Dijkstra Monads for All

KENJI MAILLARD, Inria, France and ENS, France
DANEL AHMAN, University of Ljubljana, Slovenia
ROBERT ATKEY, University of Strathclyde, UK
GUIDO MARTÍNEZ, CIFASIS-CONICET, Argentina
CĂTĂLIN HRIŢCU, Inria, France
EXEQUIEL RIVAS, Inria, France
ÉRIC TANTER, University of Chile, Chile and Inria, France

This paper proposes a general semantic framework for verifying programs with arbitrary monadic side-effects using Dijkstra monads, which we define as monad-like structures indexed by a specification monad. We prove that any monad morphism between a computational monad and a specification monad gives rise to a Dijkstra monad, which provides great flexibility for obtaining Dijkstra monads tailored to the verification task at hand. We moreover show that a large variety of specification monads can be obtained by applying monad transformers to various base specification monads, including predicate transformers and Hoare-style pre- and postconditions. For defining correct monad transformers, we propose a language inspired by Moggi's monadic metalanguage that is parameterized by a dependent type theory. We also develop a notion of algebraic operations for Dijkstra monads, and start to investigate two ways of also accommodating effect handlers. We implement our framework in both Coq and F*, and illustrate that it supports a wide variety of verification styles for effects such as exceptions, nondeterminism, state, input-output, and general recursion.

CCS Concepts: • **Theory of computation → Program specifications**; **Program verification**; *Program semantics*; *Pre- and post-conditions*; Type theory.

Additional Key Words and Phrases: program verification, side-effects, monads, dependent types, foundations

ACM Reference Format:
Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra Monads for All. *Proc. ACM Program. Lang.* 3, ICFP, Article 104 (August 2019), 29 pages. https://doi.org/10.1145/3341708

1 INTRODUCTION

The aim of this paper is to provide a semantic framework for specifying and verifying programs with arbitrary side-effects modeled by computational monads [Moggi 1989]. We base this framework on Dijkstra monads, which have already proven valuable in practice for verifying effectful code [Protzenko and Parno 2019; Swamy et al. 2016]. A Dijkstra monad $\mathcal{D}\ A\ w$ is a monad-like structure that classifies effectful computations returning values in A and specified by $w : WA$,

Authors' addresses: Kenji Maillard, Inria, Paris, France , ENS, Paris, France; Danel Ahman, Faculty of Mathematics and Physics, University of Ljubljana, Ljubljana, Slovenia; Robert Atkey, University of Strathclyde, Glasgow, UK; Guido Martínez, CIFASIS-CONICET, Rosario, Argentina; Cătălin Hriţcu, Inria, Paris, France; Exequiel Rivas, Inria, Paris, France; Éric Tanter, Computer Science Department (DCC), University of Chile, Santiago, Chile , Inria, Paris, France.

2475-1421/2019/8-ART104
https://doi.org/10.1145/3341708

where W is what we call a *specification monad*.[1] A typical specification monad contains predicate transformers mapping postconditions to preconditions. For instance, for computations in the state monad $\mathrm{St}\,A = S \to A \times S$, a natural specification monad is $W^{\mathrm{St}}A = (A \times S \to \mathbb{P}) \to (S \to \mathbb{P})$, mapping postconditions, which in this case are predicates on final results and states, to preconditions, which are predicates on initial states (here $\mathbb{P}$ stands for the internal type of propositions). However, given an *arbitrary* monadic effect, how do we find such a specification monad? Is there a *single* specification monad that we can associate to each effect? If not, what are the *various* alternatives, and what are the constraints on this *association* for obtaining a proper Dijkstra monad?

A partial answer to this question was provided by the *Dijkstra Monads for Free* (*DM4Free*) approach of Ahman et al. [2017]: from a computational monad defined as a term in a metalanguage called DM, a (single) canonical specification monad is automatically derived through a syntactic translation. Unfortunately, while this approach works for stateful and exceptional computations, it cannot handle several other effects, such as input-output (IO), due to various syntactic restrictions in DM.

To better understand and overcome such limitations, we make the novel observation that a computational monad in DM is essentially a monad transformer applied to the identity monad; and that the specification monad is obtained by applying this monad transformer to the continuation monad $\mathrm{Cont}_{\mathbb{P}}A = (A \to \mathbb{P}) \to \mathbb{P}$. Returning to the example of state, the specification monad $W^{\mathrm{St}}A$ can be obtained from the state monad transformer $\mathrm{StT}\,M\,A = S \to M(A \times S)$. This reinterpretation of the *DM4Free* approach sheds light on its limitations: For a start, the class of supported computational monads is restricted to those that can be decomposed as a monad transformer applied to the identity monad. However, this rules out various effects such as nondeterminism or IO, for which no proper monad transformer is known [Adámek et al. 2012; Bowler et al. 2013; Hyland et al. 2007].

Further, obtaining both the computational and specification monads from the same monad transformer introduces a very tight coupling. In particular, in *DM4Free* one cannot associate *different* specification monads with a particular effect. For instance, the exception monad $\mathrm{Exc}\,A = A + E$ is associated by *DM4Free* with the specification monad $W^{\mathrm{Exc}}A = ((A + E) \to \mathbb{P}) \to \mathbb{P}$, by applying the exception monad transformer $\mathrm{ExcT}\,M\,A = M(A + E)$ to $\mathrm{Cont}_{\mathbb{P}}$. This specification monad requires the postcondition to account for both the success and failure cases. While this is often desirable, at times it may be more convenient to use the simpler specification monad $\mathrm{Cont}_{\mathbb{P}}$ directly, allowing exceptions to be thrown freely, without having to explicitly allow this in specifications. Likewise, for IO, one may wish to have rich specifications that depend on the *history* of interactions with the external world, or simpler *context-free* specifications that are as local as possible. In general, one should have the freedom to choose a specification monad that is expressive enough for the verification task at hand, but also simple enough so that verification is manageable in practice.

Moreover, even for a fixed computational monad and a fixed specification monad there can be more than one way to associate the two in a Dijkstra monad. For instance, to specify exceptional computations using $\mathrm{Cont}_{\mathbb{P}}$, we could allow all exceptions to be thrown freely—as explained above, which corresponds to a *partial correctness* interpretation—but a different choice is to prevent any exceptions from being raised at all—which corresponds to a *total correctness* interpretation. Similarly, for specifying nondeterministic computations, two interpretations are possible for $\mathrm{Cont}_{\mathbb{P}}$: a *demonic* one, in which the postcondition should hold for *all* possible result values [Dijkstra 1975], and an *angelic* one, in which the postcondition should hold for *at least one* possible result [Floyd 1967].

The key idea of this paper is to decouple the computational monad and the specification monad: instead of insisting on deriving both from the same monad transformer as in *DM4Free*, we consider

[1] Prior work has used the term "Dijkstra monad" both for the indexed structure $\mathcal{D}$ and for the index W [Ahman et al. 2017; Jacobs 2014, 2015; Swamy et al. 2013, 2016]. In order to prevent confusion, we use the term "Dijkstra monad" exclusively for the indexed structure $\mathcal{D}$ and the term "specification monad" for the index W.

them independently and only require that they are related by a *monad morphism*, i.e., a mapping between two monads that respects their monadic structure. For instance, a monad morphism from nondeterministic computations could map a finite set of possible outcomes to a predicate transformer in $(A \to \mathbb{P}) \to \mathbb{P}$. Given a finite set R of results in A and a postcondition $post : A \to \mathbb{P}$, there are only two reasonable ways to obtain a single proposition: either take the *conjunction* of $post\ v$ for every v in R (demonic nondeterminism), or the *disjunction* (angelic nondeterminism). For the case of IO, in our framework we can consider at least two monad morphisms relating the IO monad to two different specification monads, W^{Fr} and W^{Hist}, where $\mathcal{E}$ is the alphabet of IO events:

$$W^{\mathrm{Fr}}X = (X \times \mathcal{E}^* \to \mathbb{P}) \to \mathbb{P} \quad \longleftarrow \quad \mathrm{IO} \quad \longrightarrow \quad W^{\mathrm{Hist}}X = (X \times \mathcal{E}^* \to \mathbb{P}) \to (\mathcal{E}^* \to \mathbb{P})$$

While both specification monads take postconditions of the same type (predicates on the final value and the produced IO events), the produced precondition of $W^{\mathrm{Hist}}X$ has an additional argument $\mathcal{E}^*$, which denotes the history of of interactions (i.e., IO events) with the external world.

This paper makes the following **contributions**:

- We propose a new semantic framework for verifying programs with arbitrary monadic effects using Dijkstra monads. By decoupling the computational monad from the specification monad we remove all previous restrictions on supported computational monads. Moreover, this decoupling allows us to flexibly choose the specification monad and monad morphism most suitable for the verification task at hand. We investigate a large variety of specification monads that are obtained by applying monad transformers to various base monads, including predicate transformers (e.g., weakest preconditions and strongest postconditions) and Hoare-style pre- and postconditions. This flexibility allows a wide range of verification styles for nondeterminism, IO, and general recursion—none of which was possible with *DM4Free*.
- We give the first general definition of Dijkstra monads as a monad-like structure indexed by a specification monad ordered by precision. We show that any monad morphism gives rise to a Dijkstra monad, and that from any such Dijkstra monad we can recover the monad morphism. More generally, we construct an adjunction between Dijkstra monads and a generalization of monad morphisms, monadic relations, which induces the above-mentioned equivalence.
- We recast *DM4Free* as a special case of our new framework. For this, we introduce SM, a principled metalanguage for defining correct-by-construction monad transformers. The design of SM is inspired by DM and Moggi's monadic metalanguage, but it is parameterized by an arbitrary dependent type theory instead of a set of simple types. We show that under a natural linearity condition SM terms give rise to correct-by-construction monad transformers (satisfying all the usual laws) as well as canonical monadic relations, defined from a logical relation. This allows us to reap the benefits of the *DM4Free* construction when it works well (e.g., state, exceptions), and to explicitly provide monad morphisms when it does not (e.g., nondeterminism, IO).
- We give an account of Plotkin and Power's algebraic operations for Dijkstra monads. We show that a monad morphism equips both its specification monad and the corresponding Dijkstra monad with algebraic operations. We also start to investigate two approaches to effect handlers. The first approach, in which the specification of operations is induced by the handler, allows us to both provide a uniform treatment of *DM4Free*'s hand-rolled examples of exception handling, and subsume the prior work on weakest exceptional preconditions. However, this approach seems inherently limited to exceptions. The second approach, in which operations have to be given specifications upfront, enables us to also accommodate handlers for effects other than exceptions, for instance for general recursion, based on McBride's free monad technique.
- We illustrate the generality of our semantic framework by applying it to the verification of simple monadic programs in both Coq and F*.

Paper structure. We start by reviewing the use of monads in effectful programming and the closest related approaches for reasoning about such programs (§2). We then give a gentle overview of our approach through illustrative examples (§3). After this, we dive into the technical details: First, we show how to obtain a wide range of specification monads by applying monad transformers to base specification monads (§4). Then, we show the tight and natural correspondence between Dijkstra monads, and monadic relations and monad morphisms (§5). We also study algebraic operations and effect handlers for Dijkstra monads (§6). Finally, we outline our implementations of these ideas in F* and Coq (§7), before discussing related (§8) and future work (§9).

Supplementary materials include: (1) verification examples and implementation of our framework in F* (https://github.com/FStarLang/FStar/tree/dm4all/examples/dm4all); (2) verification examples and a formalization in Coq (https://gitlab.inria.fr/kmaillar/dijkstra-monads-for-all); (3) an online appendix with further technical details (https://arxiv.org/abs/1903.01237).

2 BACKGROUND: MONADS AND MONADIC REASONING

We start by briefly reviewing the use of monads in effectful programming, as well as the closest related approaches for verifying monadic programs.

2.1 The Monad Jungle Book

Side effects are an important part of programming. They arise in a multitude of shapes, be it imperative algorithms, nondeterministic operations, potentially diverging computations, or interactions with the external world. These various effects can be uniformly captured by the algebraic structure known as a *computational monad* [Benton et al. 2000; Moggi 1989]. This uniform interface is provided via a type MA of computations returning values of type A; a function $\mathtt{ret}^M : A \to MA$ that coerces a value $v : A$ to a trivial computation, for instance seeing v as a stateful computation leaving the state untouched; and a function $\mathtt{bind}^M\ m\ f$ that sequentially composes the monadic computations $m : MA$ with $f : A \to MB$, for instance threading through the state. Equations specify that $\mathtt{ret}^M$ does not have any computational effect, and that $\mathtt{bind}^M$ is associative.

The generic monad interface $(M, \mathtt{ret}^M, \mathtt{bind}^M)$ is, however, not enough to write programs that exploit the underlying effect. To this end, each computational monad also comes with *operations* for causing effects. We briefly recall a few examples of computational monads and their operations:

Exceptions: A computation that can potentially throw exceptions of type E can be represented by the monad $\mathrm{Exc}\,A = A + E$. Returning a value v is the obvious left injection, while sequencing m with f is given by applying f to v if $m = \mathtt{inl}\ v$, or $\mathtt{inr}\ e$ if $m = \mathtt{inr}\ e$, i.e., when m raised an exception. The operation $\mathtt{throw} : E \to \mathrm{Exc}\,\mathbb{0}$ is defined by right injection. When we take $E = \mathbb{1}$, exceptions also give us a simple model of partiality (the monad being $\mathrm{Div}\,A = A + \mathbb{1}$).

State: A stateful computation can be modeled as a state-passing function, i.e., $\mathrm{St}\,A = S \to A \times S$, where S is the type of the state. Returning a value v is the function $\lambda s.\ \langle v,\ s\rangle$ that produces the value v and the unmodified state, whereas binding m to f is obtained by threading through the state, i.e. $\lambda s.\mathbf{let}\ \langle v,\ s'\rangle = m\ s\ \mathbf{in}\ f\ v\ s'$. The state monad comes with operations $\mathtt{get} : \mathrm{St}\,S = \lambda s.\ \langle s,\ s\rangle$ to retrieve the state, and $\mathtt{put} : S \to \mathrm{St}\,\mathbb{1} = \lambda s.\lambda s'.\ \langle *,\ s\rangle$ to overwrite it.

Nondeterminism: A nondeterministic computation can be represented by a finite set of possible outcomes, i.e. $\mathrm{NDet}\,A = \mathcal{P}_{\mathsf{fin}}(A)$. Returning a value v is provided by the singleton $\{v\}$, whereas sequencing m with f amounts to forming the union $\bigcup_{v \in m} f\ v$. This monad comes with an operation $\mathtt{pick} : \mathrm{NDet}\,\mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$, which nondeterministically chooses a boolean value, and an operation $\mathtt{fail} : \mathrm{NDet}\,\mathbb{0} = \emptyset$, which unconditionally fails.

Interactive input-output (IO): An interactive computation with input type I and output type O can be represented by the inductively defined monad $\mathrm{IO}\,A = \mu Y.A + (I \to Y) + O \times Y$, which describes three possible kinds of computations: either return a value (A), expect to receive an

input and then continue ($I \to Y$), or output and continue ($O \times Y$). Returning v is constructing a leaf, whereas sequencing m with f amounts to tree grafting: replacing each leaf with value a in m with the tree $f\,a$. The operations for IO are $\texttt{input} : \mathrm{IO}\,I$ and $\texttt{output} : O \to \mathrm{IO}\,\mathbb{1}$.

2.2 Reasoning About Computational Monads

Many approaches have been proposed for reasoning about effectful programs; we review the ones closest to ours. In an imperative setting, Hoare introduced a *program logic* to reason about properties of programs [Hoare 1969]. The judgments of this logic are *Hoare triples* $\{\, pre \,\}\ c\ \{\, post \,\}$. Intuitively, if the precondition *pre* is satisfied, then running the program c leaves us in a situation where *post* is satisfied, provided that c terminates. For imperative programs—i.e., statements changing the program's state—*pre* and *post* are predicates over states.

