Proof Abstraction for Imperative Languages[⋆]

William L. Harrison

Dept. of Computer Science, University of Missouri,
Columbia, Missouri, USA

Abstract. Modularity in programming language semantics derives from abstracting over the structure of underlying denotations, yielding semantic descriptions that are more abstract and reusable. One such semantic framework is Liang's modular monadic semantics in which the underlying semantic structure is encapsulated with a monad. Such abstraction can be at odds with program verification, however, because program specifications require access to the (deliberately) hidden semantic representation. The techniques for reasoning about modular monadic definitions of imperative programs introduced here overcome this barrier. And, just like program definitions in modular monadic semantics, our program specifications and proofs are representation-independent and hold for whole classes of monads, thereby yielding proofs of great generality.

Keywords: Monads, Monad Transformers, Language Semantics, Program Specification and Verification.

1 Introduction

Modular monadic semantics (MMS) provides a powerful abstraction principle for denotational definitions via the use of monads and monad transformers [13,2,21] and MMS supports a modular, "mix and match" approach to semantic definition. MMS has been successfully applied to a wide variety of programming languages as well as to language compilers [8,6].

What is not well-recognized is the impact that the semantic factorization by monad transformers in MMS has on program specification and verification. Modularity comes with a price! The monad parameter to an MMS definition is a "black box" (i.e., its precise type structure is unknown) and must remain so if program abstraction is to be preserved. Yet, this makes reasoning with MMS language definitions using standard techniques frequently impossible. How does one reason about MMS specifications without sacrificing modularity and reusability? Furthermore, is there a notion of *proof* abstraction for MMS akin to its notion of *program* abstraction? This paper provides answers in the affirmative to these questions for imperative languages.

[⋆] This research supported in part by subcontract GPACS0016, System Information Assurance II, through OGI/Oregon Health & Sciences University.

N. Kobayashi (Ed.): APLAS 2006, LNCS 4279, pp. 97–113, 2006.

This paper presents a novel form of specification for reasoning about MMS definitions called *observational program specification* (OPS), as well as related proof techniques useful for proving such specifications. To reason about MMS definitions (which are parameterized by monads), it is necessary to parameterize the specifications themselves by monads as well. This is precisely what OPS does by lifting predicates to the computational level, and we refer to such lifted predicates as *observations.* Both MMS definitions and OPS specifications are parameterized by a monad that hides underlying denotational structure, thereby allowing greater generality in both programs and proofs alike. And just as MMS provides a notion of program abstraction, OPS provides a notion of *proof* abstraction. Observational program specifications and proofs are representation-independent, holding for whole classes of monads, thereby yielding proofs of great generality.

The methodology pursued here is as follows. Axioms characterizing algebraically the behavior of state monads are defined, and it is demonstrated that these axioms are preserved under monad transformer application. Then, a denotational semantics for the simple imperative language with loops is given in terms of state monads. Using OPS and "observation" computations, Hoare's classic programming logic [9] for this language is embedded into its own state-monadic semantics. Furthermore, it is demonstrated that the inference rules of this logic are derivable from the embedding, relying only on the state monad axioms and facts about observations. This provides a notion of proof abstraction for the simple imperative language because proofs in Hoare logic can now be lifted to any monad with state regardless of other effects it encapsulates!

This paper has the following structure. Section 2 motivates OPS, and Section 3 outlines background material necessary to understand this paper, including overviews of monads and monad transformers. In Section 4, the axiomatization of state monads and their preservation properties with respect to monad transformer application are stated and proved. In Section 5, the notion of observations is made precise. Section 6 presents the embedding of Hoare logic, and also the proof of soundness of this embedding. Section 7 compares the present work with related research. Conclusions and future work are outlined in Section 8.

2 Introducing Observational Specifications

As an example, consider the correctness of an imperative construct $\mathsf{p!}$ defined in a monad with a state Sto. Generally [26,15], a partial correctness specification of an imperative feature like this would take the form of a relation $\Re$ between input and output states σ_0 and σ_1, so that $\sigma_0 \,\Re\, \sigma_1$ means that the state σ_1 may result from the execution of $\mathsf{p!}$ in σ_0. If $\mathsf{p!}$ were defined in the single state monad $\mathsf{St}\,a = \mathit{Sto} \rightarrow a \times \mathit{Sto}$, then the correctness of $\mathsf{p!}$ would be written:

$$\forall \sigma_0 : \mathit{Sto}.\ \sigma_0 \,\Re\, (\pi_2(\mathsf{p!}\ \sigma_0)) \tag{1}$$

where π_2 is the second projection function $\lambda(-, x).x$. However, if $\mathsf{p!}$ were reinterpreted in the "Environment+State" monad $\mathsf{EnvSt}\,a = \mathit{Env} \rightarrow \mathit{Sto} \rightarrow a \times \mathit{Sto}$, then the above correctness specification would be rewritten as:

$$\forall \rho_0 : \mathit{Env}.\ \forall \sigma_0 : \mathit{Sto}.\ \sigma_0 \,\Re\, (\pi_2(\mathsf{p!}\ \rho_0\ \sigma_0)) \tag{2}$$

One can see from these two examples that every monad in which $\mathsf{p!}$ is interpreted requires a new correctness specification! Because specifications (1) and (2) rely on the fixed structure of St and EnvSt, respectively, there is no way of reusing them when $\mathsf{p!}$ is reinterpreted in another monad; or in other words, they are *representation-dependent* specifications. Consequently, each new specification will require a new proof as well. Because state monads may be arbitrarily complex—consider those in Figure 1—this makes proof abstraction attractive.

How does one develop a notion of *proof* abstraction akin to MMS *program* abstraction? The key insight here is that, because the language definitions we use are parameterized by a monad, it is necessary to develop a specification style that is also parameterized by a monad. The first step is to add a new, distinguished value type *prop*, denoted by the discrete CPO $\{\mathsf{tt}, \mathsf{ff}\}$. The type *prop* must be distinguished from the *Bool* type in languages which have recursive *Bool*-valued functions because the denotation of *Bool* in such cases is a pointed CPO. In the present work, it is sufficient to identify *prop* with *Bool* because the language considered here does not allow recursion over booleans.

Assume that g is a monadic operator which reads the current *Sto* state. For example in St, it would simply be $\lambda\sigma.(\sigma, \sigma)$, and it would have a similar definition in EnvSt. Then, the correctness condition $(\sigma_0 \,\Re\, \sigma_1) \in \mathit{prop}$ may then be a computed value for appropriate stores σ_0 and σ_1:

$$\begin{array}{l} \mathsf{g} \star \lambda\sigma_0. \\ \mathsf{p!} \star \lambda\_. \\ \mathsf{g} \star \lambda\sigma_1. \\ \quad \eta(\sigma_0 \,\Re\, \sigma_1) \end{array} \quad = \quad \begin{array}{l} \mathsf{p!} \star \lambda\_. \\ \quad \eta(\mathsf{tt}) \end{array} \tag{3}$$

What does this equation mean? Examining the left-hand side of Equation 3, the execution of $\mathsf{p!}$ is couched between two calls to g, of which the first call returns the input store σ_0 and the second call returns the output store σ_1 resulting from executing $\mathsf{p!}$. Note that σ_1 will reflect any updates to the store made by $\mathsf{p!}$. Finally, the truth-value of the *prop* expression $(\sigma_0 \,\Re\, \sigma_1)$ is returned. The right-hand side of Equation 3 executes $\mathsf{p!}$ and then always returns tt. Observe also that it was necessary to execute $\mathsf{p!}$ on the right-hand side so that identical effects (e.g., store updates and non-termination) would occur on both sides of the equation. Equation 3 requires that $(\sigma_0 \,\Re\, \sigma_1)$ be tt for all input and output stores σ_0 and σ_1, respectively, which is precisely what we want.

