)) \text{ else } \psi)(x).
\end{align*}
\]

Thus defined, \( wp_{\Sigma}(x \leftarrow q, \psi) \) is derivably a precondition:

**Lemma 27.** For all \( p, \psi \),

\[
\Delta_{\Sigma} \vdash_{P} \{ wp_{\Sigma}(x \leftarrow p, \psi) \} x \leftarrow p \{ \psi \}.
\]

Hence, \( wp_{\Sigma}(x \leftarrow p, \psi) \subseteq wp(x \leftarrow p, \psi) \).

To prove that \( wp_{\Sigma} \) is actually the same as \( wp \), we essentially need to show that \( wp \) satisfies the recursive definition of \( wp_{\Sigma} \), where the right-to-left implication is shown in the same way as Lemma 27. For the left-to-right implications, the only problematic case is \text{do}. Indeed, we need to postulate compatibility of preconditions with sequential composition as an additional requirement on the underlying monad:

**Lemma and Definition 28** (Sequential Compatibility). For programs \( p, q \) and an assertion \( \psi \), we have

\[
wp(x \leftarrow (\text{do } y \leftarrow p; q), \psi) \subseteq wp(y \leftarrow p, wp(x \leftarrow q, \psi))
\]

iff

\[
[x \leftarrow (\text{do } y \leftarrow p; q)] \psi \text{ implies } [y \leftarrow p] \ wp(x \leftarrow q, \psi)
\]

If these conditions hold for all \( p, q, \psi \), we say that \( (T, P) \) is sequentially compatible.

**Remark 29.** Sequential compatibility is reminiscent of the definition of a monad admitting dynamic logic \([31]\), a notable difference being that in order-enriched monads, \( wp \) becomes definable, so that only its properties instead of its existence need to be postulated. *Mutatis mutandis*, essentially the same counterexample as in \([31]\), a continuation monad, shows that not all monads are sequentially compatible.

**Lemma 30.** Let \( (T, P) \) be sequentially compatible. Then \( wp_{\Sigma}(x \leftarrow p, \psi) \supseteq wp(x \leftarrow p, \psi) \) for all \( p, \psi \).

**Proof:** Induction over \( p \).

The generic relative completeness theorem is now immediate:

**Theorem 31** (Generic Relative Completeness). Let \( (T, P) \) be sequentially compatible. Then \( T, P \vdash \{ \phi \} x \leftarrow p \{ \psi \} \Rightarrow \Delta_{\Sigma} \vdash_{P} \{ \phi \} x \leftarrow p \{ \psi \} \).

**Proof:** Let \( T, P \vdash \{ \phi \} x \leftarrow p \{ \psi \} \). Then the inequality assertion \( \phi \subseteq wp(x \leftarrow p, \psi) \), which is expressible by Lemmas 27 and 30, is valid in \( P \) and hence can be used to derive \( \{ \phi \} x \leftarrow p \{ \psi \} \) by (wk) from \( \{ wp(x \leftarrow p, \psi) \} x \leftarrow p \{ \psi \} \). The latter is derivable by Lemma 27.

**Example 32.** The generic relative completeness theorem instantiates to relative completeness of our Hoare calculus over any of the monads listed in Example 8 (as all these monads are easily shown to be sequentially compatible). The concrete instantiation depends, of course, on the choice of basic programs. We consider some examples in more detail:

1. **Partial state monad:** Working with a state set \( S = L \rightarrow V \) where \( L \) is a set of locations and \( V \) is a set of values (say, natural numbers), we can introduce a type \( V \) to represent the set of values, and basic programs \( \text{write}_l : V \rightarrow T1 \) and \( \text{read}_l : PV \) with the expected interpretation for each \( l \in L \). We thus recover the standard Hoare calculus by introducing axioms capturing the usual weakest preconditions,

\[
wp(\text{write}_l(z), \psi) = \psi'
\]

where \( \psi' \) is obtained from \( \psi \) by replacing all occurrences of \( \text{read}_l \) with \( \text{ret}(z) \). In this case, our generic completeness result instantiates exactly to Cook’s relative completeness theorem \([5]\).