Hoare's approach can be directly adapted to the monadic setting by replacing imperative programs c with monadic computations $m : M\,A$. This approach was first proposed in Hoare Type Theory [Nanevski et al. 2008b], where a *Hoare monad* of the form $\texttt{HST}\ pre\ A\ post$ augments the state monad over A with a precondition $pre : S \to \mathbb{P}$ and postcondition $post : A \times S \to \mathbb{P}$. So while preconditions are still predicates over initial states, postconditions are now predicates over both final states and results. While this approach was successfully extended to a few other effects [Delbianco and Nanevski 2013; Nanevski et al. 2008a, 2013], there is still no general story on how to define a Hoare monad or even just the shape of pre- and postconditions for an arbitrary effect.

A popular alternative to proving properties of imperative programs is Dijkstra's *weakest precondition calculus* [Dijkstra 1975]. The main insight of this calculus is that we can typically compute a weakest precondition $\mathrm{wp}(c, post)$ such that $pre \Rightarrow \mathrm{wp}(c, post)$ if and only if $\{\, pre \,\}\ c\ \{\, post \,\}$, and therefore partly automate the verification process by reducing it to a logical decision problem. Swamy et al. [2013] observed that it is possible to adopt Dijkstra's technique to ML programs with state and exceptions elaborated to monadic style. They propose a notion of *Dijkstra monad* of the form $\texttt{DST}\ A\ wp$, where *wp* is a predicate transformer that specifies the behavior of the monadic computation. These predicate transformers are represented as functions that, given a postcondition on the final state, and the result value of type A or an exception of type E, calculate a corresponding precondition on the initial state. Their predicate transformer type can be written as follows:

$$W^{ML} \quad = \quad \underbrace{((A + E) \times S \to \mathbb{P})}_{\textit{postconditions}} \to \underbrace{(S \to \mathbb{P})}_{\textit{preconditions}}.$$

In subsequent work, Swamy et al. [2016] extend this to programs that combine multiple sub-effects. They compute more efficient weakest preconditions by using Dijkstra monads that precisely capture the actual effects of the code, instead of verifying everything using W^{ML} above. For example, pure computations are verified using a Dijkstra monad whose specifications have type:

$$W^{\mathrm{Pure}} A \quad = \quad \mathrm{Cont}_{\mathbb{P}}\, A \quad = \quad (A \to \mathbb{P}) \to \mathbb{P},$$

while stateful (but exception-free) computations are verified using specifications of type:

$$W^{\mathrm{St}} A \quad = \quad (A \times S \to \mathbb{P}) \to (S \to \mathbb{P}).$$

Recently, Ahman et al.'s [2017] *DM4Free* work shows that these originally disparate specification monads can be uniformly derived from computational monads defined in their DM metalanguage.

An important observation underlying these techniques is that predicate transformers have a natural monadic structure. For instance, it is not hard to see that the predicate transformer type W^{Pure} is simply the continuation monad with answer type $\mathbb{P}$, that W^{St} is the state monad transformer applied to W^{Pure}, and that W^{ML} is the state and exceptions monad transformers applied to W^{Pure}. It is this monadic structure that supports writing computations that carry their own specification. In the next section, we show that it is also the basis for what we call a *specification monad*.

3 A GENTLE INTRODUCTION TO DIJKSTRA MONADS FOR ALL

In this section we introduce a few basic definitions and illustrate the main ideas of our semantic framework on various relatively simple examples. We start from the observation that the kinds of specifications most commonly used in practice form *ordered monads* (§3.1). On top of this we define *effect observations*, as just monad morphisms between a computation and a specification monad (§3.2), and give various examples (§3.3). Finally, we explain how to use effect observations to obtain Dijkstra monads, and how to use Dijkstra monads for program verification (§3.4).

3.1 Specification Monads

The realization that predicate transformers form monads [Ahman et al. 2017; Jacobs 2014, 2015; Swamy et al. 2013, 2016] is the starting point to provide a uniform notion of specifications. Generalizing over prior work, we show that this is true not only for weakest precondition transformers, but also for strongest postconditions, and pairs of pre- and postconditions (see §4.1). Intuitively, elements of a specification monad can be used to specify properties of some computation, e.g., W^{Pure} can specify pure or nondeterministic computations, and W^{St} can specify stateful computations.

The specification monads we consider are *ordered*. Formally, a monad W is ordered when WA is equipped with a preorder $\leq^{WA}$ for each type A, and $\mathtt{bind}^W$ is monotonic in both arguments:
$\forall (w_1 \leq^{WA} w_1'). \forall (w_2\ w_2' : A \to WB). (\forall x : A.\ w_2\ x \leq^{WB} w_2'\ x) \Rightarrow \mathtt{bind}^W\ w_1\ w_2 \leq^{WB} \mathtt{bind}^W\ w_1'\ w_2'$
This order allows specifications to be compared as being either more or less precise. For example, for the specification monads W^{Pure} and W^{St}, the ordering is given by

$$w_1 \leq w_2 : W^{\mathrm{Pure}} A \quad \Leftrightarrow \quad \forall (p : A \to \mathbb{P}).\ w_2\ p \Rightarrow w_1\ p$$
$$w_1 \leq w_2 : W^{\mathrm{St}} A \quad \Leftrightarrow \quad \forall (p : A \times S \to \mathbb{P})(s : S).\ w_2\ p\ s \Rightarrow w_1\ p\ s$$

For W^{Pure} and W^{St} to form ordered monads, it turns out that we need to restrict our attention to *monotonic* predicate transformers, i.e., those mapping (pointwise) stronger postconditions to stronger preconditions. This technical condition, quite natural from the point of view of verification, will be assumed implicitly for all the predicate transformers, and will be studied in detail in §4.1.

As explained in §2.2, a powerful way to construct specification monads is to apply monad transformers to existing specification monads, e.g., applying $\mathrm{ExcT}\ M\ A = M\ (A + E)$ to W^{Pure} we get

$$W^{\mathrm{Exc}} A \quad = \quad \mathrm{ExcT}\ W^{\mathrm{Pure}}\ A \quad = \quad ((A + E) \to \mathbb{P}) \to \mathbb{P} \quad \cong \quad (A \to \mathbb{P}) \to (E \to \mathbb{P}) \to \mathbb{P}$$

W^{Exc} is a natural specification monad for programs that can throw exceptions, transporting a normal postcondition in $A \to \mathbb{P}$ and an exceptional postcondition in $E \to \mathbb{P}$ to a precondition in $\mathbb{P}$. Further specification monads using this idea will be introduced along with the examples in §3.3.

3.2 Effect Observations

Now that we have a presentation of specifications as elements of a monad, we can relate computational monads to such specifications. Since an object relating computations to specifications provides a particular insight to the potential effects of the computation, they have been called *effect observations* [Katsumata 2014]. As explained in §1, a computational monad can have effect observations into multiple specification monads, or multiple effect observations into a single specification monad. Using the exceptions computational monad Exc as running example, we argue that *monad morphisms* provide a natural notion of effect observation in our setting, and we provide example monad morphisms supporting this claim. Further examples are explored in §3.3.

Effect observations are monad morphisms. As explained in §2.1, computations throwing exceptions can be modeled by monadic expressions $m : \mathrm{Exc}\ A = A + E$. A natural way to specify m is to consider the specification monad $W^{\mathrm{Exc}} A = ((A + E) \to \mathbb{P}) \to \mathbb{P}$ and to map m to the predicate transformer $\theta^{\mathrm{Exc}}(m) = \lambda p.\ p\ m : W^{\mathrm{Exc}} A$, applying the postcondition p to the computation m.

The mapping $\theta^{\mathrm{Exc}} : \mathrm{Exc} \to W^{\mathrm{Exc}}$ relating the computational monad Exc and the specification monad W^{Exc} is parametric in the return type A, and it verifies two important properties with respect to the monadic structures of Exc and W^{Exc}. First, a returned value is specified by itself:

$$\theta^{\mathrm{Exc}}(\mathtt{ret}^{\mathrm{Exc}}\, v) = \theta^{\mathrm{Exc}}(\mathtt{inl}\, v) = \lambda p.\, p\,(\mathtt{inl}\, v) = \mathtt{ret}^{W^{\mathrm{Exc}}}\, v$$

and second, θ preserves the sequencing of computations:

$$\theta^{\mathrm{Exc}}(\mathtt{bind}^{\mathrm{Exc}}\ (\mathtt{inl}\, v)\ f) = \theta^{\mathrm{Exc}}(f v) = \mathtt{bind}^{W^{\mathrm{Exc}}}(\mathtt{ret}^{W^{\mathrm{Exc}}} v)\ (\theta^{\mathrm{Exc}} \circ f) = \mathtt{bind}^{W^{\mathrm{Exc}}}\ \theta^{\mathrm{Exc}}(\mathtt{inl}\, v)\ (\theta^{\mathrm{Exc}} \circ f)$$

$$\theta^{\mathrm{Exc}}(\mathtt{bind}^{\mathrm{Exc}}\ (\mathtt{inr}\, e)\ f) = \theta^{\mathrm{Exc}}(\mathtt{inr}\, e) = \mathtt{bind}^{W^{\mathrm{Exc}}}\ \theta^{\mathrm{Exc}}(\mathtt{inr}\, e)\ (\theta^{\mathrm{Exc}} \circ f)$$

These properties together prove that θ^{Exc} is a monad morphism. More importantly, they allow us to compute specifications from computations *compositionally*, e.g., the specification of `bind` can be computed from the specifications of its arguments. This leads us to the following definition:

Definition 1 (Effect observation). *An* effect observation θ *is a monad morphism from a computational monad M to a specification monad W. More explicitly, it is a family of maps $\theta_A : M\,A \longrightarrow W\,A$, natural in A and such that for any $v : A$, $m : M\,A$ and $f : A \to M\,B$ the following equations hold:*

$$\theta_A(\mathtt{ret}^M\, v) = \mathtt{ret}^W\, v \qquad\qquad \theta_B(\mathtt{bind}^M\, m\, f) = \mathtt{bind}^W\, (\theta_A\, m)\, (\theta_B \circ f)$$

Specification monads are not canonical. When writing programs using the exception monad, we may want to write pure sub-programs that actually do not raise exceptions. In order to make sure that these sub-programs are pure, we could use the previous specification monad and restrict ourselves to postconditions that map exceptions to false ($\bot$): hence raising an exception would have an unsatisfiable precondition. However, as outlined in §1, a simpler solution is possible. Taking as specification monad W^{Pure}, we can define the following effect observation $\theta^{\bot} : \mathrm{Exc} \to W^{\mathrm{Pure}}$ by

$$\theta^{\bot}(\mathtt{inl}\, v) = \lambda p.\, p\, v \qquad\qquad \theta^{\bot}(\mathtt{inr}\, e) = \lambda p.\, \bot$$

This effect observation gives a *total correctness* interpretation to exceptions, which prevents them from being raised at all. As such, we have effect observations from Exc to both W^{Exc} and W^{Pure}.

Effect observations are not canonical. Looking closely at the effect observation $\theta^{\bot}$, it is clear that we made a rather arbitrary choice when mapping every exception $\mathtt{inr}\, e$ to $\bot$. Mapping $\mathtt{inr}\, e$ to true ($\top$) instead also gives us an effect observation, $\theta^{\top} : \mathrm{Exc} \to W^{\mathrm{Pure}}$. This effect observation assigns a trivial precondition to the `throw` operation, providing a *partial correctness* interpretation: given a program $m : \mathrm{Exc}\, A$ and a postcondition $p : A \to \mathbb{P}$, if $\theta^{\top}(m)(p)$ is satisfiable and m evaluates to $\mathtt{inl}\, v$ then $p\, v$ holds; but m may also raise any exception instead. Thus, $\theta^{\bot}, \theta^{\top} : \mathrm{Exc} \to W^{\mathrm{Pure}}$ are two natural effect observations into the *same* specification monad. Even more generally, we can vary the choice for each exception; in fact, effect observations $\theta : \mathrm{Exc} \to W^{\mathrm{Pure}}$ are in one-to-one correspondence with maps $E \to \mathbb{P}$ (see §4.4 for a general account of this correspondence).

3.3 Examples of Effect Observations

When specifying and verifying monadic programs, there is generally a large variety of options regarding both the specification monads and the effect observations. We will now revisit more computational monads from §2.1, and present various natural effect observations for them.

Monad transformers. Even though there is, in general, no canonical effect observation for a computational monad, for the case of a monad $\mathcal{T}(\mathrm{Id})$ (i.e., a monad obtained by the application of a monad transformer to the identity monad) we can build a canonical specification monad, namely $\mathcal{T}(W^{\mathrm{Pure}})$, and a canonical effect observation into it. The effect observation is obtained simply by lifting the $\mathtt{ret}^{W^{\mathrm{Pure}}} : \mathrm{Id} \to W^{\mathrm{Pure}}$ function through the $\mathcal{T}$ transformer. This is the main idea behind our reinterpretation of the *DM4Free* approach [Ahman et al. 2017]. For instance, for the exception monad $\mathrm{Exc} = \mathrm{ExcT}(\mathrm{Id})$ and the specification monad $W^{\mathrm{Exc}} = \mathrm{ExcT}\,(W^{\mathrm{Pure}})$, the effect

observation θ^{Exc} arises as simply $\theta^{\mathrm{Exc}} = \mathrm{ExcT}(\mathtt{ret}^{W^{\mathrm{Pure}}}) = \lambda m\, p.\ p\, m$. More generally, for any monad transformer $\mathcal{T}$ (e.g. StT, ExcT, StT $\circ$ ExcT, ExcT $\circ$ StT) and any specification monad W (so not just W^{Pure}, but also e.g., any basic specification monad from §4.1) we have a monad morphism

$$\theta^{\mathcal{T}} \quad : \quad \mathcal{T}(\mathrm{Id}) \quad \xrightarrow{\mathcal{T}(\mathtt{ret}^{W^{\mathrm{Pure}}})} \quad \mathcal{T}(W^{\mathrm{Pure}})$$

providing effect observations for stateful computations with exceptions, or for computations with rollback state. However, not all computational monads arise as a monad transformer applied to the identity monad. The following examples illustrate the possibilities in such cases.

Nondeterminism. The computational monad NDet admits effect observations to the specification monad W^{Pure}. Given a nondeterministic computation $m : \mathrm{NDet}\, A$ represented as a finite set of possible outcomes, and a postcondition $post : A \to \mathbb{P}$, we obtain a set P of propositions by applying $post$ to each element of m. There are then two natural ways to interpret P as a single proposition:

- ▷ we can take the conjunction $\bigwedge_{p \in P} p$, which corresponds to the weakest precondition such that *any* outcome of m satisfies $post$ (*demonic nondeterminism*); or
- ▷ we can take the disjunction $\bigvee_{p \in P} p$, which corresponds to the weakest precondition such that *at least one* outcome of m satisfies $post$ (*angelic nondeterminism*).

To see that both these choices lead to monad morphisms $\theta^{\forall}, \theta^{\exists} : \mathrm{NDet} \to W^{\mathrm{Pure}}$, it is enough to check that taking the conjunction when $P = \{p\}$ is a singleton is equivalent to p, and that a conjunction of conjunctions $\bigwedge_{a \in A} \bigwedge_{p \in P_a} p$ is equivalent to a conjunction on the union of the ranges $\bigwedge_{p \in \bigcup_{a \in A} P_a} p$—and similarly for disjunctions. Both conditions are straightforward to check.

Interactive Input-Output. Let us now consider programs in the IO monad (§2.1). We want to define an effect observation $\theta : \mathrm{IO} \to W$, for some specification monad W to be determined. A first thing to note is that since no equations constrain the `input` and `output` operations, we can specify their interpretations $\theta(\mathtt{input}) : W\, I$ and $\forall(o : O).\ \theta(\mathtt{output}\, o) : W\, \mathbb{1}$ separately from each other.