Equation 3 is a representation-*independent* specification of $\mathsf{p!}$. In the single store monad St, it means precisely the same thing as (1), while in the monad EnvSt, (3) means exactly the same thing as (2). In fact, Equation 3 makes sense in any monad where $\mathsf{p!}$ makes sense—consider the state monads in Figure 1. Such monads are called *state* monads—a notion made precise in Section 4. It is called an *observational* specification because the left-hand side of (3) gathers certain data from different stages in the computation (i.e., stores σ_0 and σ_1) and "observes" whether or not $(\sigma_0 \,\Re\, \sigma_1)$ holds.

$$
\begin{aligned}
\mathsf{M}_0\alpha &= \mathit{Sto} \to \alpha \times \mathit{Sto}\\
\mathsf{M}_1\alpha &= e_1 \to (s_1 \to (s_2 \to (\mathit{Sto} \to ((((\alpha \times s_1) \times s_2) \times \mathit{Sto}) + \mathit{err}_1)))))\\
\mathsf{M}_2\alpha &= e_1 \to ((\alpha \to (\mathit{Sto} \to ((\mathit{ans}_1 \times \mathit{Sto}) + \mathit{err}_1))) \to (\mathit{Sto} \to ((\mathit{ans}_1 \times \mathit{Sto}) + \mathit{err}_1)))\\
\mathsf{M}_3\alpha &= \ e_1 \to (e_2 \to\\
&\qquad ((\alpha \to ((\mathit{ans}_1 \to (\mathit{Sto} \to (((\mathit{ans}_2 \times \mathit{Sto}) + \mathit{err}_1) + \mathit{err}_2)))\\
&\qquad\quad \to (\mathit{Sto} \to (((\mathit{ans}_2 \times \mathit{Sto}) + \mathit{err}_1) + \mathit{err}_2))))\\
&\qquad\qquad \to ((\mathit{ans}_1 \to (\mathit{Sto} \to (((\mathit{ans}_2 \times \mathit{Sto}) + \mathit{err}_1) + \mathit{err}_2)))\\
&\qquad\qquad\quad \to (\mathit{Sto} \to (((\mathit{ans}_2 \times \mathit{Sto}) + \mathit{err}_1) + \mathit{err}_2)))))\\
&\qquad\qquad\qquad \vdots
\end{aligned}
$$

Fig. 1. State Monads on store *Sto* may be arbitrarily complex, complicating "brute force" induction on their types. Each of these monads may be created through applications of the state, environment, CPS, and error monad transformers (see Figure 2).

3 Background

This section outlines the background material necessary to understand the present work. Due to space constraints, we must assume of necessity that the reader is familiar with monads. Below we present a brief overview of monad transformers and modular monadic semantics and discuss how program modularity and abstraction arise within MMS language specifications.

Monads, Monad Transformers and Liftings. This section provides a brief overview and readers requiring more background should consult the related work (especially, Liang et al. [14]).

A structure $(\mathsf{M}, \eta, \star)$ is a *monad* if, and only if, M is a type constructor (functor) with associated operations *bind* $(\star : \mathsf{M}\alpha \to (\alpha \to \mathsf{M}\beta) \to \mathsf{M}\beta)$ and *unit* $(\eta : \alpha \to \mathsf{M}\alpha)$ obeying the well-known "monad laws" [14]:

$$
\begin{array}{rl}
(\eta\ a) \star k = k\ a & \text{(left unit)}\\
x \star \eta = x & \text{(right unit)}\\
x \star (\lambda a.(k\ a \star h) = (x \star k) \star h & \text{(assoc)}
\end{array}
$$

Given two monads, M and M', it is natural to ask if their composition, $\mathsf{M}\circ\mathsf{M}'$, is also a monad, but it is well-known that monads generally do not compose in this simple manner [2]. However, *monad transformers* do provide a form of monad composition [2,14,21]. When applied to a monad M, a monad transformer T creates a new monad M'. For example, the state monad transformer, $(\mathsf{StateT}\, s)$, is shown in Figure 2. (Here, the s is a type argument, which can be replaced by any type which is to be "threaded" through the computation.) Note that $(\mathsf{StateT}\, s\, \mathsf{Id})$ is identical to the state monad $(\mathsf{St}\, a = s \to a \times s)$. The state monad transformer also provides *update* u and *get* g operations to update and read, respectively, the new state in the "larger" monad. Figure 2 also presents (the endofunction parts of) three other commonly-used monad transformers: *environments* EnvT, *continuation-passing* ContT, and *exceptions* ErrorT. The monad

State Monad Transformer ($\mathsf{StateT}\, s$)

$$\mathsf{S}\alpha = \mathsf{StateT}\, s\, \mathsf{M}\, \alpha = s \to \mathsf{M}(\alpha \times s)$$

$$\eta_{\mathsf{S}} : \alpha \to \mathsf{S}\alpha$$
$$\eta_{\mathsf{S}}\, x = \lambda\sigma.\, \eta_{\mathsf{M}}(x, \sigma)$$

$$(\star_{\mathsf{S}}) : (\mathsf{S}\alpha) \to (\alpha \to \mathsf{S}\beta) \to (\mathsf{S}\beta)$$
$$x \star_{\mathsf{S}} f = \lambda\sigma_0.\, (x\, \sigma_0) \star_{\mathsf{M}} (\lambda(a, \sigma_1).f\, a\, \sigma_1)$$

$$\mathrm{lift}_{\mathsf{S}} : \mathsf{M}\alpha \to \mathsf{S}\alpha$$
$$\mathrm{lift}_{\mathsf{S}}\, x = \lambda\sigma.\, x \star_{\mathsf{M}} \lambda y.\, \eta_{\mathsf{M}}(y, \sigma)$$

$$\mathsf{u} : (s \to s) \to \mathsf{S}()$$
$$\mathsf{u}(\Delta : s \to s) = \lambda\sigma.\, \eta_{\mathsf{M}}((), \Delta\, \sigma)$$

$$\mathsf{g} : \mathsf{S}s$$
$$\mathsf{g} = \lambda\sigma.\, \eta_{\mathsf{M}}(\sigma, \sigma)$$

Environment Transformer ($\mathsf{EnvT} e$)

$$\begin{aligned}
\mathsf{E}\alpha &= \mathsf{EnvT}\, e\, \mathsf{M}\, \alpha = e \to \mathsf{M}\, \alpha \\
\mathrm{lift}_{\mathsf{E}}\, x &= \lambda\, (\rho : e).\, x \\
\mathsf{rdEnv} &: \mathsf{E}e \\
\mathsf{rdEnv} &= \lambda\, (\rho : e).\, \eta_{\mathsf{M}}\, \rho \\
\mathsf{inEnv} &: e \to \mathsf{E}\alpha \to \mathsf{E}\alpha \\
\mathsf{inEnv}\, \rho\, \varphi &= \lambda\, (\_ : e).\, \varphi\, \rho
\end{aligned}$$

CPS Transformer ($\mathsf{ContT} ans$)

$$\begin{aligned}
\mathsf{C}\alpha &= \mathsf{ContT}\, ans\, \mathsf{M}\, \alpha \\
&= (\alpha \to \mathsf{M}\, ans) \to \mathsf{M}\, ans \\
\mathrm{lift}_{\mathsf{C}}\, x &= (x \star_{\mathsf{M}})
\end{aligned}$$

Error Transformer ($\mathsf{ErrorT} err$)

$$\begin{aligned}
\mathsf{Err}\, \alpha &= \mathsf{ErrorT}\, err\, \mathsf{M}\, \alpha = \mathsf{M}\alpha + err \\
\mathrm{lift}_{\mathsf{Err}}\, x &= x \star_{\mathsf{M}} \lambda\, v.\, \eta_{\mathsf{M}}(\mathrm{inj}_l v)
\end{aligned}$$

Fig. 2. Examples of Monad Transformers: state (left); environment, cps and error (right) monad transformers

laws are preserved by monad transformers [13,2]. Please see Liang et al. [14] for further details.