2. **Non-deterministic state monad:** In otherwise the same setup as above, we can introduce non-deterministic basic programs, such as the \( \text{havoc} \) construct found in Boogie \([1]\) and ESC/Java \([7]\), which assigns an arbitrary value to a location. In this case, we have to specify \( wp(\text{havoc}_l, \psi) = \forall z. \psi' \) where as above, \( \psi' \) is obtained from \( \psi \) by replacing all occurrences of
read\textsubscript{1} with ret(z). Similarly, we can specify a non-deterministic coin toss : T\textsuperscript{2} with \( wp(x \leftarrow \text{toss}, \psi) = \psi[\perp/x] \wedge \psi[T/x] \), which then allows coding binary nondeterministic choice as \( p + q = \text{if toss then } p \text{ else } q \), and non-deterministic iteration as \( \text{init } x \leftarrow \text{ret } x \text{ in } x \leftarrow p^* = \text{while toss do } x \leftarrow p \). Our framework yields a relatively complete Hoare calculus for such languages.

3. Additional computational features: Our calculus remains sound and relatively complete when features such as exceptions, resumptions (used for modelling interleaving parallelism) or local state are added. In order to verify properties of interest for programs using these features, one does need additional logical scaffolding beyond the simplistic before-after of Hoare logic — e.g. abnormal postconditions to deal with exceptions, and stepwise invariants for resumptions. However, the basic Hoare calculus remains a crucial ingredient in the verification of such programs, and in fact some of the extended features required in the full verification logic can be encoded into the base calculus [30], [9].

4. Domain-theoretic and topological state monads: For these monads, we obtain the same results as indicated above but now for a non-classical assertion language. In other words, we show that classicality of the assertion logic is inessential for purposes of relative completeness; an added benefit of the non-classical setup is that we now know more about assertions — specifically that they are open subsets of the state space, which in the cases discussed in Example 8 means they are determined locally by finite information.

VIII. STRONGEST POSTCONDITIONS

In the classical setting, one has a dual calculus of strongest postconditions complementing weakest preconditions. In the general case, it turns out the situation is more complicated. Since \( \Omega \) is a frame, we can, of course, put

\[
sp(x, p) = \bigcap \{ \phi \mid [x \leftarrow p] \phi \}
\]

We can then accommodate preconditions by precomposing them with programs: \( sp(\phi, x, p) = sp(x, do \phi; p) \). However, unlike in the dual case of weakest preconditions, it will turn out that \( sp(x, p) \) is not always a postcondition of \( p \). We therefore introduce specific terminology for the positive case:

Definition 33 (Admitting Strongest Postconditions). Let \((\mathcal{T}, \mathcal{P})\) be a predicated monad. We say that \((\mathcal{T}, \mathcal{P})\) admits strongest postconditions if

\[
[x \leftarrow p] \ sp(x, p)
\]

for every program \( p \). A program \( p \) admits strongest postconditions if \( \{ \phi \} \ x \leftarrow p \ \{ sp(\phi, x, p) \} \) for all \( \phi \).

Proposition 34. Let \((\mathcal{T}, \mathcal{P})\) be a predicated monad that admits strongest postconditions. Then the following are equivalent.

1) \((\mathcal{T}, \mathcal{P})\) is sequentially compatible.
2) For all \( p, q, \psi \), \( [x \leftarrow (do \ y \leftarrow p; q)] \psi \) implies that \( sp(y, p) \sqsubseteq wp(x \leftarrow q, \psi) \) is valid in \( \mathcal{P} \).
3) For all \( p, q \), \( sp(y, p) = x \leftarrow q \ \{ sp(x, do \ y \leftarrow p; q) \} \).

In the classical case, strongest postconditions always exist:

Proposition 35. Let \((\mathcal{T}, \mathcal{P})\) be a predicated monad. If \( P1 \) is classical then \((\mathcal{T}, \mathcal{P})\) admits strongest postconditions.

The proof makes use of the ‘drinker’s paradox’ in the form \( \bigvee_i (a_i \Rightarrow \bigwedge_i a_i) \), which depends on classicality. If \( P1 \) is not classical, it may happen that not all programs admit strongest postconditions; the following examples suggest that admitting strongest postconditions relates to computational feasibility.