Simple effect observations for IO can already be provided using the specification monad W^{Pure}. The interpretation of the `output` operation in this simple case needs to provide a result in $\mathbb{P}$ from an output element $o : O$ and a postcondition $p : \mathbb{1} \to \mathbb{P}$. Besides returning a constant proposition (like for $\theta^{\bot}, \theta^{\top}$ in §3.2), a reasonable interpretation is to forget the `output` operation and return $p\, *$ (where $*$ is the unit value). For the definition of $\theta(\mathtt{input}) : (I \to \mathbb{P}) \to \mathbb{P}$, we are given a postcondition $post : I \to \mathbb{P}$ on the possible inputs and we need to build a proposition. Two canonical solutions are to use either the universal quantification $\forall(i : I).\ post\ i$, requiring that the postcondition is valid for the continuation of the program for any possible input; or the existential quantification $\exists(i : I).\ post\ i$, meaning that there exists some input such that the program's continuation satisfies the postcondition, analogously to the two modalities of evaluation logic [Moggi 1995; Pitts 1991].

To get more interesting effect observations accounting for inputs and outputs we can, for instance, extend W^{Pure} with *ghost state* [Owicki and Gries 1976] capturing the list of executed IO events.[2] We can do this by applying the state monad transformer with state type $\mathrm{list}\, \mathcal{E}$ to W^{Pure}, obtaining the specification monad $W^{\mathrm{HistST}}\, A = (A \times \mathrm{list}\, \mathcal{E} \to \mathbb{P}) \to \mathrm{list}\, \mathcal{E} \to \mathbb{P}$, for which we can provide interpretations of `input` and `output` that also keep track of the history of events via ghost state:

$$\theta^{\mathrm{HistST}}(\mathtt{output}\, o) \quad = \quad \lambda(p : \mathbb{1} \times \mathrm{list}\, \mathcal{E} \to \mathbb{P})\, (log : \mathrm{list}\, \mathcal{E}).\ p\, \langle *, (\mathrm{Out}\, o) :: log \rangle \qquad : W^{\mathrm{HistST}}(\mathbb{1})$$

$$\theta^{\mathrm{HistST}}(\mathtt{input}) \quad = \quad \lambda(p : I \times \mathrm{list}\, \mathcal{E} \to \mathbb{P})\, (log : \mathrm{list}\, \mathcal{E}).\ \forall i.\ p\, \langle i, (\mathrm{In}\, i) :: log \rangle \qquad : W^{\mathrm{HistST}}(I)$$

This specification monad is however somewhat inconvenient in that postconditions are written over the *global* history of events, instead of over the events of the expression in question. Further, one can write specifications that "shrink" the global history of events, such as $\lambda p\, log.\ p\, \langle *, [] \rangle$, which *no* expression satisfies. For these reasons, we introduce an *update monad* [Ahman and Uustalu

[2] Importantly, the ghost state only appears in specifications and not in user programs; these still use only (stateless) IO.

2013] variant of W^{HistST}, written W^{Hist}, which provides a more concise way to describe the events. In particular, in W^{Hist} the postcondition specifies only the events produced by the expression, while the precondition is still free to specify any previously-produced events, allowing us to define:

$$\theta^{\text{Hist}}(\texttt{output}\ o) \quad = \quad \lambda(p : \mathbb{1} \times \text{list}\,\mathcal{E} \to \mathbb{P})\,(log : \text{list}\,\mathcal{E}).\ p\,\langle *, [\text{Out}\,o]\rangle \qquad : W^{\text{Hist}}(\mathbb{1})$$

$$\theta^{\text{Hist}}(\texttt{input}) \quad = \quad \lambda(p : I \times \text{list}\,\mathcal{E} \to \mathbb{P})\,(log : \text{list}\,\mathcal{E}).\ \forall i.\, p\,\langle i, [\text{In}\,i]\rangle \qquad : W^{\text{Hist}}(I)$$

While $W^{\text{Hist}} = W^{\text{HistST}}$, the two monads differ in their `ret` and `bind` functions. For instance,

$$\texttt{bind}^{W^{\text{HistST}}}\ w\ f \quad = \quad \lambda p\ log.\ w\,\big(\lambda\,\langle x,\ log'\rangle\ .\ f\ x\ p\ log'\big)\ log$$

$$\texttt{bind}^{W^{\text{Hist}}}\ w\ f \quad = \quad \lambda p\ log.\ w\,\big(\lambda\,\langle x,\ log'\rangle\ .\ f\ x\ \big(\lambda(y, log'').\ p\,\langle y, log' +\!\!+ log''\rangle\big)\ (log +\!\!+ log')\big)\ log$$

where the former overwrites the history, while the latter merely augments it with new events.

While W^{Hist} provides a good way to reason about IO, some IO programs do not depend on past interactions. For these, we can provide an even more parsimonious specification monad by applying the writer transformer to W^{Pure}. The resulting specification monad W^{Fr} then allows us to define

$$\theta^{\text{Fr}}(\texttt{output}\ o) \quad = \quad \lambda(p : \mathbb{1} \times \text{list}\,\mathcal{E} \to \mathbb{P}).\ p\,\langle *, [\text{Out}\,o]\rangle \quad : \quad W^{\text{Fr}}(\mathbb{1})$$

$$\theta^{\text{Fr}}(\texttt{input}) \quad = \quad \lambda(p : I \times \text{list}\,\mathcal{E} \to \mathbb{P}).\ \forall i.\, p\,\langle i, [\text{In}\,i]\rangle \quad : \quad W^{\text{Fr}}(I)$$

This is in fact a special case of W^{Hist} where the history is taken to be $\mathbb{1}$ [Ahman and Uustalu 2013].

In fact, there is even more variety possible here, e.g., it is straightforward to write specifications that speak only of output events and not input events, and vice versa. It is also easy to extend this style of reasoning to combinations of IO and other effects. For instance, we can simultaneously reason about state changes and IO events by considering computations in $\text{IOSt}\,A = S \to \text{IO}(A \times S)$, resulting from applying the state monad transformer to IO, together with the specification monad $W^{\text{IOSt}}\,A = (A \times S \times \text{list}\,\mathcal{E} \to \mathbb{P}) \to S \to \text{list}\,\mathcal{E} \to \mathbb{P}$. As such, we recover the style proposed by Malecha et al. [2011], though they also cover separation logic, which we leave as future work.

Being able to choose between specification monads and effect observations allows one to keep the complexity of the specifications low when the properties are simple, yet increase it if required.

3.4 Recovering Dijkstra Monads

We now return to Dijkstra monads, which provide a practical and automatable verification technique in dependent type theories like F* [Swamy et al. 2016], where they are a primitive notion, and Coq, where they can be embedded via dependent types. We explain how a Dijkstra monad can be obtained from a computational monad, a specification monad, and an effect observation relating them. Then we show how the obtained Dijkstra monad can be used for actual verification.

Stateful computations. Let us start with stateful computations as an illustrative example, taking the computational monad St, the specification monad W^{St}, and the following effect observation:

$$\theta^{\text{St}} \quad : \quad \text{St} \to W^{\text{St}}$$

$$\theta^{\text{St}}(m) \quad = \quad \lambda post\ s_0.\ post\ (m\ s_0)$$

We begin by defining the Dijkstra monad type constructor, $\text{ST} : (A : \text{Type}) \to W^{\text{St}}\ A \to \text{Type}$. The type $\text{ST}\ A\ w$ contains all those computations $c : \text{St}\ A$ that are correctly specified by w. We say that c is *correctly specified* by w when $\theta^{\text{St}}(c) \leq w$, that is, when w is weaker than (or equal to) the specification given from the effect observation. Unfolding the definitions of $\leq$ and θ^{St}, this intuitively says that for any initial state s_0 and postcondition $post : A \times S \to \mathbb{P}$, the precondition $w\ post\ s_0$ computed by w is enough to ensure that c returns a value $v : A$ and a final state s_1 satisfying $post\ \langle v,\ s_1\rangle$; in other words, $w\ post\ s_0$ implies the weakest precondition of c.

The concrete definition for the type of a Dijkstra monad can vary according to the type theory in question. For instance, in our Coq development, we define it (roughly) as a dependent pair of a

computation $c : \mathrm{St}\ A$ and a proof that c is correctly specified by w. In F*, it is instead a primitive notion. In the rest of this section, we shall not delve into such representation details.

The Dijkstra monad ST is equipped with monad-like functions $\mathtt{ret}^{\mathrm{ST}}$ and $\mathtt{bind}^{\mathrm{ST}}$ whose definitions come from the computational monad St, while their specifications come from the specification monad W^{St}. The general shape for the `ret` and `bind` of the obtained Dijkstra monad is:[3]

$$\begin{array}{rcl} \mathtt{ret}^{\mathrm{ST}} & = \mathtt{ret}^{\mathrm{St}} & : \ (v : A) \to \mathrm{ST}\ A\ (\mathtt{ret}^{W^{\mathrm{St}}}\ v) \\ \mathtt{bind}^{\mathrm{ST}} & = \mathtt{bind}^{\mathrm{St}} & : \ (c : \mathrm{ST}\ A\ w_c) \to (f : (x : A) \to \mathrm{ST}\ B\ (w_f\ x)) \to \mathrm{ST}\ B\ (\mathtt{bind}^{W^{\mathrm{St}}}\ w_c\ w_f) \end{array}$$

which, after unfolding the state-specific definitions becomes:

$$\begin{array}{rcll} \mathtt{ret}^{\mathrm{ST}} & = \mathtt{ret}^{\mathrm{St}} & : & (v : A) \to \mathrm{ST}\ A\ (\lambda post\ s_0.\ post\ \langle v,\ s_0\rangle) \\ \mathtt{bind}^{\mathrm{ST}} & = \mathtt{bind}^{\mathrm{St}} & : & (c : \mathrm{ST}\ A\ w_c) \to (f : (x : A) \to \mathrm{ST}\ B\ (w_f\ x)) \\ & & & \to \mathrm{ST}\ B\ (\lambda p\ s_0.\ w_c\ (\lambda\ \langle x,\ s_1\rangle\ .\ w_f\ x\ p\ s_1)\ s_0) \end{array}$$

The operations of the computational monad are also reflected into the Dijkstra monad, with their specifications are computed by θ^{St}. Given $op^{\mathrm{St}} : (x_1 : A_1) \to \cdots \to (x_n : A_n) \to \mathrm{St}\ B$, we can define

$$op^{\mathrm{ST}} \quad = \quad op^{\mathrm{St}} \quad : \quad (x_1 : A_1) \to \cdots \to (x_n : A_n) \to \mathrm{ST}\ B\ (\theta^{\mathrm{St}}(op^{\mathrm{St}}\ x_1\ \ldots\ x_n))$$

Concretely, for state, we get the following two operations for the Dijkstra monad ST:

$$\mathtt{get} \quad : \quad \mathrm{ST}\ S\ (\lambda p\ s_0.\ p\ \langle s_0,\ s_0\rangle), \qquad \mathtt{put} \quad : \quad (s : S) \to \mathrm{ST}\ \mathbb{1}\ (\lambda p\ s_0.\ p\ \langle *,\ s\rangle).$$

Given this refined version of the state monad, computing specifications of (non-recursive) programs becomes simply a matter of doing *type inference* to compositionally lift the program to a specification and then unfolding the specification by (type-level) computation. For instance, given $\mathtt{modify}\ (f : S \to S) = \mathtt{bind}^{\mathrm{ST}}\ \mathtt{get}\ (\lambda x.\ \mathtt{put}(f x))$, both F* and Coq can infer the type

$$\mathrm{ST}\ \mathbb{1}\ (\mathtt{bind}^{W^{\mathrm{St}}}\ (\lambda p\ s_0.\ p\ \langle s_0,\ s_0\rangle)\ (\lambda s\ p\ s_0.\ p\ \langle *,\ f s\rangle)) \quad = \quad \mathrm{ST}\ \mathbb{1}\ (\lambda p\ s_0.\ p\ \langle *,\ f\ s_0\rangle)$$

which precisely describes the behavior of `modify`. Program verification then amounts to proving that, given a programmer-provided type-annotation $\mathrm{ST}\ \mathbb{1}\ w$ for $\mathtt{modify} f$, the specification w is weaker than the inferred specification.

Demonic nondeterminism. The previous construction is independent from how the computational monad, the specification monad, and the effect observation were obtained. The exact same approach can be followed for the NDet monad coupled with any of its effect observations. We use the demonic one here, for which the `pick` and `fail` actions for the Dijkstra monad have types:

$$\mathtt{pick}^{\mathrm{ND}_{\text{✠}}} \quad : \quad \mathrm{ND}_{\text{✠}}\ \mathbb{B}\ (\lambda p.\ p\ \mathtt{true} \wedge p\ \mathtt{false}) \qquad \mathtt{fail}^{\mathrm{ND}_{\text{✠}}} \quad : \quad \mathrm{ND}_{\text{✠}}\ \mathbb{0}\ (\lambda p.\ \top)$$

With this, we can define and verify F* (or Coq) functions like the following:

```
let rec pickl (l:list α) : NDD α (λ p → ∀x. elem x l ⟹ p x) =
  match l with | [] → fail () | x::xs → if pick () then x else pickl xs
let guard (b:bool) : NDD unit (λ p → b ⟹ p ()) = if b then () else fail ()
```

The `pickl` function nondeterministically chooses an element from a list, guaranteeing in its specification that the chosen value belongs to it. The `guard` function checks that a given boolean condition holds, failing otherwise. The specification of `guard b` ensures that `b` is true in the continuation. Using these two functions, we can write and verify concise nondeterministic programs, such as the one below that computes Pythagorean triples. The specification simply says that every result (if any!) is a Pythagorean triple, while in the implementation we have some concrete bounds for the search:

```
let pyths () : NDD (int & int & int) (λ p → ∀x y z. x*x + y*y = z*z ⟹ p (x,y,z)) =
  let l = [1;2;3;4;5;6;7;8;9;10] in let (x,y,z) = (pickl l, pickl l, pickl l) in guard (x*x + y*y = z*z); (x,y,z)
```

[3] If the representation of the Dijkstra monad is dependent pairs, then the code here does not typecheck as-is and requires some tweaking. For this section we will assume Dijkstra monads are defined as *refinements* of the computational monad, without any explicit proof terms to carry around. In our Coq implementation we use `Program` and evars to hide such details.

Input-Output. We illustrate Dijkstra monads for multiple effect observations from IO. First, we consider the context-free interpretation $\theta^{\mathrm{Fr}} : \mathrm{IO} \longrightarrow W^{\mathrm{Fr}}$, for which IO operations have the interface:

$$\begin{aligned} \mathtt{input}^{\mathrm{IO}^{\mathrm{Fr}}} &: \ \mathrm{IOFree}\ I\ (\lambda p.\ \forall (i : I).\ p\ \langle i, [\mathrm{In}\ i]\rangle) \\ \mathtt{output}^{\mathrm{IO}^{\mathrm{Fr}}} &: \ (o : O) \longrightarrow \mathrm{IOFree}\ \mathbb{1}\ (\lambda p.\ p\ \langle *, [\mathrm{Out}\ o]\rangle) \end{aligned}$$

We can define and specify a program that duplicates its input (assuming an implicit coercion $I <: O$):

```
let duplicate () : IOFree unit (λ p → ∀x. p ((), [In x; Out x; Out x])) = let x=input() in output x; output x
```

However, with this specification monad, we cannot reason about the history of *previous* IO events. To overcome this issue, we can switch the specification monad to W^{Hist} and obtain

$$\begin{aligned} \mathtt{input}^{\mathrm{IO}^{\mathrm{Hist}}} &: \ \mathrm{IOHist}\ I\ (\lambda p\, h.\ \forall i.\ p\ \langle i, [\mathrm{In}\ i]\rangle) \\ \mathtt{output}^{\mathrm{IO}^{\mathrm{Hist}}} &: \ (o : O) \longrightarrow \mathrm{IOHist}\ \mathbb{1}\ (\lambda p\, h.\ p\ \langle *, [\mathrm{Out}\ o]\rangle) \end{aligned}$$

The computational part of this Dijkstra monad fully coincides with that of $\mathrm{IO}^{\mathrm{Fr}}$, but the specifications are much richer. For instance, we can define the following computation:

$$\mathtt{mustHaveOccurred} \ = \ \lambda\_.\ \mathtt{ret}^{\mathrm{IO}^{\mathrm{Hist}}}\ * \ : \ (o : O) \longrightarrow \mathrm{IOHist}\ \mathbb{1}\ (\lambda p\, h.\ \mathrm{Out}\ o \in h \wedge p\ \langle *, []\rangle)$$

which has no computational effect, yet requires that a given value o was already been outputted before it is called. This is *weakening* the specification of $\mathtt{ret}^{\mathrm{IO}^{\mathrm{Hist}}}\ *$ (namely, $\mathtt{ret}^{W^{\mathrm{Hist}}}\ * = \lambda p\, h.\ p\ \langle *, []\rangle$) to have a stronger precondition. By having this amount of access to the history, one can verify that certain invariants are respected. For instance, the following program will verify successfully:

```
let print_increasing (i:int) : IOHist unit (λ p h → ∀h'. p ((), h')) =
    output i; (* pure computation *) mustHaveOccurred i; (* another pure computation *) output (i+1)
```

The program has a "trivial" specification: it does not guarantee anything about the trace of events, nor does it put restrictions on the previous log. However, internally, the call to `mustHaveOccurred` has a precondition that `i` was already output, which can be proven from the postcondition of `output i`. If this `output` is removed, the program will (rightfully) fail to verify.