Observe that, if M has operators defined by earlier monad transformer applications, then those operators must be redefined for the "larger" monad ($\mathsf{T}\,\mathsf{M}$). This is known as *lifting* the operators through T. Lifting is the main technical issue in [2,14]; it is related to, but should not be confused with, the lift operators in Figure 2). For each monad transformer T presented in Figure 2, the liftings of the update and get operators from M to ($\mathsf{T}\,\mathsf{M}$) are ($\mathrm{lift}_{\mathsf{T}} \circ \mathsf{u}$) and ($\mathrm{lift}_{\mathsf{T}}\, \mathsf{g}$).

The Lifting Laws capture the behavior of the lift function [14] associated with a monad transformer. Liang's definition of monad transformer requires that a lift function obeying the Lifting Laws be defined and, in his thesis[13], he defines lift operators for a wide range of monad transformers (including those in Figure 2) and verifies the Lifting Laws for them.

Definition 1 (Lifting Laws). *For monad transformer t, and monad m:* $\mathrm{lift} \circ \eta_m = \eta_{tm}$ *and* $\mathrm{lift}(x \star_m f) = (\mathrm{lift}\ x) \star_{tm} (\mathrm{lift} \circ f)$.

Modular Monadic Semantics & Program Abstraction. The principal advantage of the MMS approach to language definition is that the underlying denotational *model* can be arbitrarily complex without complicating the denotational *description* unnecessarily—what we have referred to earlier as separability. The beauty of MMS is that the equations defining $[\![t]\!]$ can be reinterpreted in a variety of monads M. To borrow a term from the language of abstract data types, the monadic semantics of programming languages yields *representation-independent* definitions. This is what prompts some authors (notably Espinosa [2]) to refer to MMS as the "ADT approach to language definition."

$\boxed{\text{Functional}}$	$\boxed{\text{Modular Monadic}}$
$\mathcal{F}\llbracket t \rrbracket : \mathit{Int}$	$\mathcal{M}\llbracket t \rrbracket : \mathsf{Id}\,\mathit{Int}$
$\mathcal{F}\llbracket i \rrbracket = i$	$\mathcal{M}\llbracket i \rrbracket = \eta(i)$
$\mathcal{F}\llbracket -e \rrbracket = -\mathcal{F}\llbracket e \rrbracket$	$\mathcal{M}\llbracket -e \rrbracket = \mathcal{M}\llbracket e \rrbracket \star \lambda v.\eta(-v)$
$\mathcal{F}\llbracket t \rrbracket : \mathit{Sto} \to \mathit{Int} \times \mathit{Sto}$	$\mathcal{M}\llbracket t \rrbracket : \mathsf{St}\,\mathit{Int}$
$\mathcal{F}\llbracket i \rrbracket\sigma = (i, \sigma)$	$\mathcal{M}\llbracket i \rrbracket = \eta(i)$
$\mathcal{F}\llbracket -e \rrbracket\sigma = \text{let } (v, \sigma') = \mathcal{F}\llbracket e \rrbracket\sigma$	$\mathcal{M}\llbracket -e \rrbracket = \mathcal{M}\llbracket e \rrbracket \star \lambda v.\eta(-v)$
$\qquad\qquad \text{in } \ (-v, \sigma')$	

Fig. 3. Program Abstraction via Modular Monadic Semantics. When the functional definition (left column, top row) is re-interpreted in a different type (left column, bottom row), the text of its definition changes radically. In the MMS setting (right column), no such change is required.

Let us consider standard functional-style language definitions and why they are representation-dependent. Consider the left column in Figure 3; it gives functional-style definitions for a simple expression language *Exp* with constants and negation. Note that the two functional semantics, $\mathcal{F}\llbracket - \rrbracket$, are defined in two settings corresponding to the identity and state monads. Both definitions of $\mathcal{F}\llbracket - \rrbracket$ are very representation-dependent—the very text of the definitions must be completely rewritten when the semantic setting changes. In contrast, MMS semantic equations ($\mathcal{M}\llbracket - \rrbracket$ in the right column of Figure 3) are free from the details of the underlying denotation because the monadic unit and bind operations handle any extra computational "stuff" (stores, environments, continuations, etc.). Since negation does not use any of this data, the same equations for $\mathcal{M}\llbracket - \rrbracket$ define *Exp* for all monads!

4 State Monads and Their Axiomatization

State monads are monads that capture the notion of computation associated with imperative programs. This section introduces the axiomatization for state monads. First, the appropriate signature is defined (state monad structures), and then the state monad axioms are given as equations on this signature. Theorem 1 shows how state monads may be created, and Theorem 2 demonstrates that any monad transformer (according to Liang's definition [14,13]) preserves imperative behavior. Lemma 1 provides a convenient generalization of the state monad axioms.

State Monad Structure. The quintuple $(\mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, \tau)$ is a *state monad structure* when: $(\mathsf{M}, \eta, \star)$ is a monad with operations unit $\eta : \alpha \to \mathsf{M}\alpha$ and bind $\star : \mathsf{M}\alpha \to (\alpha \to \mathsf{M}\beta) \to \mathsf{M}\beta$, and additional operations on τ update $\mathsf{u} : (\tau \to \tau) \to \mathsf{M}()$ and get $\mathsf{g} : \mathsf{M}\tau$. We will refer to a state monad structure $(\mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, \tau)$ simply as M if the associated operations and state type τ are clear from context. Please note that a single monad $(\mathsf{M}, \eta, \star)$ may have multiple state effects, each corresponding to multiple state monad structures.

State Monad Axiomatization. Let $\mathsf{M} = (\mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, \tau)$ be a state monad structure. M is a *state monad* if the following equations hold for any $f, g : \tau \rightarrow \tau$,

$$\begin{aligned} \mathsf{u}\, f \star \lambda\_.\mathsf{u}\, g &= \mathsf{u}\,(g \circ f) && \text{(sequencing)} \\ \mathsf{g} \star \lambda\sigma_0.\mathsf{g} \star \lambda\sigma_1.\eta(\sigma_0, \sigma_1) &= \mathsf{g} \star \lambda\sigma.\eta(\sigma, \sigma) && \text{(get-get)} \\ \mathsf{g} \star \lambda\sigma_0.\mathsf{u}\, f \star \lambda\_.\mathsf{g} \star \lambda\sigma_1.\eta(\sigma_0, \sigma_1) &= \mathsf{g} \star \lambda\sigma.\mathsf{u}\, f \star \lambda\_.\eta(\sigma, f\sigma) && \text{(get-update-get)} \end{aligned}$$

Axiom (sequencing) shows how updating by f and then updating by g is the same as just updating by their composition $(g \circ f)$. Axiom (get-get) requires that performing two g operations in succession retrieves precisely the same value. Axiom (get-update-get) states that retrieving the state before and after updating with f is the same as retrieving the state before and applying f directly.