Example 36. 1. Domain-theoretic state monad: A program \( p : S \rightarrow (A \times S)_\perp \) has a strongest postcondition iff its image has a Scott open hull. Thus, for algebraic \( S \), a Scott continuous state transformer \( f : S \rightarrow (A \times S)_\perp \) admits strongest postconditions iff \( f \) is a compact element of \( S \rightarrow (A \times S)_\perp \), i.e. preserves compact elements. E.g. for \( S = L \rightarrow V_\perp \) as in Example 8, the compact elements of \( S \) are the finite states, i.e. those with only finitely many allocated locations. Then, a state transformer \( f : S \rightarrow (A \times S)_\perp \) admits strongest postconditions, i.e. is compact, iff it allocates only finitely many locations when run from a finite state \( s \).

2. Topological State Monad: When the state set \( S \) is \( T_1 \), then only open sets have open hulls, so that a continuous state transformer \( f : S \rightarrow (A \times S)_\perp \) admits strongest postconditions iff \( f \) is open. (In particular, it is not generally case that compact elements of \( TA \) admit strongest postconditions. Comparing this to the above, note that the Scott topology is only \( T_0 \).) When \( S \) is Stone, i.e. compact Hausdorff with a clopen base, then this means that \( f \) preserves clopens. E.g., in the case \( S = L \rightarrow V_\perp \) as in Example 8 (which is a Stone space), the clopens are precisely the subsets of \( S \) defined by constraints on finitely many locations — that is, admitting strongest postconditions means writing only to finitely many locations.

IX. CONCLUSION AND RELATED WORK

We have introduced a Hoare logic for programs with order-enriched effects encapsulated as monads. For this logic, we have proved relative completeness. This result is formulated as a generic completeness theorem, instantiated to completeness results for numerous monadic models; e.g. it reproduces Cook’s original completeness result but also a range of further completeness theorems for programs with additional or different computational effects, with more powerful basic operations, and with different assertion logics including logics of (Scott) open sets. Our formalisation utilizes the approach of domain theory in combination with recent developments on algebraic operations for computational effects [24], [25], [26]. This allows for a seamless integration of the assertion language with the programming language. In particular, we have shown that appropriate enrichment of a monad yields a natural frame structure on a submonad of innocent computations.

The way of enriching the monad we use is more general than the standard approach by enriching the underlying category. Instead we essentially enrich the corresponding Kleisli category. A form of monad enrichment similar to ours appears in [12]
for the entirely different purpose of defining trace semantics of coalgebraic languages.

X. FURTHER WORK

Although our assertion language extends intuitionistic first order logic in case the underlying category is $\text{Set}$, it is quite weak in the general case, and, e.g., does not in general have implication; we emphasize that this actually lends additional strength to our relative completeness result. General criteria under which the assertion language does support full intuitionistic first-order logic are under investigation. Whereas we currently do not have a perspective to induce such a structure solely from properties of the base category in any interesting cases other than $\text{Set}$, in view of Theorem 18 there is hope to obtain a strong logic of assertions by using properties of the category in combination with the innocence condition. The direction of primary interest in this respect is the case when the base category is a topos. We expect that the chances to succeed in endowing $P$ with a complete Heyting algebra critically depend on the notion of partiality offered by the topos. E.g. one can argue that assertions over the partial state monad $S \rightarrow (X \times S)$ considered over a topos support intuitionistic first-order logic once the partial function type is interpreted in such a way that $X \rightarrow 1$ is isomorphic to $P(X)$.

Additional perspectives for further work include coverage for computational effects whose full specification escapes the basic before/after paradigm that underlies Hoare logic, such as I/O, exceptions, and numerical properties — our calculus is sound and complete for monads with such features but does not express all of the requisite properties, such as abnormal termination. We expect some of these features to be encodable into the basic setting by means of additional observational operations in the spirit of [30]; a true conceptual unification of verification logics for these features however remains the subject of ongoing investigations. Orthogonally to these efforts, we are also working on an extension of our calculus with separation logic features [28] using the tensor product of monads as a generic counterpart of effect separation.

REFERENCES