Finally, when considering the specification monad W^{IOSt}, we have both state and IO operations:

$$\begin{aligned} \mathtt{input}^{\mathrm{IOSt}} &: \mathrm{IOSt}\ I\ (\lambda p\, s\, h.\ \forall i.\ p\ \langle i, s, [\mathrm{In}\ i]\rangle) & \mathtt{get}^{\mathrm{IOSt}} &: \mathrm{IOSt}\ S\ (\lambda p\, s\, h.\ p\ \langle s, s, []\rangle) \\ \mathtt{output}^{\mathrm{IOSt}} &: (o : O) \longrightarrow \mathrm{IOSt}\ \mathbb{1}\ (\lambda p\, s\, h.\ p\ \langle *, s, [\mathrm{Out}\ o]\rangle) & \mathtt{put}^{\mathrm{IOSt}} &: (s : S) \longrightarrow \mathrm{IOSt}\ \mathbb{1}\ (\lambda p\, \_\, h.\ p\ \langle *, s, []\rangle) \end{aligned}$$

where $(\mathtt{input}^{\mathrm{IOSt}}, \mathtt{output}^{\mathrm{IOSt}})$ keep state unchanged, and $(\mathtt{get}^{\mathrm{IOSt}}, \mathtt{put}^{\mathrm{IOSt}})$ do not perform any IO. With this, we can write and verify programs that combine state and IO in non-trivial ways, e.g.,

```
let do_io_then_rollback_state () : IOST unit (λ s h p → ∀i . p (() , s , [In i; Out (s+i+1)])) =
  let x = get () in let y = input () in put (x+y); (* pure computation *) let z = get () in output (z+1); put x
```

The program mutates the state in order to compute output from input, possibly interleaved with pure computations, but eventually rolls it back to its initial value, as mandated by its specification.

Effect polymorphic functions. Even though the operations `ret` and `bind` provided by a (strong) monad can seem somewhat restrictive at first, they still allow us to write functions that are generic in the underlying computational monad. One example is the following `mapW` function on lists, generic in the monad `W` (similar to the `mapM` function in Haskell):

```
let rec mapW (l : list α) (f : α→ W β) : W (list β) =
  match l with [] → ret [] | x :: xs → bind (f x) (λ y → bind (mapW xs f) (λ ys → ret (y :: ys)))
```

When working with Dijkstra monads, we can use the `mapW` function as a generic specification for the same computation when expressed using an arbitrary Dijkstra monad `D` indexed by `W`:[4]

[4]These last examples are written in F⋆ syntax, but only implemented in Coq, since Dijkstra monads are not first class in F⋆.

```
let rec mapD (l : list α) (w : α→ W β) (f : (a:α) → D β(w a)) : D (list β) (mapW l w) =
  match l with [] → ret [] | x :: xs → let y = f x in let ys = mapD xs w f in y :: ys
```

where mapD takes the list l, the specification for what is to happen to each element of the list, w, and an implementation of that specification, f. It builds an effectful computation that produces a list, specified by the extension of the element-wise specification w to the whole list by mapW.

Analogously, we can implement a generic iterator combinator provided we have an invariant w : W unit for the loop body : nat → D unit w such that the invariant satisfies bind w (λ() → w) ≤ w:

```
let rec for_in (range : list nat) (body : nat → D unit w) : D unit w =
  match range with [] → () | i :: range → body i ; for_in range body
```

Here we use not only the monadic operations but also the possibility to weaken the specification bind w (λ() → w) computed from the second branch of the match to the specification w by assumption.

In all the examples in this section, we used Dijkstra monads obtained via the same general recipe (see §5 for details) from the same kinds of ingredients: a computational monad, a specification monad, and an effect observation from the former to the latter. This enables a uniform treatment of effects for verification, and opens the door for verifying rich properties of effectful programs.

4 DEFINING SPECIFICATION MONADS

To enable various verification styles, in §3 we introduced various specification monads arising from the application of monad transformers to the monad of predicate transformers W^{Pure}. In this section, we start by observing that W^{Pure} is not the only natural basic specification monad on which to stack monad transformers (§4.1). We then present our *specification metalanguage* SM, as a means for defining correct-by-construction monad transformers (§4.2). SM is a more principled variant of the DM language of Ahman et al. [2017], and similarly to DM, we give SM a semantics based on logical relations. Observing that not all SM terms give rise to monad transformers (§4.3), we extract conditions under which we are guaranteed to obtain monad transformers, providing an explanation for the somewhat artificial syntactic restrictions in DM. Finally, we also discuss a principled way to derive effect observations into W^{Pure} and W^{St} from algebras of computational monads (§4.4).

4.1 Basic Specification Monads

We consider several basic specification monads, whose relationship is summarized by Figure 1.

Predicate monad. Arguably the simplest way to specify a computation is to provide a postcondition on its outcomes. This can be done by considering the specification monad $\mathcal{P}\mathrm{red}\, A = A \to \mathbb{P}$ (the covariant powerset monad) with order $p_1 \leq^{\mathcal{P}\mathrm{red}} p_2 \iff \forall (a : A).\, p_1\, a \Rightarrow p_2\, a$. To specify the behavior of returning values, we can always map a value $v : A$ to the singleton predicate $\mathtt{ret}^{\mathcal{P}\mathrm{red}}\, v = \lambda y.\, (y = v) : \mathcal{P}\mathrm{red}\, A$. And given a predicate $p : \mathcal{P}\mathrm{red}\, A$ and a function $f : A \to \mathcal{P}\mathrm{red}\, B$, the predicate on B defined by $\mathtt{bind}^{\mathcal{P}\mathrm{red}}\, p\, f = \lambda b.\, \exists a.\, p\, a \wedge f\, a\, b$ specifies the behavior of sequencing two computations, where the first computation produces a value a satisfying p and, under this assumption, the second computation produces a value satisfying $f\, a$. While a specification $p : \mathcal{P}\mathrm{red}\, A$ provides information on the outcome of the computation, it cannot require preconditions, so computations need to be defined independently of any logical context. To give total correctness specifications to computations with non-trivial preconditions, for instance specifying that the division function div x y requires y to be non-zero, we need more expressive specification monads.

Pre-/postcondition monad. One more expressive specification monad is the monad of pre- and postconditions $\mathcal{P}\mathrm{re}\mathcal{P}\mathrm{ost}\, A = \mathbb{P} \times (A \to \mathbb{P})$, bundling a precondition together with a postcondition. Here the behavior of returning a value $v : A$ is specified by requiring a trivial precondition and ensuring as above a singleton postcondition: $\mathtt{ret}^{\mathcal{P}\mathrm{re}\mathcal{P}\mathrm{ost}}\, v = \langle \top,\ \lambda a.\, a = v \rangle : \mathcal{P}\mathrm{re}\mathcal{P}\mathrm{ost}\, A$. And,

given $p = \langle pre,\ post \rangle : \mathcal{PrePost}\, A$ and a function $f = \lambda a.\ \langle pre'\ a,\ post'\ a \rangle : A \to \mathcal{PrePost}\, B$, the sequential composition of two computations is naturally specified by defining

$$\mathtt{bind}^{\mathcal{PrePost}}\, p\, f = \langle\ (pre \wedge \forall a.\, post\ a \implies pre'\ a)\ ,\ \lambda b.\ \exists a.\, post\ a \wedge post'\ a\ b\ \rangle : \mathcal{PrePost}\, B$$

The resulting precondition ensures that the precondition of the first computation holds and, assuming the postcondition of the first computation, the precondition of the second computation also holds. The resulting postcondition is then simply the conjunction of the postconditions of the two computations. The order on $\mathcal{PrePost}$ naturally combines the pointwise forward implication order on postconditions with the backward implication order on preconditions.

We formally show that this specification monad is more expressive than the predicate monad above: Any predicate $p : \mathcal{P}\mathrm{red}\, A$ can be coerced to $(\top, p) : \mathcal{PrePost}\, A$, and in the other direction, any pair $(pre, post) : \mathcal{PrePost}\, A$ can be approximated by the predicate $post$, giving rise to a Galois connection, as illustrated in Figure 1. While the monad $\mathcal{PrePost}$ is intuitive for humans, generating efficient verification conditions is generally easier for predicate transformers [Leino 2005].

Forward predicate transformer monad. The predicate monad $\mathcal{P}\mathrm{red}$ can be extended in an alternative way. Instead of fixing a precondition as in $\mathcal{PrePost}$, a specification can be a function from preconditions to postconditions, for instance producing the strongest postcondition of computation for any precondition $pre{:}\mathbb{P}$ given as argument. Intuitively, such a forward predicate transformer on A has type $\mathbb{P} \to (A \to \mathbb{P})$. However, to obtain a monad (i.e., satisfying the expected laws), we have to consider the smaller type $\mathcal{SP}\mathrm{ost}\, A = (pre{:}\mathbb{P}) \to (A \to \mathbb{P}_{/pre})$ of predicate transformers monotonic with respect to pre, where $\mathbb{P}_{/pre}$ is the subtype of propositions implying pre. Returning a value $v : A$ is specified by the predicate transformer $\mathtt{ret}^{\mathcal{SP}\mathrm{ost}}\, v = \lambda pre\, a.\ pre \wedge a = v$, and the sequential composition of two computations is specified as the predicate transformer $\mathtt{bind}^{\mathcal{SP}\mathrm{ost}}\, m\, f = \lambda pre\, b.\ \exists a.\ f\ a\ (m\ pre\ a)\ b$, for $m : \mathcal{SP}\mathrm{ost}\, A$ and $f : A \to \mathcal{SP}\mathrm{ost}\, B$.

Backward predicate transformer monad. As explained in §2.2, backward predicate transformers can be described using the continuation monad with propositions $\mathbb{P}$ as the answer type, namely, $\mathrm{Cont}_{\mathbb{P}}\, A = (A \to \mathbb{P}) \to \mathbb{P}$. Elements $w : \mathrm{Cont}_{\mathbb{P}}\, A$ are predicate transformers mapping a postcondition $post : A \to \mathbb{P}$ to a precondition $w\ post : \mathbb{P}$, for instance the weakest precondition of the computation. Pointwise implication is a natural order on $\mathrm{Cont}_{\mathbb{P}}\, A$:

$$w_1 \le w_2 : \mathrm{Cont}_{\mathbb{P}}\, A \quad \Leftrightarrow \quad \forall (p : A \to \mathbb{P}).\ w_2\, p \Rightarrow w_1\, p$$

However, $\mathrm{Cont}_{\mathbb{P}}$ is not an ordered monad with respect to this order because its $\mathtt{bind}$ is not monotonic. In order to obtain an ordered monad, we restrict our attention to the submonad W^{Pure} of $\mathrm{Cont}_{\mathbb{P}}$ containing the *monotonic* predicate transformers, that is those $w : \mathrm{Cont}_{\mathbb{P}}\, A$ such that

$$\forall (p_1\, p_2 : A \to \mathbb{P}).\quad (\forall (a : A).\, p_1\, a \Rightarrow p_2\, a) \quad \Rightarrow \quad w\, p_1 \Rightarrow w\, p_2,$$

which is natural in verification: we want stronger postconditions to map to stronger preconditions.

This specification monad is more expressive than the pre-/postcondition one above [Swamy et al. 2016]. Formally, a pair $(pre, post) : \mathcal{PrePost}\, A$ can be mapped to the predicate transformer

$$\lambda (p : A \to \mathbb{P}).\ pre \wedge (\forall (a : A).\ post\ a \Rightarrow p\ a) \quad : \quad W^{\mathrm{Pure}}\, A,$$

and vice versa, a predicate transformer $w : W^{\mathrm{Pure}}\, A$ can be approximated by the pair

$$(\quad w\, (\lambda a.\ \top)\quad ,\quad \lambda a.\ (\forall p.\, w\, p \Rightarrow p\, a)\quad) \quad : \quad \mathcal{PrePost}\, A$$

These two mappings define a Galois connection, as illustrated in Figure 1. Further, this Galois connection exhibits $\mathcal{PrePost}\, A$ as the submonad of $W^{\mathrm{Pure}}\, A$ of *conjunctive* predicate transformers, i.e., predicate transformers w commuting with non-empty conjunctions/intersections.

Finally, both W^{Pure} and $\mathcal{SP}\mathrm{ost}$ can be embedded into an even more expressive specification monad $\mathcal{R}\mathrm{el}\mathcal{PrePost}$ consisting of relations between preconditions and postconditions satisfying a few conditions, the full details of which can be found in our Coq formalization.

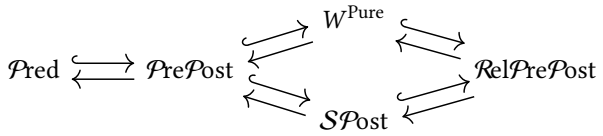

Fig. 1. Relationships between basic specification monads (each pair forms a Gallois connection)

$$C ::= \mathbb{M}A \mid C_1 \times C_2 \mid (x : A) \to C \mid C_1 \to C_2 \quad A \in \mathit{Type}_{\mathcal{L}}$$
$$t ::= \texttt{ret} \mid \texttt{bind} \mid \langle t_1,\ t_2 \rangle \mid \pi_i\ t \mid x \mid \lambda^\circ x.\ t \mid t_1\ t_2 \mid \lambda x.\ t \mid t\ u \quad u \in \mathit{Term}_{\mathcal{L}}$$

Fig. 2. Syntax of SM

$$\frac{}{A \vdash_{\text{SM}} \texttt{ret} : A \to \mathbb{M}A} \qquad \frac{}{A, B \vdash_{\text{SM}} \texttt{bind} : \mathbb{M}A \to (A \to \mathbb{M}B) \to \mathbb{M}B} \qquad \frac{\Gamma, x : C_1 \vdash_{\text{SM}} t : C_2}{\Gamma \vdash_{\text{SM}} \lambda^\circ x.\ t : C_1 \to C_2}$$

Fig. 3. Selected typing rules for SM

4.2 Defining Monad Transformers

We use *monad transformers* [Liang et al. 1995] to construct more complex specification monads from the basic ones above (and in some cases also to derive effect observations §3.3). However, defining a monad transformer and proving that it satisfies all the expected laws requires significant effort. In this section, we introduce a *Specification Metalanguage*, SM, and a translation from SM to correct-by-construction monad transformers in a base dependent type theory $\mathcal{L}$ (where $\mathcal{L}$ is a parameter of SM). More precisely, our translation takes as input a monad in SM subject to two extra conditions, *covariance* and *linearity*, and produces a correct monad transformer in $\mathcal{L}$.