Theorem 1 shows that a state monad may be created from any monad through the application of the state monad transformer. Theorem 2 shows that the monad resulting from a monad transformer application to a state monad (i.e., one obeying the state monad axioms) will also obey the state monad axioms. Proofs of both theorems appear in [7].

Theorem 1 (StateT creates a state monad). *For any monad* M*, let monad* $\mathsf{M}' = \mathsf{StateT}\, \mathit{sto}\, \mathsf{M}$ *and also* $\mathsf{u} : (\mathit{sto} \rightarrow \mathit{sto}) \rightarrow \mathsf{M}'()$ *and* $\mathsf{g} : \mathsf{M}'\mathit{sto}$ *be the non-proper morphisms added by* $(\mathsf{StateT}\, \mathit{sto})$*. Then* $(\mathsf{M}', \eta_{\mathsf{M}'}, \star_{\mathsf{M}'}, \mathsf{u}, \mathsf{g}, \mathit{sto})$ *is a state monad.*

Theorem 2 (Monad transformers preserve stateful behavior). *For any state monad* $\mathsf{M} = (\mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, \mathit{sto})$ *and monad transformer* T *(see Figure 2), the following state monad structure is a state monad:*

$$(\mathsf{T}\, \mathsf{M}, \eta', \star', (\mathrm{lift} \circ \mathsf{u}), \mathrm{lift}(\mathsf{g}))$$

where η'*,* $\star'$*, and* lift *are the monadic unit, bind, and lifting operations, respectively, defined by* T*.*

Lemma 1 states a number of properties of the g and u morphisms which will be useful later in the case study of Section 6.

Lemma 1. *Let* $(\mathsf{M}, \star, \eta, \mathsf{u}, \mathsf{g}, \tau)$ *be a state monad and* $\mathtt{getloc(x)} = \mathsf{g} \star \lambda\sigma.\eta(\sigma\, \mathtt{x})$ $(\mathtt{getloc(x)}$ *reads location* $\mathtt{x})$*. For any* $\mathcal{F} : \tau \times \tau \rightarrow \mathsf{M}a$ *and* $\Delta : \tau \rightarrow \tau$*:*

$$\mathsf{g} \star \lambda\sigma.\mathsf{g} \star \lambda\sigma'.\mathcal{F}(\sigma, \sigma') = \mathsf{g} \star \lambda\sigma.\mathsf{g} \star \lambda\sigma'.\mathcal{F}(\sigma, \sigma) \tag{a}$$

$$\mathsf{g} \star \lambda\sigma.\mathsf{u}\Delta \star \lambda\_.\mathsf{g} \star \lambda\sigma'.\mathcal{F}(\sigma, \sigma') = \mathsf{g} \star \lambda\sigma.\mathsf{u}\Delta \star \lambda\_.\mathcal{F}(\sigma, \Delta\sigma) \tag{b}$$

$$\mathsf{u}[x \mapsto v] \star \lambda\_.\mathtt{getloc(x)} = \mathsf{u}[x \mapsto v] \star \lambda\_.\eta(v) \tag{c}$$

5 Formalizing Observations

An *observation* is a computation which reads (and only reads!) data such as states and environments, and then observes the truth or falsity of a relation. With OPS, one inserts observations within a computation to capture information about its state or progress. In this way, they are rather reminiscent of the pre- and post-conditions of Hoare semantics, and we formalize this intuition below in Section 6. This section investigates the properties that must hold of a computation for it to be considered an observation.

Obviously, observations must manifest no observable effects (e.g., changing states, throwing exceptions, or calling continuations) or else they will affect the computation being specified. This property—called *innocence*—requires that the outcome of the computation being specified must be the same with or without interspersed observations and is defined below. Secondly, observing a relation twice in succession must yield the same truth value as observing a relation just once; this property is called *idempotence* below. Finally, the order in which two successive observations should be irrelevant. This property is called *non-interference* below.

An M-computation φ is *innocent*, if, and only if, for all M-computations γ,

$$\varphi \star \lambda\_.\, \gamma = \gamma \star \lambda v.\, \varphi \star \lambda\_.\, \eta\, v = \gamma$$

This says that the effects manifested by φ are irrelevant to γ and may be discarded. Computations φ and γ are *non-interfering* (written $\varphi \# \gamma$) means:

$$\varphi \star \lambda v.\gamma \star \lambda w.\eta(v, w) = \gamma \star \lambda w.\varphi \star \lambda v.\eta(v, w)$$

If $\varphi\#\gamma$, then their order is of no consequence. The relation # is clearly symmetric. Lastly, a computation φ is *idempotent* if, and only if,

$$\varphi \star \lambda v.\varphi \star \lambda w.\eta(v, w) = \varphi \star \lambda w.\eta(w, w)$$

That is, successive φ are identical to a single φ. The following lemma shows that idempotence may be used in a more general setting. A similar result for non-interference (not shown) holds by similar reasoning.

Lemma 2. *If $\varphi : \mathsf{M}\alpha$ is idempotent and $f : \alpha \times \alpha \to \mathsf{M}\beta$, then*

$$\varphi \star \lambda v.\varphi \star \lambda w.f(v, w) = \varphi \star \lambda w.f(w, w)$$

Proof. Applying the function "$\star f$" to both sides of the idempotence definition and using the associative and left-unit monad laws yields:

$$\begin{aligned}(\varphi \star \lambda v.\varphi \star \lambda w.\eta(v, w)) \star f &= \varphi \star \lambda v.\varphi \star \lambda w.(\eta(v, w) \star f)\\ &= \varphi \star \lambda v.\varphi \star \lambda w.f(v, w)\\ (\varphi \star \lambda w.\eta(w, w)) \star f &= \varphi \star \lambda w.(\eta(w, w) \star f)\\ &= \varphi \star \lambda w.f(w, w)\end{aligned}$$

□

Notice that stateful computation can easily lose innocence:

$$\mathsf{g} \neq \mathsf{u}[\lambda l.l + 1] \star \lambda\_.\mathsf{g}, \text{ and } \mathsf{g} \neq \mathsf{g} \star \lambda\sigma.\mathsf{u}[\lambda l.l + 1] \star \lambda\_.\eta(\sigma)$$

Continuation-manipulating computations like `callcc` ("*call with current continuation*") can also lose innocence, because they can jump to an arbitrary continuation κ_0:

$$\eta(5) \neq \eta(5) \star \lambda v.(\texttt{callcc}\ \lambda\kappa.\kappa_0 7) \star \lambda\_.\eta(v)$$

If Ω produces an error or is non-terminating, then it is not innocent:

$$\eta(5) \neq \eta(5) \star \lambda v.\Omega \star \lambda\_.\eta(v) = \Omega,$$

Examples of innocent computations. Some computations are always innocent. For example, any computation constructed from an environment monad's "read" operators (e.g., rdEnv), an environment monad's "in" operators (e.g., inEnv, assuming its argument are innocent), or from the "get" operators of a state monad (e.g., g) are always innocent. Unit computations (such as $\eta(x)$, for any x) are also always innocent. Knowing that a computation is innocent is useful in the proofs developed below, not only because an innocent computation commutes with any other computation, but because it can be also be added to any computation without effect. That is, for any arbitrary computations φ_1, φ_2 and innocent computation Υ,

$$\varphi_1 \star \lambda v.\varphi_2 = \Upsilon \star \lambda x.\varphi_1 \star \lambda v.(\Upsilon \star \lambda y.\varphi_2)$$

The values x and y computed by Υ can be used as snapshots to characterize the "before" and "after" behavior of φ_1 just as the states σ_0 and σ_1 computed by g were used in Equation 3.