SM is an expressive language in which many different monads can be defined in a natural way, for example *reader* $\mathbb{Rd}(X : \text{Type}) = \mathcal{I} \to \mathbb{M}\,X$; *writer* $\mathbb{Wr}(X : \text{Type}) = \mathbb{M}(X \times O)$; *exceptions* $\mathbb{Exc}(X : \text{Type}) = \mathbb{M}(X + \mathcal{E})$; *state* $\mathbb{St}(X : \text{Type}) = \mathcal{S} \to \mathbb{M}(X \times \mathcal{S})$; *monotonic state* $\mathbb{Mon}\mathbb{St}(X) = (s_0 : \mathcal{S}) \to \mathbb{M}(X \times (s_1 : \mathcal{S}) \times s_0 \preccurlyeq s_1)$, where $\preccurlyeq$ is some preorder on states $\mathcal{S}$; and *continuations* $\mathbb{Cont}_{\mathcal{A}\text{ns}}(X) = (X \to \mathbb{M}\,\mathcal{A}\text{ns}) \to \mathbb{M}\,\mathcal{A}\text{ns}$. The symbol $\mathbb{M}$ stands for an arbitrary base monad, and the *covariance* condition states that it appears only in the codomain of arrows. The more involved *linearity* condition concerns the `bind` of these monads. With the exception of continuations (see §4.3), all these SM monads satisfy these extra conditions and thus lead to proper monad transformers.

Definition 2 (Monad transformer). *A monad transformer [Liang et al. 1995] is given by*
- ▷ *a function $\mathcal{T}$ mapping monads M to monads $\mathcal{T}M$,*
- ▷ *equipped with a monad morphism $\texttt{lift}_M : M \to \mathcal{T}M$,*
- ▷ *assigning functorially to each monad morphism $\theta : M_1 \to M_2$ a monad morphism $\mathcal{T}\theta : \mathcal{T}M_1 \to \mathcal{T}M_2$,*
- ▷ *and such that the $\texttt{lift}_M$ is natural in M, that is for any monad morphism $\theta : M_1 \to M_2$,*

$$\mathcal{T}\theta \circ \texttt{lift}_{M_1} = \texttt{lift}_{M_2} \circ \theta$$

- ▷ *moreover, they need to preserve the order structure present on the (ordered) monads as well as the monotonicity of morphisms, and the lifts themselves should also be monotonic,*

i.e., $(\mathcal{T}, \texttt{lift})$ is a pointed endofunctor on the category of (ordered) monads [Lüth and Ghani 2002].

Building monad transformers. The design of SM, whose syntax is presented in Figure 2, has been informed by the goal of defining monad transformers. First, since we want a mapping from monads to monads, we introduce the type constructor $\mathbb{M}$ standing for an arbitrary base monad, as well as terms `ret` and `bind`. Second, in order to describe monads internally to SM, we add function types $(x : A) \to C[x]$ and $C_1 \to C_2$. We allow dependent function types only when the domain is

$$[\![\mathbb{M}A]\!]_M = MA \qquad [\![C_1 \times C_2]\!]_M = [\![C_1]\!]_M \times [\![C_2]\!]_M \qquad [\![(x : A) \to C]\!]_M = (x : A) \to [\![C]\!]_M$$

$$[\![C_1 \to C_2]\!]_M = (f : [\![C_1]\!]_M \to [\![C_2]\!]_M) \times (\forall (m_1 \leq^{C_1} m_1'). f\ m_1 \leq^{C_2} f\ m_1')$$

$$m \leq^{\mathbb{M}\,A} m' = m \leq_A^M m' \qquad \langle m_1,\ m_2\rangle \leq^{C_1 \times C_2} \langle m_1',\ m_2'\rangle = m_1 \leq^{C_1} m_1' \wedge m_2 \leq^{C_2} m_2'$$

$$f \leq^{(x:A)\to C[x]} f' = \forall (x : A). f\ x \leq^{C[x]} f'\ x \qquad f \leq^{C_1 \to C_2} f' = \forall (m_1 \leq^{C_1} m_1'). f\ m_1 \leq^{C_2} f'\ m_1'$$

Fig. 4. Elaboration from SM to $\mathcal{L}$

in $\mathcal{L}$, leading to two different type formers. We write dependent abstractions as $\lambda x.\ t$, whereas we write the non-dependent type as $\lambda^\circ x.\ t$. In Figure 3 we present the typing rules of $\mathbb{M}$, `ret`, and `bind`, leaving the remaining standard SM typing rules for the online appendix. To define our monad transformers, we use monads *internal to* SM, given by

▷ a type constructor $X : \text{Type} \vdash_{\text{SM}} C[X]$;
▷ terms $A : \text{Type} \vdash_{\text{SM}} \texttt{ret}^C : A \to C[A]$ and $A, B : \text{Type} \vdash_{\text{SM}} \texttt{bind}^C : (A \to C[B]) \to C[A] \to C[B]$;
▷ such that the monadic laws are derivable in the equational theory of SM.

Now, given a monad C internal to SM, we want to define the corresponding monad transformer $\mathcal{T}^C$ evaluated at a monad M in the base language $\mathcal{L}$, essentially as the substitution of M for $\mathbb{M}$. In order to make this statement precise, we define a denotation $[\![-]\!]_M^{(\gamma)}$ in $\mathcal{L}$ of SM types (Figure 4) and terms (provided in the appendix together with the equational theory) parametrized by M. This denotation preserves the equational theory of SM, provided $\mathcal{L}$ has extensional dependent products and pairs. As such, C induces the following mapping from monads to monads:

$$\mathcal{T}^C \quad : \quad (M, \texttt{ret}, \texttt{bind}) \qquad \longmapsto \qquad ([\![C]\!]_M, [\![\texttt{ret}^C]\!]_M, [\![\texttt{bind}^C]\!]_M)$$

For instance, taking $C = \mathbb{St}$, the definition evaluates to $\mathcal{T}^{\mathbb{St}} M = X \mapsto \mathcal{S} \to M(X \times \mathcal{S})$.

To build the `lift` for $\mathcal{T}^C$, the key observation is that the denotation $[\![C]\!]_M$ of an SM type C in $\mathcal{L}$ can be endowed with an M-algebra structure $\alpha_M^C : M[\![C]\!]_M \to [\![C]\!]_M$[5]. This M-algebra structure is defined by induction on the structure of the SM type C, using the free algebra when $C = \mathbb{M}A$ and the pointwise defined algebra in all the other cases. This M-algebra structure allows us to then define a lifting function from the monad M to the monad $[\![C]\!]_M$ as follows:

$$\texttt{lift}_{M,X}^C \quad : \quad M(X) \xrightarrow{M(\texttt{ret}^{[\![C]\!]_M})} M[\![C]\!]_M(X) \xrightarrow{\alpha^C_{M,X}} [\![C]\!]_M(X) = \mathcal{T}^C M(X)$$

For instance, $\texttt{lift}_{M,X}^{\mathbb{St}}\,(m : M\,X) = \lambda(s : \mathcal{S}).\ M(\lambda(x : X).\ \langle x, s\rangle)\,m : \mathcal{S} \to M(X \times S)$. The result that SM type formers are automatically equipped with an algebra structure explains why SM features products, but not sums since the latter cannot be equipped with an algebra structure in general.

This $\texttt{lift}_M^C : M \to [\![C]\!]_M$ needs to be *natural*, that is, the following diagram should commute:

$$\begin{array}{ccccc}
M\,A & \xrightarrow{M(\texttt{ret}_A^C)} & M[\![C]\!]_M\,A & \xrightarrow{\alpha^C_{M,A}} & [\![C]\!]_M\,A \\
{\scriptstyle M f}\downarrow & \circlearrowleft & \downarrow{\scriptstyle M[\![C]\!]_M f} & ? & \downarrow{\scriptstyle [\![C]\!]_M f} \\
M\,B & \xrightarrow[M(\texttt{ret}_B^C)]{} & M[\![C]\!]_M\,B & \xrightarrow[\alpha^C_{M,B}]{} & [\![C]\!]_M\,A
\end{array}$$

for any A, B and $f : A \to B$. The left square commutes automatically by the naturality of $M(\texttt{ret}^C)$. For the right square to commute, however, $[\![C]\!]_M f = \texttt{bind}^{[\![C]\!]_M}(\texttt{ret}^{[\![C]\!]_M} \circ f)$ should be an M-algebra homomorphism. We can ensure it by asking that $\texttt{bind}^C$ maps functions to M-algebra homomorphisms, a condition that can be syntactically captured by a linearity condition in a modified type system for SM equipped with a *stoup*, which is a distinguished variable in the context

[5] An *M-algebra* is an object X together with a map $\alpha : MX \longrightarrow X$, which is required to respect $\texttt{ret}^M$ and $\texttt{bind}^M$.

such that the term is linear with respect to that variable [Egger et al. 2014; Munch-Maccagnoni 2013]. We omit this refined type system here and refer to the online appendix for the complete details. We call this condition on the monad $(C, \mathtt{ret}^C, \mathtt{bind}^C)$ internal to SM *the linearity of* $\mathtt{bind}^C$.

Action on monad morphism. To define a monad transformer, we still need to build a functorial action mapping monad morphism $\theta : M_1 \to M_2$ between monads M_1, M_2 in $\mathcal{L}$ to a monad morphism $[\![C]\!]_{M_1} \to [\![C]\!]_{M_2}$. However, the denotation of the arrow $C_1 \to C_2$ does not allow for such a functorial action since C_1 necessarily contains a subterm $\mathbb{M}$ in a contravariant position. In order to get an action on monad morphisms, we first build a (logical) relation between the denotations. Given M_1, M_2 monads in $\mathcal{L}$ and a family of relations $R_A \subset M_1 A \times M_2 A$ indexed by types A, we build a relation $\{\!|C|\!\}^R_{M_1,M_2} \subset [\![C]\!]_{M_1} \times [\![C]\!]_{M_2}$ as follows

$$\begin{aligned} m_1\ \{\!|\mathbb{M}A|\!\}\ m_2 &= m_1\ R_A\ m_2 \\ (m_1, m_1')\ \{\!|C_1 \times C_2|\!\}\ (m_2, m_2') &= m_1\ \{\!|C_1|\!\}\ m_2 \wedge m_1'\ \{\!|C_2|\!\}\ m_2' \\ f_1\ \{\!|(x : A) \to C|\!\}\ f_2 &= \forall (x : A).\ f_1\ x\ \{\!|C\ x|\!\}\ f_2\ x \\ f_1\ \{\!|C_1 \to C_2|\!\}\ f_2 &= \forall m_1\ m_2.\ m_1\ \{\!|C_1|\!\}\ m_2 \Rightarrow f_1\ m_1\ \{\!|C_2|\!\}\ f_2\ m_2 \end{aligned}$$

Now, when a type C in SM comes with the data of an internal monad, the relational denotation $\{\!|C|\!\}^-_{M,W}$ maps not only families of relations to families of relations, but also preserves the following structure that we call a *monadic relation*:

Definition 3 (Monadic relation). *A monadic relation $\mathcal{R} : M \leftrightarrow W$ between a computational monad M and a specification monad W, consists of:*

- ▷ *a family of relations $\mathcal{R}_A : M\,A \times W\,A \to \mathbb{P}$ indexed by type A*
- ▷ *such that returned values are related $(\mathtt{ret}^M\ v)\,\mathcal{R}_A\,(\mathtt{ret}^W\ v)$ for any value $v : A$*
- ▷ *and such that sequencing of related values is related*

$$\frac{m_1\,\mathcal{R}_A\,w_1 \qquad \forall (x : A).\,(m_2\,x)\,\mathcal{R}^B\,(w_2\,x)}{(\mathtt{bind}^M\ m_1\ m_2)\,\mathcal{R}^B\,(\mathtt{bind}^W\ w_1\ w_2)}$$

The simplest example of monadic relation is the graph of a monad morphism $\theta : M \to W$. Given a monadic relation, we extend the relational translation to terms and obtain the so-called fundamental lemma of logical relations.

Theorem 1 (Fundamental lemma of logical relations). *For any monads M_1, M_2 in $\mathcal{L}$, monadic relation $\mathcal{R} : M_1 \leftrightarrow M_2$, term $\Gamma \vdash_{SM} t : C$ and substitutions $\gamma_1 : [\![\Gamma]\!]_{M_1}$ and $\gamma_2 : [\![\Gamma]\!]_{M_2}$, if for all $(x : C') \in \Gamma$, $\gamma_1(x)\ \{\!|C'|\!\}^{\mathcal{R}}_{M_1,M_2}\ \gamma_2(x)$ then $[\![t]\!]^{\gamma_1}_{M_1}\ \{\!|C|\!\}^{\mathcal{R}}_{M_1,M_2}\ [\![t]\!]^{\gamma_2}_{M_2}$.*

As a corollary, an internal monad C in SM preserves monadic relations, the relational interpretation of $\mathtt{ret}^C$ and $\mathtt{bind}^C$ providing witnesses to the preservation of the monadic structure. In particular, any monad morphism $\theta : M_1 \to M_2$ defines a monadic relation $\{\!|C|\!\}^{\mathrm{graph}(\theta)}_{M_1,M_2} : [\![C]\!]_{M_1} \leftrightarrow [\![C]\!]_{M_2}$. It turns out that if C is moreover *covariant*, meaning that it does not contain any occurrence of an arrow $C_1 \to C_2$ where C_1 is a type in SM, then the relational denotation $\{\!|C|\!\}^{\mathrm{graph}(\theta)}_{M_1,M_2}$ with respect to any monad morphism $\theta : M_1 \to M_2$ is actually the graph of a monad morphism. To summarize:

Theorem 2 (Construction of monad transformer from SM). *Given a monad C internal to SM such that $\mathtt{bind}^C$ satisfies the linearity criterion, we obtain:*

- ▷ *if C is covariant, then $\mathcal{T}^C$ equipped with $\mathtt{lift}^C_M : M \to \mathcal{T}^C M$ is a (ordered) monad transformer;*
- ▷ *if C is not covariant, $\mathcal{T}^C$ defines a pointed endofunctor on the category of (ordered) monads and monadic relations.*

We note that the resulting design for SM is close to Moggi's monadic metalanguage, since it contains the same type formers: a unary type former $\mathbb{M}$, products $C_1 \times C_2$ and functions $A \to C$. The

main difference is that SM is not parameterized on a set of simple base types but on a dependent type theory $\mathcal{L}$. As such, SM captures the essential elements of the metalanguage DM of Ahman et al. [2017], leaving the non-necessary parts, such as sum types, to the base language $\mathcal{L}$.

4.3 The Continuation Monad Pseudo-Transformer

Crucially, the internal continuation monad $\mathbb{C}\mathsf{ont}_{\mathcal{A}\mathrm{ns}}$ does *not* verify the conditions to define a monad transformer since it is not covariant in $\mathbb{M}$. We study this (counter-)example in detail since it extends the definition of Jaskelioff and Moggi [2010] to monadic relations and clarifies the prior work of Ahman et al. [2017], where a Dijkstra monad was obtained in a similar way.