Are innocent computations "pure"? A similar, but less general, notion to innocence is *purity* (attributed sometimes, apparently erroneously [18], to Moggi although the origins of the term are unclear). An M-computation φ is *pure* if, and only if, $\exists v.\varphi = \eta_{\mathsf{M}}(v)$. An innocent computation may be seen as "pure in any context." Consider the (innocent, but not pure) computation g. It is not the case that $\exists v.\mathsf{g} = \eta_{\mathsf{M}}(v)$, because g will return a different state depending on the context in which it occurs.

Three operations are used with observations. The first of these, $\mathsf{ITE} : \mathsf{M}\,\mathit{prop} \times \mathsf{M}(\tau) \times \mathsf{M}(\tau) \rightarrow \mathsf{M}(\tau)$, defines an observational version of if-then-else, while the last two, $\mathsf{AND}, \Rrightarrow$: $\mathsf{M}(\mathit{prop}) \times \mathsf{M}(\mathit{prop}) \rightarrow \mathsf{M}(\mathit{prop})$, are computational liftings of propositional connectives. These functions are defined as:

$$\begin{aligned} \mathsf{ITE}(\theta, u, v) &= \theta \star \lambda \mathit{test}.\text{if } \mathit{test} \text{ then } u \text{ else } v \\ \theta_1 \ \mathsf{AND}\ \theta_2 &= \theta_1 \star \lambda p_1.\theta_2 \star \lambda p_2.\eta(p_1 \wedge p_2) \\ \theta_1 \Rrightarrow \theta_2 &= \theta_1 \star \lambda p_1.\theta_2 \star \lambda p_2.\eta(p_1 \supset p_2) \end{aligned}$$

Here, $\wedge$, $\neg$, and $\supset$ are the ordinary propositional connectives on *prop* with the usual truth table definitions. The AND connective could be written using "short-circuit" evaluation so that it would not evaluate its second argument when the first produces ff. However, AND is intended to be applied only to innocent computations and its "termination behavior" on that restricted domain is identical to a short-circuiting definition. Lemma 3 is a property of ITE used in Section 6.

Lemma 3. $\mathsf{ITE}(\theta, x, y) \star f = \mathsf{ITE}(\theta, x \star f, y \star f)$ *for* $\theta : \mathsf{M}\,prop$.

Proof of Lemma 3.

$$\begin{aligned}
\mathsf{ITE}(\theta, x, y) \star f &= (\theta \star \lambda\beta.\ \text{if } \beta \text{ then } x \text{ else } y) \star f \\
&= \theta \star (\lambda\beta.(\text{if } \beta \text{ then } x \text{ else } y) \star f) \\
&= \theta \star (\lambda\beta.\ \text{if } \beta \text{ then } x \star f \text{ else } y \star f) \\
&= \mathsf{ITE}(\theta, x \star f, y \star f)
\end{aligned}$$

□

6 A Case Study in OPS: Hoare Logic Embedding

In this section, we show how OPS may be used to derive a programming logic for the simple imperative language with loops from its state-monadic denotational semantics. The programming logic developed here is the familiar axiomatic semantics of Hoare [9]. The soundness of the derived logic relies entirely on properties of monads and the state monad transformer; specifically, these are the state monad creation and preservation theorems (Theorems 1 and 2). These properties are key to the proof abstraction technique presented in this paper because they allow the logic to be interpreted soundly in *any* layered monad constructed with the state monad transformer.

First, we provide an overview of the syntax, semantics, and programming logic for simple imperative language with loops. Then, we develop the embedding of Hoare logic within OPS, and here is the first use of observations to model assertions (e.g., $\{x = 0\}$). The main result, Theorem 3, states that the rules of Hoare logic may be derived from the observational embedding of Hoare triples within any state-monadic semantics $[\![-]\!]$.

Syntax, Semantics, & Logic of the While Language. Figure 4 presents the syntax of the while language $\mathcal{L}$ and its programming logic. In most respects, it is entirely conventional, and it is expected that the reader has seen such definitions many times. Hoare's original logic [9], which is considered here, has a simple assertion logic, amounting to a quantifier-free logic with a single predicate $\leq$. For the sake of simplicity, we identify boolean expressions with assertions, and place them in the same syntactic class $\mathcal{B}$.

Figure 5 presents an MMS definition for $\mathcal{L}$ defined for any state monad. It is entirely conventional, except that the meaning of booleans is defined in terms of the observational embedding of assertions. The assertion embedding $\lceil - \rceil$ is the usual definition of boolean expressions.

Innocence, Non-interference, & Idempotence of $[\![e]\!]$ and $\lceil P \rceil$. It is necessary to demonstrate that the derivation of Hoare logic (presented below) is sound and the proof of this (in Theorem 3) relies on the interaction properties from Section 5 (namely, innocence, non-interference, and idempotence) hold for the assertion embedding and expression semantics of Figure 5; Lemma 4 shows just that.

$$
\begin{array}{lrcl}
\text{(Values)} & \mathcal{V} & = & () + \mathit{Int} + \mathit{prop} \\
\text{(Language)} & \mathcal{L} & ::= & \mathcal{C} \mid \mathcal{E} \mid \mathcal{B} \\
\text{(Assertions)} & \mathcal{B} & ::= & \texttt{true} \mid \texttt{false} \mid \mathcal{E}\,\texttt{leq}\,\mathcal{E} \mid \mathcal{B}\,\texttt{and}\,\mathcal{B} \mid \texttt{not}\,\mathcal{B} \\
\text{(Expressions)} & e \in \mathcal{E} & ::= & \mathit{Var} \mid \mathit{Int} \mid -\mathcal{E} \mid \mathcal{E}\texttt{+}\mathcal{E} \\
\text{(Commands)} & c \in \mathcal{C} & ::= & \texttt{skip} \mid \mathit{Var}\texttt{:=}\mathcal{E} \mid \mathcal{C}\ \texttt{;}\ \mathcal{C} \mid \texttt{if}\ \mathcal{B}\ \texttt{then}\ \mathcal{C}\ \texttt{else}\ \mathcal{C} \mid \texttt{while}\ \mathcal{B}\ \texttt{do}\ \mathcal{C} \\
\text{(Triples)} & \mathcal{T} & ::= & \{\mathcal{B}\}\ \mathcal{C}\ \{\mathcal{B}\}
\end{array}
$$

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}\ \text{(Skip)} \qquad \frac{\{P\,\texttt{and}\,b\}\ c_1\ \{Q\} \quad \{P\,\texttt{and}\,(\texttt{not}\,b)\}\ c_2\ \{Q\}}{\{P\}\ \texttt{if}\ \ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{Q\}}\ \text{(Cond)}$$

$$\frac{}{\{P[x/e]\}\ x\texttt{:=}e\ \{P\}}\ \text{(Assign)} \qquad \frac{\{P\,\texttt{and}\,b\}\ c\ \{P\}}{\{P\}\ \texttt{while}\ \ b\ \texttt{do}\ c\ \{P\,\texttt{and}\,(\texttt{not}\,b)\}}\ \text{(Iter)}$$

$$\frac{\{P\}\ c_1\ \{Q\} \quad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1\ \texttt{;}\ c_2\ \{R\}}\ \text{(Seq)} \qquad \frac{P' \supset P \quad \{P\}\,c\,\{Q\} \quad Q \supset Q'}{\{P'\}\ c\ \{Q'\}}\ \text{(Weaken)}$$

Fig. 4. Abstract Syntax & Inference rules for Simple Imperative Language. Lower case latin letters e and c typically refer to expressions and commands, respectively.