While SM gives us both the computational continuation monad $[\![\mathbb{C}\mathsf{ont}_{\mathcal{A}\mathrm{ns}}]\!]_{\mathrm{Id}} = \mathrm{Cont}_{\mathcal{A}\mathrm{ns}}$ and the corresponding specification monad $[\![\mathbb{C}\mathsf{ont}_{\mathcal{A}\mathrm{ns}}]\!]_{\mathrm{Cont}_{\mathbb{P}}} = \mathrm{Cont}_{\mathrm{Cont}_{\mathbb{P}}(\mathcal{A}\mathrm{ns})}$, we only get a monadic relation between the two and not a monad morphism. We write this monadic relation as follows:

$$[\![\mathbb{C}\mathsf{ont}_{\mathcal{A}\mathrm{ns}}]\!]_{\mathrm{Id}} \quad \longleftarrow\!\!\{\!|\mathbb{C}\mathsf{ont}_{\mathcal{A}\mathrm{ns}}|\!\}^{\mathsf{ret}}_{\mathrm{Id},\mathrm{Cont}_{\mathbb{P}}}\!\!\longrightarrow \quad [\![\mathbb{C}\mathsf{ont}_{\mathcal{A}\mathrm{ns}}]\!]_{\mathrm{Cont}_{\mathbb{P}}}$$

One probably wonders what are the elements related by this relation? Unfolding the definition, we get that a computation $m : [\![\mathbb{C}\mathsf{ont}_{\mathcal{A}\mathrm{ns}}]\!]_{\mathrm{Id}}(X)$ and a specification $w : [\![\mathbb{C}\mathsf{ont}_{\mathcal{A}\mathrm{ns}}]\!]_{\mathrm{Cont}_{\mathbb{P}}}(X)$ are related if

$$\begin{aligned}
&m \;\{\!|\mathbb{C}\mathsf{ont}_{\mathcal{A}\mathrm{ns}}|\!\}^{\mathsf{ret}}_{\mathrm{Id},\mathrm{Cont}_{\mathbb{P}}}\; w \\
&\Leftrightarrow \forall (k : X \to \mathcal{A}\mathrm{ns})\,(w_k : X \to \mathrm{Cont}_{\mathbb{P}}(\mathcal{A}\mathrm{ns})).\,(\forall (x : X).\,\mathsf{ret}\,(k\,x) = w_k\,x) \Rightarrow \mathsf{ret}\,(m\,k) = w\,w_k \\
&\Leftrightarrow \forall (k : X \to \mathcal{A}\mathrm{ns}).\,\mathsf{ret}\,(m\,k) = w\,(\lambda x.\,\mathsf{ret}\,(k\,x)) \\
&\Leftrightarrow \forall (k : X \to \mathcal{A}\mathrm{ns})\,(p : \mathcal{A}\mathrm{ns} \to \mathbb{P}).\,w\,(\lambda x\,q.\,q\,(k\,x))\,p = p\,(m\,k)
\end{aligned}$$

For illustration, if we take $\mathcal{A}\mathrm{ns} = \mathbb{1}$, the last condition reduces to $\forall (p : \mathbb{P}).\,w\,(\lambda x\,q.\,q)\,p = p$, in particular any sequence $x_0, \ldots, x_n$ induces an element $w = \lambda k\,p.\,k\,x_0(\ldots k\,x_n\,p) : [\![\mathbb{C}\mathsf{ont}_{\mathcal{A}\mathrm{ns}}]\!]_{\mathrm{Cont}_{\mathbb{P}}}(X)$ that can be seen as a specification revealing some intensional information about the computation m at hand, namely, that the continuation k was called with the arguments $x_0, \ldots, x_n$ in this particular order. Computationally however, in the case of $\mathcal{A}\mathrm{ns} = \mathbb{1}$, m is extensionally equal to $\lambda k.\,* : \mathrm{Cont}_{\mathbb{1}}$.

4.4 Effect Observations from Monad Algebras

While monad transformers $\mathcal{T}$ enable us to derive complex specification monads, they can only help us to automatically derive effect observations of the form $\theta^{\mathcal{T}} : \mathcal{T}(\mathrm{Id}) \longrightarrow \mathcal{T}(W)$ (see §3.3), which only slightly generalize the *DM4Free* construction. In all other cases in §3, we had to define effect observations by hand. However, when the specification monad has a specific shape, such as W^{Pure}, there is in fact a simpler way to define effect observations. For instance, in §3.2 effect observations $\theta^{\bot}, \theta^{\top} : \mathrm{Exc} \to W^{\mathrm{Pure}}$ were used to specify the total and partial correctness of programs with exceptions, by making a global choice of allowing or disallowing exceptions. Here we observe that such hand-rolled effect observations can in fact be automatically derived from M-algebras.

As shown by Hyland et al. [2007], there is a one-to-one correspondence between monad morphisms $M \to \mathrm{Cont}_R$ and M-algebras $M\,R \to R$. We can extend this to the ordered setting: for instance, effect observations $\theta : M \to W^{\mathrm{Pure}}$ correspond one-to-one to M-algebras $\alpha : M\,\mathbb{P} \to \mathbb{P}$ that are monotonic with respect to the free lifting on $M\,\mathbb{P}$ of the implication order on $\mathbb{P}$. Intuitively, α describes a global choice of how to assign a specification to computations in M in a way that is compatible with ret^M and bind^M, e.g., such as disallowing all (or perhaps just some) exceptions.

Based on this correspondence, the effect observations $\theta^{\bot}$ and $\theta^{\top}$ arise from the Exc-algebras $\alpha^{\bot} = \lambda\_.\,\bot$ and $\alpha^{\top} = \lambda\_.\,\top$. Similarly, the effect observations for nondeterminism from §3.3 arise from the NDet-algebras $\alpha^{\forall}$ and $\alpha^{\exists}$, taking respectively the conjunction and disjunction of a set of propositions in $\mathrm{NDet}(\mathbb{P})$, as follows: $\theta^{\forall}(m) = \lambda p.\,\alpha^{\forall}\,(\mathrm{NDet}(p)\,m)$ and $\theta^{\exists}(m) = \lambda p.\,\alpha^{\exists}\,(\mathrm{NDet}(p)\,m)$. Conversely, we can recover the NDet-algebra $\alpha^{\forall}$ as $\lambda m.\,\theta^{\forall}_{\mathbb{P}}(m)\,\mathrm{id}_{\mathbb{P}}$, respectively $\alpha^{\exists}$ as $\lambda m.\,\theta^{\exists}_{\mathbb{P}}(m)\,\mathrm{id}_{\mathbb{P}}$.

Importantly, this correspondence is not limited to W^{Pure}, but applies to continuation monads with any answer type. For instance, taking the answer type to be $S \to \mathbb{P}$, we can recover the effect observation $\theta^{\mathrm{St}} : \mathrm{St} \to W^{\mathrm{St}}$, where $W^{\mathrm{St}} A \cong \mathrm{MonCont}_{S \to \mathbb{P}} A = (A \to (S \to \mathbb{P})) \to (S \to \mathbb{P})$, from the St-algebra $\alpha^{\mathrm{St}} = \lambda(f : S \to (S \to \mathbb{P}) \times S)\,(s : S).\ (\pi_1\,(f\,s))\,(\pi_2\,(f\,s)) \ : \ \mathrm{St}(S \to \mathbb{P}) \to S \to \mathbb{P}$.

5 DIJKSTRA MONADS FROM EFFECT OBSERVATIONS

As illustrated in §3.4, Dijkstra monads can be obtained from effect observations $\theta : M \to W$ between computational and specification monads. As we shall see this construction is generic and leads to a categorical equivalence between Dijkstra monads and effect observations. In this section, we introduce more formally the notion of Dijkstra monad using dependent type theory, seen as the internal language of a comprehension category [Jacobs 1993], and then build a category $\mathcal{DM}\mathrm{on}$ of Dijkstra monads. In order to compare this notion of Dijkstra monads to effect observations, we also introduce a category of monadic relations $\mathcal{M}\mathrm{on}\mathcal{R}\mathrm{el}$ and show that there is an adjunction

$$\int \ \dashv \ \mathfrak{pre} \quad : \quad \mathcal{M}\mathrm{on}\mathcal{R}\mathrm{el} \longrightarrow \mathcal{DM}\mathrm{on}. \tag{1}$$

Intuitively, an adjunction establishes a correspondence between objects of two categories, here $\mathcal{M}\mathrm{on}\mathcal{R}\mathrm{el}$ and $\mathcal{DM}\mathrm{on}$. An adjunction always provides an equivalence of categories if we restrict our attention to objects that are in one-to-one correspondence, those for which the unit (resp. the counit) of the adjunction is an isomorphism. When we restrict the adjunction above, we obtain an equivalence between Dijkstra monads and effect observations. For the sake of explanation, we proceed in two steps: first, we consider Dijkstra monads and effect observations over specification monads with a discrete order (i.e., ordinary monads), describing the above adjunction in this situation; later, we extend this construction to general preorders, thus obtaining the actual adjunction we are interested in. We denote categories defined over non-discrete specification monads with $\cdot^{\leq}$.

Dependent type theory and comprehension categories. We work in an extensional type theory with dependent products $(x : A) \to B$, strong sums $(x : A) \times B$, an identity type $x =^A y$ for $x, y : A$ (where the type A is usually left implicit), a type of (proof-irrelevant) propositions $\mathbb{P}$, and quotients of equivalence relations. This syntax is the internal language of a comprehension category [Jacobs 1993] with enough structure and we will write $\mathcal{T}ype$ for any such category. This interpretation of type theory allows us to call any object $\Gamma \in \mathcal{T}ype$ a type. [6]

Dijkstra monads. A *Dijkstra monad* over a (specification) monad W is given by

▷ for each type A and specification $w : W\,A$, a type $\mathcal{D}\,A\,w$ of "computations specified by w"

▷ return and bind functions specified respectively by the return and bind of W

$$\mathtt{ret}^{\mathcal{D}} : (x : A) \to \mathcal{D}\,A\,(\mathtt{ret}^{W}\,x)$$

$$\mathtt{bind}^{\mathcal{D}} : \mathcal{D}\,A\,w_1 \to ((x : A) \to \mathcal{D}\,B\,w_2(x)) \to \mathcal{D}\,B\,(\mathtt{bind}^{W}\,w_1\,w_2)$$

▷ such that the following monadic equations about $\mathtt{ret}^{\mathcal{D}}$ and $\mathtt{bind}^{\mathcal{D}}$ hold

$$\mathtt{bind}^{\mathcal{D}}\,m\,\mathtt{ret}^{\mathcal{D}} = m \qquad \mathtt{bind}^{\mathcal{D}}\,(\mathtt{ret}^{\mathcal{D}}\,x)\,f = f\,x$$

$$\mathtt{bind}^{\mathcal{D}}\,(\mathtt{bind}^{\mathcal{D}}\,m\,f)\,g = \mathtt{bind}^{\mathcal{D}}\,m\,(\lambda x.\,\mathtt{bind}^{\mathcal{D}}\,(f\,x)\,g)$$

where $m : \mathcal{D}\,A\,w, x : A, f : (x : A) \to \mathcal{D}\,B\,(w'\,x), g : (y : B) \to \mathcal{D}\,C\,(w''\,y)$ for A, B, C any types and $w : W\,A, w' : (x : A) \to W\,B, w'' : (y : B) \to W\,C$. Note that the typing of these equations depends on the monadic equations for W and they would not be well-typed otherwise.

In order to use multiple Dijkstra monads, that is multiple effects, in a single program, we need a way to go from one to another, not only at the level of computations, but also at the level of specifications. A morphism of Dijkstra monads from $\mathcal{D}_1\,A\,(w_1 : W_1\,A)$ to $\mathcal{D}_2\,A\,(w_2 : W_2\,A)$ provides exactly that: it is a pair $(\Theta^{W}, \Theta^{\mathcal{D}})$ of a monad morphism $\Theta^{W} : W_1 \to W_2$ mapping specifications of

[6] Under a mild condition that the category is *democratic* [Clairambault and Dybjer 2014].

the source Dijkstra monad to specifications of the target Dijkstra monad, and a family of maps

$$\Theta^{\mathcal{D}}_{A,w_1} \quad : \quad \mathcal{D}_1\, A\, w_1 \quad \longrightarrow \quad \mathcal{D}_2\, A\, (\Theta^W\, w_1)$$

indexed by types A and specifications $w_1 : W_1\, A$, satisfying the following axioms

$$\Theta^{\mathcal{D}}(\mathtt{ret}^{\mathcal{D}_1}\, x) = \mathtt{ret}^{\mathcal{D}_2}\, x, \qquad \Theta(\mathtt{bind}^{\mathcal{D}_1}\, m\, f) = \mathtt{bind}^{\mathcal{D}_2}\, (\Theta^{\mathcal{D}}\, m)\, (\Theta^{\mathcal{D}} \circ f).$$

This gives a category $\mathcal{DM}\mathrm{on}$ of Dijkstra monads and morphisms between them.

Monadic Relations. Given a monadic relation $\mathcal{R} : M \leftrightarrow W$ (Def. 3) between a computational monad M and a specification monad W, we construct a Dijkstra monad $\mathfrak{pre}\ \mathcal{R}$ on W as follows:

$$(\mathfrak{pre}\ \mathcal{R})\, A\, (w : W\, A) \quad = \quad (m : M\, A) \times m\, \mathcal{R}_A\, w \tag{2}$$

That is $(\mathfrak{pre}\ \mathcal{R})\, A\, w$ consists of those elements m of $M\, A$ that are related by $\mathcal{R}$ to the specification w. When $\mathcal{R}$ is the graph of a monad morphism θ (or equivalently, $\mathcal{R}$ is functional), $\mathfrak{pre}(\mathcal{R} : M \leftrightarrow W)$ maps an element $w : W\, A$ to its preimage $\theta^{-1}(w) = \{m : M\, A \mid \theta(m) = w\}$.

Conversely, any Dijkstra monad $\mathcal{D}$ over W yields a monad structure on

$$\textstyle\int \mathcal{D}\, A = (w : W\, A) \times \mathcal{D}\, A\, w$$

and the projection of the first component is a monad morphism $\pi_1 : \int \mathcal{D} \to W$.

In order to explain the relation between these two operations $\mathfrak{pre}$ and $\int -$, we introduce the category $\mathcal{M}\mathrm{on}\mathcal{R}\mathrm{el}$ of monadic relations. An object of $\mathcal{M}\mathrm{on}\mathcal{R}\mathrm{el}$ is a pair of monads M, W together with a monadic relation $\mathcal{R} : M \leftrightarrow W$ between them. A morphism between $\mathcal{R}^1 : M_1 \leftrightarrow W_1$ and $\mathcal{R}^2 : M_2 \leftrightarrow W_2$ is a pair (Θ^M, Θ^W) where $\Theta^M : M_1 \longrightarrow M_2$ and $\Theta^W : W_1 \longrightarrow W_2$ such that

$$\forall (m : MA)\, (w : WA).\, m\, \mathcal{R}^1_A\, w \implies \Theta^M(m)\, \mathcal{R}^2_A\, \Theta^W(w). \tag{3}$$

The construction $\mathfrak{pre}$ extends to a functor on $\mathcal{M}\mathrm{on}\mathcal{R}\mathrm{el}$ by sending a pair (Θ^W, Θ^M) to a pair $(\Theta^W, \Theta^{\mathcal{D}})$, where $\Theta^{\mathcal{D}}_{A,w}$ is the restriction of Θ^M_A to the appropriate domain. Conversely, $\int$ packs up a pair $(\Theta^W, \Theta^{\mathcal{D}})$ as (Θ^W, Θ^M), where $\Theta^M_A(w, m) = \Theta^{\mathcal{D}}_{A,w}(m)$. Since Θ^M maps the inverse image of w to the inverse image of $\Theta^W(w)$, condition (3) holds. Moreover, this gives rise to a natural bijection

$$\mathcal{M}\mathrm{on}\mathcal{R}\mathrm{el}(\textstyle\int \mathcal{D}, \mathcal{R}) \qquad \cong \qquad \mathcal{DM}\mathrm{on}(\mathcal{D}, \mathfrak{pre}\ \mathcal{R})$$

that establishes the adjunction (1). We can restrict (1) to an equivalence by considering only those objects for which the unit (resp. counit) of the adjunction is an isomorphism. Every Dijkstra monad $\mathcal{D}$ is isomorphic to its image $\mathfrak{pre}\ (\int \mathcal{D})$, whereas a monadic relation $\mathcal{R}$ is isomorphic to $\int(\mathfrak{pre}\ \mathcal{R})$ if and only if it is functional, i.e., a monad morphism. This way we obtain an equivalence of categories between $\mathcal{DM}\mathrm{on}$ and the category of effect observations on monads with discrete preorder.

The ordered setting. Recall that in the examples of §3.4, the Dijkstra monads $\mathcal{D}\, A\, (w : W\, A)$ we derived from effect observations $\theta : M \longrightarrow W$ naturally made use of the order on W to compare programmer-provided specifications to type-inferred ones. This order structure on W can be naturally lifted to Dijkstra monads, by requiring $\mathcal{D}$ to be equipped with a *weakening* structure

$$\mathtt{weaken} \quad : \quad w_1 \leq_A w_2 \times \mathcal{D}\, A\, w_1 \quad \longrightarrow \quad \mathcal{D}\, A\, w_2$$

such that the following axioms hold (where we conflate the propositions $w_1 \leq w_2$ and their proofs)

$$\mathtt{weaken}\langle w \leq w, m\rangle = m, \qquad \mathtt{weaken}\langle w_1 \leq w_2 \leq w_3, m\rangle = \mathtt{weaken}\langle w_2 \leq w_3, \mathtt{weaken}\langle w_1 \leq w_2, m\rangle\rangle$$

$$\mathtt{bind}^{\mathcal{D}}\, (\mathtt{weaken}\langle w_m \leq w'_m, m\rangle)\, (\lambda a.\, \mathtt{weaken}\langle w_f\, a \leq w'_f\, a, f\, a\rangle) =$$

$$\mathtt{weaken}\langle \mathtt{bind}^W\, w_m\, w_f \leq \mathtt{bind}^W\, w'_m\, w'_f, \mathtt{bind}^{\mathcal{D}}\, m\, f\rangle.$$

Such pairs of an ordered monad W and a Dijkstra monad $\mathcal{D}\, A\, (w : W\, A)$ with a weakening structure form a category $\mathcal{DM}\mathrm{on}^{\leq}$, whose morphisms are pairs of a monotonic monad morphism and a Dijkstra monad morphism preserving the weakening structure. Further, the definition of $\mathfrak{pre}$ extends similarly straightforwardly to the ordered setting: given a monad morphism $\theta : M \to W$, we define

$$(\mathfrak{pre}\ \theta)\, A\, (w : W\, A) \quad = \quad (m : M\, A) \times \theta(m) \leq_A w \tag{4}$$

This definition coincides with (2) when the order $\leq_A$ on $W\,A$ is discrete. Moreover, we can equip $\mathfrak{pre}\ \theta$ with a weakening structure: $\mathsf{weaken}\langle w_1 \leq w_2, \langle m, \theta(m) \leq w_1\rangle\rangle = \langle m, \theta(m) \leq w_1 \leq w_2\rangle$.

The same construction can be performed starting with an *upward closed* monadic relation $\mathcal{R} : M \leftrightarrow W$, i.e., such that M has a discrete order and $\forall m.\, \forall (w_1 \leq_A^W w_2).\, m\,\mathcal{R}_A\,w_1 \implies m\,\mathcal{R}_A\,w_2$. Doing so, we obtain a functor $\mathfrak{pre} : \mathcal{M}\mathrm{on}\mathcal{R}\mathrm{el}^{\leq} \longrightarrow \mathcal{DM}\mathrm{on}^{\leq}$ from the category of upward-closed monadic relations to the category of ordered Dijkstra monads with a weakening structure.

However, when trying to build a left adjoint $\int$ to $\mathfrak{pre}$ exactly as before, there is a small mismatch with the expected construction on practical examples. Indeed, starting from a monad morphism $\theta : M \to W$, $\int(\mathfrak{pre}\ \theta)$ reduces to $(\Sigma M, W, \pi_1)$ where $\Sigma M\,A = (w : W\,A) \times (m : M\,A) \times \theta_A(m) \leq_A w$, which is unfortunately not isomorphic to M. The problem is that we get one copy of m for each admissible specification $w : W\,A$. These copies, however, are non-essential since the weakening structure of $\mathfrak{pre}\ \theta$ identifies them. As such, to define $\int$, we need to further quotient them [7], defining

$$\int\,\mathcal{D}\,A = ((w : W\,A) \times \mathcal{D}\,A\,w)\,/\!\sim$$

where $\sim$ is generated by $\langle w, c\rangle \sim \langle w', \mathsf{weaken}\langle w \leq w', c\rangle\rangle$, giving us the desired adjunction $\int \dashv \mathfrak{pre}$.

To summarize, we can construct Dijkstra monads with weakening out of effect observations and the other way around. Moreover, when starting from an effect observation $\theta : M \to W$, then $\int(\mathfrak{pre}\ \theta)$ is equivalent to θ. This result shows that we do not lose anything when moving from effect observations to Dijkstra monads, and that we can, in practice, use either the effect observation or the Dijkstra monad presentation, picking the one that is most appropriate for the task at hand.

6 ALGEBRAIC EFFECTS AND EFFECT HANDLERS FOR DIJKSTRA MONADS

In §2.1, we noted that all our example computational monads come with corresponding canonical side-effect causing operations. This is an instance of a general approach to modeling computational effects algebraically using operations (specifying the sources of effects) and equations (specifying their behavior), as pioneered by Plotkin and Power [2002, 2003]. From the programmer's perspective, *algebraic effects* naturally enable programming against an abstract interface of operations instead of a concrete implementation of a monad, with the accompanying notion of *effect handlers* enabling one to modularly define different fit-for-purpose implementations of these abstract interfaces.

6.1 Algebraic Effects for Dijkstra Monads

We begin by showing how effect observations naturally equip both the specification monad and the corresponding Dijkstra monad with algebraic operations in the sense of Plotkin and Power [2002, 2003]. We observed several instances of this phenomenon for state, IO, and nondeterminism in §3.4, and we can now explain it formally in terms of algebraic effects and effect observations.

Algebraic operations. For any monad M, an *algebraic operation* $\mathsf{op} : I \rightsquigarrow O$ with input (parameter) type I and output (arity) type O is a family $\mathsf{op}_A^M : I \times (O \to MA) \to MA$ that satisfies the following coherence law for all $i : I$, $m : O \to MA$, and $f : A \to MB$ [Plotkin and Power 2003]:

$$\mathsf{bind}^M\,(\mathsf{op}_A^M\,\langle i, m\rangle)\,f = \mathsf{op}_B^M\,\langle i, \lambda o.\,\mathsf{bind}^M\,(m\,o)\,f\rangle \tag{5}$$

For NDet, the two operations are $\mathtt{pick} : \mathbb{1} \rightsquigarrow \mathbb{B}$ and $\mathtt{fail} : \mathbb{1} \rightsquigarrow \mathbb{0}$. For St, the operations are $\mathtt{get} : \mathbb{1} \rightsquigarrow S$ and $\mathtt{put} : S \rightsquigarrow \mathbb{1}$. Plotkin and Power also showed that such algebraic operations are in one-to-one correspondence with *generic effects* $\mathsf{gen}_{\mathsf{op}}^M : I \to MO$, which are often a more natural presentation for programming. For example, the generic effect corresponding to the put operation for St has type $S \to \mathrm{St}\,\mathbb{1}$. They are interconvertible with algebraic operations as follows:

$$\mathsf{gen}_{\mathsf{op}}^M\,i = \mathsf{op}_O^M\,\langle i, \lambda o.\,\mathtt{ret}^M o\rangle \qquad \mathsf{op}_A^M\,\langle i, m\rangle = \mathsf{bind}^M\,(\mathsf{gen}_{\mathsf{op}}^M i)\,(\lambda o.\,m\,o) \tag{6}$$

[7] We conjecture that an alternative and more symmetric solution would be to equip our Dijkstra monads with an additional order, but this does not correspond to the examples we obtain in practice.

Plotkin and Power also show that signatures *Sig* of algebraic operations determine many computational monads (except continuations) once they are also equipped with suitable sets of equations *Eq*. In the following we write $T_{(Sig,Eq)}$, abbreviated as T, for the monad determined by (Sig, Eq).

Effect observations. In §5, we saw that Dijkstra monads are equivalent to effect observations $\theta : M \to W$. When $M = T$, then since θ is a monad morphism, it automatically transports any algebraic operations on the computation monad T to the (ordered) specification monad W:

$$\mathsf{op}^W_A\ \langle i, w\rangle = \mu^W_A\ (\theta_{WA}\ (\mathsf{op}^T_{WA}\ \langle i, \lambda o.\ \mathtt{ret}^T (w\, o)\rangle)) \tag{7}$$

where $\mu^W : W \circ W \to W$ is the *multiplication* (or *join*) of W, defined as $\mu^W_A\, w = \mathtt{bind}^W\, w\, (\lambda w'.\ w')$.

This derivation of algebraic operations is in fact a result of a more general phenomenon. Namely, given any monad morphism $\theta : M \to W$, we get a family of M-algebras on W, natural in A, by

$$\mu^W_A \circ \theta_{WA} : MWA \longrightarrow WWA \longrightarrow WA$$

Furthermore, the derived algebraic operations op^W_A (resp. the derived M-algebras on W) are monotonic with respect to the free lifting of the preorder $\leq^{W\,A}$ on WA to TWA (resp. to MWA).

The derivation of operations on the specification monad from operations on the computational monad, via the effect observation, explains how we are able to systematically generate (computationally natural) specifications for operations in §3.4. For instance, taking the effect observation $\theta^\forall : \mathrm{NDet} \to W^{\mathrm{Pure}}$ for demonic nondeterminism, the induced operations we get on W^{Pure} are

$$\mathtt{pick}^{W^{\mathrm{Pure}}}_A : \mathbb{1} \times (\mathbb{B} \to W^{\mathrm{Pure}} A) \to W^{\mathrm{Pure}} A \qquad \mathtt{fail}^{W^{\mathrm{Pure}}}_A : \mathbb{1} \times (\mathbb{0} \to W^{\mathrm{Pure}} A) \to W^{\mathrm{Pure}} A$$

$$\mathtt{pick}^{W^{\mathrm{Pure}}}_A \langle (), w\rangle = \lambda p.\ w\, \mathtt{true}\, p \wedge w\, \mathtt{false}\, p \qquad \mathtt{fail}^{W^{\mathrm{Pure}}}_A \langle (), w\rangle = \lambda p.\ \top$$

Dijkstra monads. Finally, we show that the Dijkstra monad $\mathcal{D} = \mathfrak{pre}\ \theta$ derived from a given effect observation $\theta : T \to W$ in (4) also supports algebraic operations, with their computational structure given by the operations of T and their specificational structure given by the operations of W derived in (7). This completes the process of lifting operations from computational monads to Dijkstra monads that we sketched in §3.4. In detail, we define an algebraic operation for $\mathcal{D}$ as

$$\mathsf{op}^{\mathcal{D}}_A : (i : I) \to \big(c : (o : O) \to \mathcal{D}\, A\, (w\, o)\big) \to \mathcal{D}\, A\, (\mathsf{op}^W_A\ \langle i, w\rangle)$$

$$\mathsf{op}^{\mathcal{D}}_A\, i\, c = \big\langle\ \mathsf{op}^T_A \langle i, \lambda o.\ c\, o\rangle\ ,\ \theta_A(\mathsf{op}^T_A \langle i, \lambda o.\ c\, o\rangle) \leq \mathsf{op}^W_A \langle i, w\rangle\ \big\rangle$$

For instance, for $\mathrm{ND}_{\maltese} = \mathfrak{pre}\ \theta^\forall$, the induced operations have the following (expected) types:

$$\mathtt{pick}^{\mathrm{ND}_{\maltese}}_A : (\_ : \mathbb{1}) \to \big(c : (b : \mathbb{B}) \to \mathrm{ND}_{\maltese}\, A\, (w\, b)\big) \to \mathrm{ND}_{\maltese}\, A\, (\lambda p.\ w\, \mathtt{true}\, p \wedge w\, \mathtt{false}\, p)$$

$$\mathtt{fail}^{\mathrm{ND}_{\maltese}}_A : (\_ : \mathbb{1}) \to \big(c : (z : \mathbb{0}) \to \mathrm{ND}_{\maltese}\, A\, (w\, z)\big) \to \mathrm{ND}_{\maltese}\, A\, (\lambda p.\ \top)$$

As we have defined $\mathsf{op}^{\mathcal{D}}$ in terms of algebraic operations for T and W, then it is easy to see that it also satisfies an appropriate variant of the algebraic operations coherence law (5), namely

$$\mathtt{bind}^{\mathcal{D}}\ (\mathsf{op}^{\mathcal{D}}_A\ \langle i, c\rangle)\ f = \mathsf{op}^{\mathcal{D}}_B\ \langle i, \lambda o.\ \mathtt{bind}^{\mathcal{D}}\ (c\, o)\ f\rangle$$

Finally, based on (6), we note that the generic effect corresponding to $\mathsf{op} : I \rightsquigarrow O$ is given by

$$\mathsf{gen}^{\mathcal{D}}_{\mathsf{op}}\, i = \big\langle\ \mathsf{gen}^{\mathcal{T}}_{\mathsf{op}}\, i\ ,\ \theta_O\, (\mathsf{gen}^{\mathcal{T}}_{\mathsf{op}}\, i) \leq \mathsf{gen}^W_{\mathsf{op}}\, i\ \big\rangle : I \to \mathcal{D}\, O\, (\mathsf{gen}^W_{\mathsf{op}}\, i)$$

Specifying operations in free monads. As we have seen above, effect observations induce specifications for algebraic operations, which in turn are used as the indices of the corresponding Dijkstra monad operations. Note that in the case of free monads, when $Eq = \emptyset$, we have the freedom to assign arbitrary specifications. Let $\mathrm{FreeM} \cong T_{(Sig,\emptyset)}$ be the free monad over some signature *Sig*:

$$\mathrm{FreeM}\ A = \mu X.A + \Sigma_{\mathsf{op}_i : I_i \rightsquigarrow O_i \in Sig} I_i \times (O_i \to X)$$

To specify the operations, we assume that for each op_i, we have a precondition $P_i : I_i \to \mathbb{P}$ and a postcondition $Q_i : I_i \times O_i \to \mathbb{P}$. From these we can build a FreeM-algebra $h : \mathrm{FreeM}(\mathbb{P}) \to \mathbb{P}$ by

$$h(\mathtt{inl}\ \phi) = \phi \qquad h(\mathtt{inr}\ \langle \mathsf{op}_i, inp, k\rangle) = P_i\ inp \wedge \forall oup.\ Q_i \langle inp, oup\rangle \to k\ oup$$

Following §4.4, we can derive an effect observation $\theta : \mathrm{FreeM} \to W^{\mathrm{Pure}}$ from h, from which we can in turn derive a W^{Pure}-indexed Dijkstra monad $\mathcal{D} = \mathrm{pre}\ \theta$ following §5. The operations op_i of the free monad lift to generic effects in $\mathcal{D}$, with specifications derived from the assumed P_i and Q_i:

$$\mathsf{gen}^{\mathcal{D}}_{\mathsf{op}_i} : (inp : I_i) \to \mathcal{D}\ O_i\ (\lambda p.\ P_i\ inp \wedge \forall oup.\ Q_i \langle inp, oup\rangle \to p\ oup)$$

As an example, consider the operation `pick` : $\mathbb{1} \rightsquigarrow \mathbb{B}$ introduced above but with the specification that always returns `true`. This is captured by the precondition $P_{\mathtt{pick}}\ \_ = \top$ and postcondition $Q_{\mathtt{pick}}\langle \_, b\rangle = (b = \mathtt{true})$, which yields the following generic effect (after simplifying its type):

$$\mathsf{gen}^{\mathcal{D}}_{\mathtt{pick}} : \mathcal{D}\ \mathbb{B}\ (\lambda p.\ p\ \mathtt{true})$$

In contrast to the demonic non-determinism specification of `pick` given above for $\mathrm{ND}_{\maltese}$, this variant of `pick` always derives its weakest precondition from the `true` case of the post condition.

In the next section, we will use the ability to assign arbitrary specifications to operations in a free monad to define the proof obligations required to verify effect handlers for those operations.

6.2 Effect Handlers for Dijkstra Monads

Of course, operations are only one side of algebraic effects: the other side concerns *effect handlers* [Plotkin and Pretnar 2013]. These are a generalization of exception handlers to arbitrary algebraic effects. They are defined by providing a concrete implementation for each (abstract) operation, such as `get`. Semantically, they denote user-defined T-algebras for the algebraic effect at hand.