Lemma 4. *Let $e, e' \in \mathcal{E}$ and $P, P' \in \mathcal{B}$. Then, $\llbracket e \rrbracket$ and $\lceil P \rceil$ are innocent and idempotent, and $\llbracket e \rrbracket \# \llbracket e' \rrbracket$, $\llbracket e \rrbracket \# \lceil P \rceil$, and $\lceil P \rceil \# \lceil P' \rceil$.*

Lemma 4 follows directly from Axiom (get-get) by straightforward structural induction on the structure of terms.

Embedding Hoare Logic within Monadic Semantics. This section describes how Hoare logic may be interpreted within the state-monadic semantics of Figure 5. First, triples (i.e., "$\{P\}\,c\,\{Q\}$") are interpreted as particular computations, and then their satisfaction is defined as particular equations between computations. We extend the assertion embedding to triples so that:

$$\lceil \{P\}\,c\,\{Q\} \rceil = \lceil P \rceil \star \lambda \mathit{pre}.\llbracket c \rrbracket \star \lambda\_.\lceil Q \rceil \star \lambda \mathit{post}.\eta(\mathit{pre} \supset \mathit{post})$$

Triple satisfaction, written "$\models \{P\}\,c\,\{Q\}$," is defined when:

$$\lceil \{P\}\,c\,\{Q\} \rceil = \llbracket c \rrbracket \star \lambda\_.\eta(\mathsf{tt})$$

We also define the satisfaction of an implication "$\models P \supset Q$" as the following equation:

$$(\lceil P \rceil \Rightarrow \lceil Q \rceil) = \eta(\mathsf{tt})$$

We now have the tools to derive the inference rules from Figure 4 from the semantics in Figure 5. Each hypothesis and conclusion gives rise to an interpretation in the semantics via the satisfaction predicate $\models \{P\}\,c\,\{Q\}$ and the observational implication $\Rightarrow$ from Section 5. Soundness for the Hoare logic embedding is what one would expect: an inference rule from Figure 4 with hypotheses $\{\mathit{hyp}_0, \ldots, \mathit{hyp}_n\}$ and conclusion c is *observationally sound* with respect to a state monad semantics, if, whenever each $\models \mathit{hyp}_i$ holds, so does $\models c$.

Assertion Embedding:

$$\begin{array}{ll}
\lceil - \rceil : \mathcal{B} \rightarrow \mathsf{M}(\mathit{prop}) & \lceil e_1\, \mathtt{leq}\, e_2 \rceil = \llbracket e_1 \rrbracket \star \lambda v_1 . \llbracket e_2 \rrbracket \star \lambda v_2 . \eta(v_1 \leq v_2) \\
\lceil \mathtt{true} \rceil \;\; = \eta(\mathsf{tt}) & \lceil \mathtt{not}\, b \rceil \quad\;\; = \lceil b \rceil \star \lambda \beta . \eta(\neg \beta) \\
\lceil \mathtt{false} \rceil = \eta(\mathsf{ff}) & \lceil b_1\, \mathtt{and}\, b_2 \rceil = \lceil b_1 \rceil\ \mathsf{AND}\ \lceil b_2 \rceil
\end{array}$$

State-monadic Semantics:

$$\begin{array}{ll}
\llbracket - \rrbracket \ : \ \mathcal{L} \rightarrow \mathsf{M}\mathcal{V} & \llbracket -e \rrbracket \qquad = \llbracket e \rrbracket \star \lambda v . \eta(-v) \\
\llbracket i \rrbracket \ \ = \eta i & \llbracket e_0 + e_1 \rrbracket = \llbracket e_0 \rrbracket \star \lambda v_0 . \llbracket e_1 \rrbracket \star \lambda v_1 . \eta(v_0 + v_1) \\
\llbracket x \rrbracket \ = \mathtt{getloc}(x) & \llbracket \mathtt{skip} \rrbracket \quad = \eta\,() \\
\llbracket b \rrbracket \ \ = \lceil b \rceil & \llbracket c_1 \ \mathtt{;}\ c_2 \rrbracket = \llbracket c_1 \rrbracket \star \lambda \_ . \llbracket c_2 \rrbracket \\
 & \llbracket x \mathtt{:=} e \rrbracket \quad = \llbracket e \rrbracket \star \lambda v . \mathsf{u}[x \mapsto v]
\end{array}$$

$$\llbracket \mathtt{if}\ \ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \rrbracket = \llbracket b \rrbracket \star \lambda \beta . \text{if } \beta \text{ then } \llbracket c_1 \rrbracket \text{ else } \llbracket c_2 \rrbracket$$
$$\llbracket \mathtt{while}\ \ b\ \mathtt{do}\ c \rrbracket \ = \mathsf{fix}(\text{unwind } \llbracket b \rrbracket\ \llbracket c \rrbracket)$$

$$\text{unwind} : \mathsf{M}\mathit{prop} \rightarrow \mathsf{M}() \rightarrow \mathsf{M}() \rightarrow \mathsf{M}()$$
$$\text{unwind } \gamma_b\ \gamma_c\ \varphi = \gamma_b \star \lambda \beta . \text{if } \beta \text{ then } (\gamma_c \star \lambda \_ . \varphi) \text{ else } \eta()$$

Fig. 5. Assertion Embedding $\lceil - \rceil$ and State-monadic Semantics $\llbracket - \rrbracket$ of $\mathcal{L}$. Both the embedding and semantics are defined for *any* state monad $(\mathsf{M}, \eta, \star, \mathsf{u}, \mathsf{g}, \mathit{Var} \rightarrow \mathit{Int})$.

Lemma 5 is a substitution lemma for assertions. Below in the statement of Lemma 5, we distinguish numbers from numerals with an underscore "_"; that is, $\underline{v} \in \mathcal{E}$ is the numeral corresponding to the number v. Lemma 5 follows by straightforward structural induction.

Lemma 5 (Substitution Lemma for Assertions). *For expression $e \in \mathcal{E}$, assertion $P \in \mathcal{B}$, and function $f : \mathit{Int} \rightarrow \mathit{prop} \rightarrow \mathsf{M}\alpha$,*

$$\llbracket e \rrbracket \star \lambda v . \lceil P[x/e] \rceil \star (f\, v) = \llbracket e \rrbracket \star \lambda v . \lceil P[x/\underline{v}] \rceil \star (f\, v) \qquad \text{(a)}$$
$$u[x \mapsto v] \star \lambda \_ . \lceil P \rceil = \lceil P[x/\underline{v}] \rceil \star \lambda \mathit{cond} . u[x \mapsto v] \star \lambda \_ . \eta(\mathit{cond}) \qquad \text{(b)}$$

Derivation of Inference Rules. This section states the observational soundness of the Hoare logic embedding presented above in Theorem 3 and presents part of its proof.

Theorem 3 ($\lceil - \rceil$ is observationally sound). *The inference rules of Hoare logic are observationally sound with respect to any state-monadic semantics $\llbracket - \rrbracket : \mathcal{L} \rightarrow \mathsf{M}V$.*

The proof of Theorem 3 proceeds by structural induction on the inference rules using straightforward equational reasoning. Each case in the proof depends on properties of effects developed above; namely, these are innocence, idempotence and non-interference. The cases for the Skip, Assign and Weaken rules are presented below. The cases for Seq and Cond are similar to those below while the Iter rule follows by fixed-point induction; lack of space prohibits presentation of their proofs here.

Case: Skip Rule.

$$\lceil \{P\}\ \texttt{skip}\ \{P\} \rceil$$
$$= \lceil P \rceil \star \lambda \mathit{pre}.\ \llbracket \texttt{skip} \rrbracket \star \lambda\_.\lceil P \rceil \star \lambda \mathit{post}.\ \eta(\mathit{pre} \supset \mathit{post})$$

{ defn. $\llbracket \texttt{skip} \rrbracket$ }

$$= \lceil P \rceil \star \lambda \mathit{pre}.\ \eta() \star \lambda\_.\lceil P \rceil \star \lambda \mathit{post}.\ \eta(\mathit{pre} \supset \mathit{post})$$

{ innocence of $\eta()$ }

$$= \lceil P \rceil \star \lambda \mathit{pre}.\ \lceil P \rceil \star \lambda \mathit{post}.\ \eta(\mathit{pre} \supset \mathit{post})$$

{ $\lceil P \rceil$ is idempotent, Lemma 4 }

$$= \lceil P \rceil \star \lambda p.\ \eta(p \supset p)$$

{ logically valid }

$$= \lceil P \rceil \star \lambda p.\ \eta\,\mathsf{tt}$$

{ innocence of $\lceil P \rceil$ & $\eta()$ }

$$= \eta() \star \lambda\_.\ \eta\,\mathsf{tt}\ =\ \llbracket \texttt{skip} \rrbracket \star \lambda\_.\eta\mathsf{tt}$$

Case: Assign Rule.

$$\lceil \{P[x/e]\}\ x\texttt{:=}e\ \{P\} \rceil$$
$$= \lceil P[x/e] \rceil \star \lambda \mathit{pre}.\llbracket x\texttt{:=}e \rrbracket \star \lambda\_.\lceil P \rceil \star \lambda \mathit{post}.\eta(\mathit{pre} \supset \mathit{post})$$

{*defn.* $\llbracket x\texttt{:=}e \rrbracket$}

$$= \lceil P[x/e] \rceil \star \lambda \mathit{pre}.\llbracket e \rrbracket \star \lambda v.\mathsf{u}[x \mapsto v] \star \lambda\_.\lceil P \rceil \star \lambda \mathit{post}.\eta(\mathit{pre} \supset \mathit{post})$$

{$\llbracket e \rrbracket \# \lceil P \rceil$, Lemma 4}

$$= \llbracket e \rrbracket \star \lambda v.\lceil P[x/e] \rceil \star \lambda \mathit{pre}.\mathsf{u}[x \mapsto v] \star \lambda\_.\lceil P \rceil \star \lambda \mathit{post}.\eta(\mathit{pre} \supset \mathit{post})$$

{*Lemma* 5(a)}

$$= \llbracket e \rrbracket \star \lambda v.\lceil P[x/\underline{v}] \rceil \star \lambda \mathit{pre}.\mathsf{u}[x \mapsto v] \star \lambda\_.\lceil P \rceil \star \lambda \mathit{post}.\eta(\mathit{pre} \supset \mathit{post})$$

{*Lemma* 5(b)}

$$= \llbracket e \rrbracket \star \lambda v.\lceil P[x/\underline{v}] \rceil \star \lambda \mathit{pre}.\lceil P[x/\underline{v}] \rceil \star \lambda \mathit{post}.\mathsf{u}[x \mapsto v] \star \lambda\_.\eta(\mathit{pre} \supset \mathit{post})$$

{ idempotence of $\lceil P[x/\underline{v}] \rceil$, Lemma 4}

$$= \llbracket e \rrbracket \star \lambda v.\lceil P[x/\underline{v}] \rceil \star \lambda \mathit{post}.\mathsf{u}[x \mapsto v] \star \lambda\_.\eta(\mathit{post} \supset \mathit{post})$$

{ logical validity }

$$= \llbracket e \rrbracket \star \lambda v.\lceil P[x/\underline{v}] \rceil \star \lambda \mathit{post}.\mathsf{u}[x \mapsto v] \star \lambda\_.\eta(\mathsf{tt})$$

{ innocence of $\lceil P[x/\underline{v}] \rceil$, Lemma 4 }

$$= \llbracket e \rrbracket \star \lambda v.\mathsf{u}[x \mapsto v] \star \lambda\_.\eta(\mathsf{tt})$$
$$= \llbracket x\texttt{:=}e \rrbracket \star \lambda\_.\eta(\mathsf{tt})$$

Case: Weakening Rule. Assume $S \Rightarrow P$ and $\models \{P\}\ c\ \{Q\}$.
To show: $\models \{S\}\ c\ \{Q\}$. Rewriting the hypotheses of the inference rule in observational form:

$$\lceil S \rceil \star \lambda s.\lceil P \rceil \star \lambda p.\eta(s \supset p) = \eta(\mathsf{tt})$$
$$\lceil P \rceil \star \lambda p.\llbracket c \rrbracket \star \lambda\_.\lceil Q \rceil \star \lambda q.\eta(p \supset q) = \llbracket c \rrbracket \star \lambda\_.\eta(\mathsf{tt})$$

From the innocence of S and because $(\mathsf{tt} \wedge x) \equiv x$:

$$\lceil S \rceil \star \lambda s.\lceil P \rceil \star \lambda p.\llbracket c \rrbracket \star \lambda\_.\lceil Q \rceil \star \lambda q.\eta(s \supset p \wedge p \supset q) = \llbracket c \rrbracket \star \lambda\_.\eta(\mathsf{tt})$$

Since $(s \supset p \wedge p \supset q) = \mathsf{tt}$ and $(s \supset p \wedge p \supset q) \supset (s \supset q)$:

$$\lceil S \rceil \star \lambda s.\lceil P \rceil \star \lambda p.\llbracket c \rrbracket \star \lambda\_.\lceil Q \rceil \star \lambda q.\eta(s \supset q) = \llbracket c \rrbracket \star \lambda\_.\eta(\mathsf{tt})$$

By the innocence of $\lceil P \rceil$ (and because "p" is a dummy variable like "_"):

$$\lceil S \rceil \star \lambda s.[\![c]\!] \star \lambda\_.\lceil Q \rceil \star \lambda q.\eta(s \supset q) = [\![c]\!] \star \lambda\_.\eta(\mathsf{tt})$$

$\therefore \; \models \{S\}\, c\, \{Q\}$ □

7 Related Work

Structuring denotational semantics with monads and monad transformers was originally proposed by Moggi [21]. There are two complementary applications of monads in denotational semantics. The first is to use monads to provide a precise typing for effects in a language, while the second uses monads for modularity via monadic encapsulation of the underlying denotational structure. MMS fits squarely in this second category. Hudak, Liang, and Jones [14] and Espinosa [2] use monads and monad transformers to create modular, extensible interpreters. Recent promising work in categorical semantics [25,4] investigates more general approaches to combining monads than with monad transformers, although the cases for certain computational monads (chiefly, the continuation monad) are apparently still open problems as of this writing.

Modularity in programming language semantics is provided by a number of semantic frameworks including *action semantics* [22], *high-level semantics* [12], and *modular monadic semantics* [14,13]. Modularity in these frameworks stems from their organization according to a notion of program abstraction called *separability* [12]: they all provide a mechanism for separating the denotational description of a language (e.g., semantic equations) from its underlying denotational representation. Modularity—or rather the separability principle underlying it—can be at odds with program verification, however, because program specifications (i.e., predicates) are typically written with respect to a *fixed* denotational structure.

Liang [13] addresses the question of reasoning about MMS definitions for monads involving a single environment. He axiomatizes the environment operators rdEnv and inEnv, and shows that these axioms hold in any monad constructed with standard monad transformers (with a weak restriction on the order of transformer application—cf. Section 3). Liang's work provided an early inspiration for this one, but OPS is more powerful in a number of respects. Firstly, observations allow specifications to make finer-grained distinctions based on predicates applied to semantic data internal to the underlying monad. The work developed in [13] only allows equations between terms in the signature $\star$ (bind), η, rdEnv, and inEnv—no statements about the computed environments are possible. Secondly, observations may characterize relationships between any data internal to the underlying monad as well.

OPS was developed to verify a particular form of MMS definition, namely, *modular compilers* [8,6]. Modular compilation is a compiler construction technique allowing the assembly of compilers for high-level programming languages from reusable compiler building blocks (RCBBs). Each RCBB is, in fact, a denotational language definition factored by a monad transformer. Modular compiler verification involves specifying the behavior and interaction of multiple, "layered" effects, instead of just a single state as is presented here. The non-interference property for observations has also been used to characterize "non-interference" information security [5] by controlling "inter-layer" interaction between security levels [7].

OPS is reminiscent of programming logics such as specification and Floyd-Hoare logics [26,23,15] with observations playing a similar role to assertions (e.g., "$\{x = 0\}$"). Evaluation logic [24] is a typed, modal logic extending the computational lambda calculus [17]. It is equipped with "evaluation" modalities signifying such properties as "if E evaluates to x, then $\phi(x)$ holds". Moggi sketches how a number of programming logics, including Hoare logic, may be embedded into evaluation logic [19] and provides a similar, but less general, axiomatization of state. Führmann [3] introduces classifications for monadic effects called "effectoids". Among these are "discardable," "copyable" and "disjoint" effectoids that correspond closely to innocent, idempotent, and non-interfering computations, respectively. Schröder and Mossakowski [27] define a similar notion to discardable/innocent as well called "side-effect free". Instead of using observations to access intermediate data from a computation, their work incorporates a modality rather like the aforementioned evaluation logic modality to interpret Hoare logic monadically. The present work differs from theirs also in that here all monads are layered (i.e., produced by applications of monad transformers). Here, the monads in which the Hoare logic embedding is valid are determined by construction alone; this is valuable considering their potential complexity (see Figure 1).

Launchbury and Sabry [11] produced an axiomatization of monadic state, later used by Ariola and Sabry [1] to prove the correctness of an implementation of monadic state. Their axioms fulfill a similar role to the state monad axioms described in Section 4. They introduce an observation-like construct for describing the shape of the store, `sto` σ c, where σ is a store and c is a computation to be executed in σ. Observations may be seen as generalizing this `sto` by relating *any* data (states, environments, etc.) internal to the monad.

Kleene algebras with tests (KAT) are two-sorted algebraic structures which form an equational system for reasoning about programs [10]. A KAT has one sort for "programs" and another sort for "tests." These tests play a similar role to observations in OPS. Non-interference and idempotence properties of observations correspond to multiplicative commutation and idempotence of tests, while innocence corresponds to the commutation of non-test elements. OPS and KAT are both equational systems, although OPS, being embedded in the host language semantics, is less abstract in some sense. An interesting open question is whether OPS may form a general class of computational models of KATs, thereby providing a more compact algebraic way of reasoning with observations.

8 Concluding Remarks

OPS is a powerful and expressive specification technique for reasoning about modular definitions without sacrificing modularity. Semantic frameworks which promote modularity (like the MMS framework considered here) do so at a cost:

reasoning about such definitions is complicated by the separability principle used to gain modularity in the first place. In the case of MMS, the source of this difficulty lies in the disparity between the incompatible settings (i.e., computational and value, respectively) of programs and specifications. The solution presented here resolves this disparity by making specifications compatible with programs through the lifting of predicates to the computational level.

Monad transformers are well known as a structure for program abstraction and this article demonstrates how they give rise to a corresponding notion of proof abstraction as well. With OPS, program proofs hold in any monad in which the program itself makes sense. If an MMS program is written for a particular signature (i.e., those operators added by monad transformers) and behavior-preserving liftings exist for that signature, then the program makes sense—that is, after all, what "liftings exist" means. It is not surprising that if a monadic interface adequately captures the behavior of that same signature, then a program proof relying on that interface should hold as well.

OPS was originally developed for verifying modular compilers [6], and its application within formal methods and high-assurance software development remains an active area of research. To that end, establishing connections between OPS and other verification formalisms—programming logics such as evaluation logic [20] and semantics-based reasoning techniques such as logical relations [16]—is expected to yield useful results.

References

1. Zena M. Ariola and Amr Sabry. Correctness of monadic state: an imperative call-by-need calculus. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 62–73, New York, NY, 1998.
2. David Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.
3. Carsten Führmann. Varieties of effects. In *FoSSaCS '02: Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, pages 144–158, London, UK, 2002. Springer-Verlag.
4. Neil Ghani and Christoph Lüth. Composing monads using coproducts. In *ACM International Conference on Functional Programming*, pages 133–144, 2002.
5. Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proceedings of the 1982 Symposium on Security and Privacy (SSP '82)*, pages 11–20. IEEE Computer Society Press, 1990.
6. William Harrison. *Modular Compilers and Their Correctness Proofs*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.
7. William Harrison and James Hook. Achieving information flow security through precise control of effects. In *18th IEEE Computer Security Foundations Workshop (CSFW05)*, June 2005.
8. William Harrison and Samuel Kamin. Metacomputation-based compiler architecture. In *5th International Conference on the Mathematics of Program Construction, Ponte de Lima, Portugal*, volume 1837 of *Lecture Notes in Computer Science*, pages 213–229. Springer-Verlag, 2000.
9. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

10. Dexter Kozen. On Hoare logic and Kleene algebra with tests. *ACM Transactions on Computational Logic*, 1(1):60–76, 2000.
11. John Launchbury and Amr Sabry. Monadic state: Axiomatization and type safety. In *ACM SIGPLAN International Conference on Functional Programming*, pages 227–238, 1997.
12. Peter Lee. *Realistic Compiler Generation.* Foundations of Computing Series. MIT Press, 1989.
13. Sheng Liang. *Modular Monadic Semantics and Compilation.* PhD thesis, Yale University, 1997.
14. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 333–343. ACM Press, 1995.
15. Jacques Loeckx, Kurt Sieber, and Ryan D. Stansifer. *The Foundations of Program Verification.* Wiley-Teubner Series in Computer Science. Wiley, Chichester, second edition edition, 1987.
16. John C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 8, pages 365–458. North-Holland, New York, NY, 1990.
17. E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
18. Eugenio Moggi. Personal communication with author.
19. Eugenio Moggi. Representing program logics in evaluation logic. Unpublished manuscript, available online., 1994.
20. Eugenio Moggi. A semantics for evaluation logic. *FUNDINF: Fundamenta Informatica*, 22, 1995.
21. Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ., 90.
22. Peter D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
23. David A. Naumann. Calculating sharp adaptation rules. *Information Processing Letters*, 77(2–4):201–208, 2001.
24. Andrew M. Pitts. Evaluation logic. In G. Birtwistle, editor, *IVth Higher Order Workshop, Banff 1990*, Workshops in Computing, pages 162–189. Springer-Verlag, Berlin, 1991.
25. Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11:69–94, 2003.
26. John C. Reynolds. *The Craft of Programming.* Prentice Hall, 1981.
27. Lutz Schröder and Till Mossakowski. Monad-independent Hoare logic in HasCASL. In *Fundamental Approaches to Software Engineering*, volume 2621 of *LNCS*, pages 261–277. Springer, 2003.