In contrast to the general story for algebraic effects in §6.1, our treatment of effect handlers for Dijkstra monads is currently more ad hoc. We have two approaches, which can roughly characterised in terms of how the operations are assigned specifications. In the first approach, we do not explicitly give a specification for each operation. Instead, the specification is induced by the handler. This approach fits well with the philosophy of Dijkstra monads with weakest precondition specifications, i.e., automatically generating the most general specification from the program. This approach works well for exceptions and allows us to reconstruct the weakest precondition semantics for exceptions with try/catch in the setting of Dijkstra monads ([Leino and van de Snepscheut 1994; Sekerinski 2012]), and put the ad-hoc examples of Ahman et al.'s [2017] *DM4Free* on a general footing.

Unfortunately, for *resumable* operations (i.e., everything except exceptions), the inevitable circularity between the handler and the handled code leads to attempts to construct inductive propositions that do not exist in Coq or F*. To resolve this problem, we also demonstrate a second approach that makes use of upfront specification of operations, as demonstrated at the end of §6.1. This specification of operations breaks the circularity, and allows handling of operations that resume, such as `pick`. However, this approach is also not yet fully satisfactory: the operation clauses of the handler must be verified *extrinsically*, in contrast to the usual methodology of Dijkstra monads.

Effect handling (1st approach). Following Plotkin and Pretnar [2013], we define the *handling* of T (determined by some (Sig, Eq)) into a monad M to be given by the following operation:

$$\mathtt{handle\text{-}with}^{T,M} : T\,A \to \left(h_{\mathsf{op}_i} : I_i \times (O_i \to MB) \to MB\right)_{\mathsf{op}_i : I_i \rightsquigarrow O_i \in Sig} \to (A \to MB) \to MB$$

$$\mathtt{handle\text{-}with}^{T,M}\ (\mathtt{ret}^T_A\ a)\ h\ f = f\ a$$

$$\mathtt{handle\text{-}with}^{T,M}\ (\mathsf{op}^T_A\ \langle i, t\rangle)\ h\ f = h_{\mathsf{op}}\ \langle i, \lambda o.\ \mathtt{handle\text{-}with}^{T,M}\ (t\ o)\ h\ f\rangle$$

where we leave the proof obligation that the operation clauses h_{op} have to satisfy the equations in Eq implicit. We refer the reader to Ahman [2018] for explicit treatment of such proof obligations.

As such, h forms a T-algebra $\alpha_h : TMB \to MB$, and $\mathtt{handle\text{-}with}^{T,M}\ (-)\ h\ p\ f$ amounts to the induced unique mediating T-algebra homomorphism $\alpha_h \circ T(f) : TA \to TMB \to MB$.

Specification monads. Based on the category theoretic view of effect handlers as user-defined T-algebras, we can define a notion of handling any monad M into some other monad M':

$$\begin{aligned}&\texttt{handle-with}^{M,M'} : M\,A \to (\alpha : MM'B \to M'B) \to (A \to M'B) \to M'B\\ &\texttt{handle-with}^{M,M'}\ m\,\alpha\,f = (\alpha \circ M(f))\,m\end{aligned}$$

where we again leave the proof obligation ensuring that α is an M-algebra implicit. Below we are specifically interested in $\texttt{handle-with}^{M,M'}$ when M and M' are specification monads because, in contrast to T, the structure of specification monads is not determined by (Sig, Eq) alone.

Dijkstra monads. Based on the smooth lifting of algebraic operations in §6.1, then when defining effect handling for the Dijkstra monad $\mathcal{D} = \mathfrak{pre}\ \theta$ induced by some effect observation $\theta : T \to W$ into some other Dijkstra monad $\mathcal{D}' = \mathfrak{pre}\ \theta'$ for $\theta' : M \to W'$, we would expect the computational (resp. specificational) structure of handling to be given by that for T (resp. W).

However, simply giving an effect observation θ turns out to be insufficient for handling $\mathcal{D}$ into $\mathcal{D}'$. Category theoretically, the problem lies in the operation cases for W giving us a T-algebra $TW'B \to W'B$, but to use $\texttt{handle-with}^{W,W'}$ (which we need to define the specification of handling) we instead need a W-algebra $WW'B \to W'B$. To overcome this difficulty, we introduce a more refined notion of effect observation, relative to the specification monad W' we are handling into.

Definition 4 (Effect observation with effect handling). *An effect observation with effect handling for an ordered monad W' is an effect observation $\theta : T \to W$ such that for any T-algebra $\alpha : TW'A \to W'A$, there is a choice of a W-algebra $\alpha_* : WW'A \to W'A$ that is (i) monotonic with respect to the orders of W and W', and (ii) which additionally satisfies the equation $\alpha_* \circ \theta_{W'A} = \alpha$.*

Intuitively, the condition (ii) expresses that α_* extends a T-algebra to a W-algebra in a way that is identity on the T-algebra structure, specifically on the algebraic operations corresponding to α.

It is worth noting that needing to turn algebras $TW'A \to W'A$ into algebras $WW'A \to W'A$ is not simply a quirk due to working with Dijkstra monads, but the same exact need arises when giving a monadic semantics to a language with effect handlers using a monad different from $T_{(Sig,Eq)}$.

Using this refined notion of effect observation, we can now define handling for Dijkstra monads. Given an effect observation $\theta : T \to W$ with effect handling for W' and another effect observation $\theta' : M \to W'$, we define the handling of $\mathcal{D} = \mathfrak{pre}\ \theta$ into $\mathcal{D}' = \mathfrak{pre}\ \theta'$ as the following operation

$$\begin{aligned}\texttt{handle-with}^{\mathcal{D},\mathcal{D}'} :\ & \mathcal{D}\,A\,w_1\\ &\to \left(h^{W'}_{\mathsf{op}_i} : I_i \times (O_i \to W'B) \to W'B\right)_{\mathsf{op}_i : I_i \rightsquigarrow O_i \in Sig}\\ &\to \left(h^{\mathcal{D}'}_{\mathsf{op}_i} : \left((i,c) : (i : I_i) \times \left((o : O_i) \to \mathcal{D}'\,B\,(w\,o)\right)\right) \to \mathcal{D}'\,B\,(h^{W}_{\mathsf{op}}\langle i, w\rangle)\right)_{\mathsf{op}_i \in Sig}\\ &\to \left((a : A) \to \mathcal{D}'\,B\,(w_2\,a)\right)\\ &\to \mathcal{D}'\,B\,(\texttt{handle-with}^{W,W'}\ w_1\,(\alpha_{h^{W'}})_*\,w_2)\end{aligned}$$

$$\begin{aligned}\texttt{handle-with}^{\mathcal{D},\mathcal{D}'}\ c_1\,h^{W'}\,h^{\mathcal{D}'}\,c_2 = \Big\langle\ & \texttt{handle-with}^{T,M}\ c_1\,h^{\mathcal{D}'}\,(\lambda a.\ c_2\,a)\,,\\ & \theta'_B\left(\texttt{handle-with}^{T,M}\ c_1\,h^{\mathcal{D}'}\,(\lambda a.\ c_2\,a)\right) \le \left(\texttt{handle-with}^{W,W'}\ w_1\,(\alpha_{h^{W'}})_*\,w_2\right)\ \Big\rangle\end{aligned}$$

where we again leave implicit the conditions ensuring that $h^{W'}$, $h^{\mathcal{D}'}$ are correct with respect to Eq.

Exception handling. One effect observation supporting handling is $\theta^{\mathrm{Exc}} : \mathrm{Exc} \to W^{\mathrm{Exc}}$ from §3.2, where Exc is determined by $(\{\texttt{throw}\}, \emptyset)$. To model handling potentially exceptional computations into other similar ones, as is often the case in languages with exceptions but no effect system, we take $\mathcal{D} = \mathcal{D}' = \mathrm{EXC} = \mathfrak{pre}\ \theta^{\mathrm{Exc}}$ and observe that $\texttt{handle-with}^{\mathrm{EXC},\mathrm{EXC}}$ can be simplified to

$$\begin{aligned}\texttt{try-catch} : \mathrm{EXC}\,A\,w_1 \to \left(h^{W^{\mathrm{Exc}}}_{\mathsf{throw}} : E \to W^{\mathrm{Exc}}B\right) \to \left(h^{\mathrm{EXC}}_{\mathsf{throw}} : (e : E) \to \mathrm{EXC}\,B\,(h^{W^{\mathrm{Exc}}}_{\mathsf{throw}}\,e)\right)\\ \to \left((a : A) \to \mathrm{EXC}\,B\,(w_2\,a)\right) \to \mathrm{EXC}\,B\left(\lambda p\,q.\ w_1\,(\lambda x.\ w_2\,x\,p\,q)\,(\lambda e.\ h^{W^{\mathrm{Exc}}}_{\mathsf{throw}}\,e\,p\,q)\right)\end{aligned}$$

in part, by defining the extension $\alpha_* : W^{\mathrm{Exc}}W^{\mathrm{Exc}}B \to W^{\mathrm{Exc}}B$ of $\alpha : \mathrm{Exc}\,W^{\mathrm{Exc}}B \to W^{\mathrm{Exc}}B$ as

$$\alpha_*\,w = \lambda p\,q.\,w\left(\lambda w'.\,\alpha\,(\texttt{inl}\,w')\,p\,q\right)\left(\lambda e.\,\alpha\,(\texttt{inr}\,e)\,p\,q\right) = \lambda p\,q.\,w\left(\lambda w'.\,w'\,p\,q\right)\left(\lambda e.\,\alpha\,(\texttt{inr}\,e)\,p\,q\right) \quad (8)$$

where the second equality holds because α is an Exc-algebra and thus $\alpha\,(\mathtt{ret}^{\mathrm{Exc}}\,v) = \alpha\,(\mathtt{inl}\,v) = v$.

On inspection, it turns out that `try-catch` corresponds exactly to Leino and van de Snepscheut's [1994] and Sekerinski's [2012] weakest exceptional preconditions for exception handlers. Furthermore, with `try-catch` we can also put Ahman et al.'s [2017] hand-rolled *DM4Free* exception handlers to a common footing. For example, we can define their integer division example as

```
let div_wp (i j:int) = λp q → (∀ x . j ≠0 ∧ x = i / j ⟹ p x) ∧ (∀ e . j = 0 ⟹ q e)
let div (i j:int) : EXC int (div_wp i j) = if j = 0 then raise div_by_zero_exn else i / j
let try_div (i j:int) : EXC int (λ p q → ∀x . p x) = try_catch (div i j) (λ _ p q → p 0) (λ _ → 0) (λ x → x)
```

where the specification of try_div says that it never throws an exception, even not div_by_zero_exn.

Of course `try-catch` is not the only way to handle exceptions. Another common use case is to handle a computation in TA into one in $\mathrm{Id}(A+E)$. While this is trivial semantically, in a programming language where elements of T are considered abstract, it allows one to get their hands on the values returned and exceptions thrown, analogously to Ahman et al.'s [2017] use of monadic reification in *DM4Free*. To capture this, we take $\mathcal{D} = \mathrm{EXC} = \wp\mathrm{re}\;\theta^{\mathrm{Exc}}$ and $\mathcal{D}' = \mathrm{PURE} = \wp\mathrm{re}\;\theta^{\mathrm{Pure}}$, and define

$$\mathtt{reify} : \mathrm{EXC}\,A\,w \to \mathrm{PURE}\,(A+E)\,\big(\lambda p.\; w\,(\lambda x.\; p\,(\mathtt{inl}\,x))\,(\lambda e.\; p\,(\mathtt{inr}\,e))\big)$$
$$\mathtt{reify}\,c = \mathtt{handle\text{-}with}^{\mathrm{EXC},\mathrm{PURE}}\,c\,\big(\lambda e.\mathtt{ret}^{W^{\mathrm{Pure}}}\,(\mathtt{inr}\,e)\big)\big(\lambda e.\mathtt{ret}^{\mathrm{PURE}}\,(\mathtt{inr}\,e)\big)\big(\lambda x.\mathtt{ret}^{\mathrm{PURE}}\,(\mathtt{inl}\,x)\big)$$

Other (non-)examples. Unfortunately, effect observations discussed in this paper other than exceptions do not support effect handling. Specifically, we are unable to define the α_* operation for these effect observations, because it corresponds to attempting to construct the specification of the handled computation knowing nothing of the intended specification of the operations.

For IO, we actually know of another specification monad for which α_* can be defined, namely, the categorical coproduct of the IO and continuation monads [Hyland et al. 2007], given by

$$W^{\mathrm{IO}}A = (A \to \mathbb{P}) \times ((I \to W^{\mathrm{IO}}A) \to \mathbb{P}) \times (O \times W^{\mathrm{IO}}A \to \mathbb{P}) \to \mathbb{P}$$

Note that compared to the specification monads for IO from §3.4, the postcondition(s) of W^{IO} have a tree-like structure that enables one to recover enough information to (recursively) define α_*.

There are however two major problems with using W^{IO} as a specification monad. First, W^{IO} is not well-defined in many categories of interest, such as Set [Hyland et al. 2007]. Second, defining W^{IO} type theoretically requires non strictly positive inductive types, which leads to inconsistency in frameworks with impredicative universes such as Coq and F* [Coquand and Paulin 1988].

Effect handling for upfront specified operations (2nd approach). We now describe an alternative approach to effect handling that avoids the above problems by making use of the upfront specified operations discussed at the end of §6.1. For simplicity, we assume that we are handling into a pure computation of type B with a postcondition $R : B \to \mathbb{P}$. We also assume that the computation to be handled performs operations $\mathtt{op}_i$ with the specifications (P_i, Q_i) as given above, yielding values of type A satisfying some postcondition Q, i.e., it has the type $\mathcal{D}\;A\;(\lambda p.\;\forall a.\,Q\,a \to p\,a)$.

The return clause of the handler then gets to assume that Q holds for its input but must ensure that R holds of its output. The operation cases of the handler are more complex. We must first write each operation clause without specification (i.e., as a function $I_i \to (O_i \to B) \to B$), and then separately prove that, assuming that the resumption argument is verified, then the final result is verified. Note that we must separately program and verify the handler clauses, contrary to the general methodology for programming with Dijkstra monads. This is due to the higher-order nature of the resumption argument. Putting all this together, we get the following handling construct:

$$\begin{aligned}\mathtt{handle} :\;& \mathcal{D}\;A\;(\lambda p.\;\forall a.\,Q\,a \to p\,a)\\ &\to ((a:A) \to Q\,a \to (b{:}B) \times R\,b)\\ &\to (h_i : I_i \to (O_i \to B) \to B)_{\mathtt{op}_i}\\ &\to (\forall inp\,k.\,(\forall oup.\,Q_i\langle inp, oup\rangle \to R\,(k\;oup)) \to P_i\;inp \to R\,(h_i\;inp\;k))_{\mathtt{op}_i}\\ &\to (b{:}B) \times R\,b\end{aligned}$$

For the "always `true`" specification of `pick`, we can write a handler for it as $\lambda\_\ k.\ k\ \texttt{true}$, which yields the trivial proof obligation $\forall k.\ (\forall b.\ b = \texttt{true} \to R\,(k\,b)) \to R\,(k\,\texttt{true})$. Note that this obligation would not hold if the handler had relied upon invoking the resumption k with `false`.

We used a variant of this second approach to verify programs with general recursion in Coq, reconstructing from first principles F*'s primitive support for total correctness, as well as its semantic termination checking [Swamy et al. 2016]. Following McBride [2015], we can describe a recursive function $f : (a : A) \to B\,a$ by its body $f_0 : (a : A) \to \mathrm{GenRec}\,(B\,a)$, where GenRec is the free monad on a single operation $\texttt{call} : (a : A) \rightsquigarrow B\,a$ and the recursive calls to f are replaced by uses of `call`. Given a well-founded order $\prec$ on A, we ask that all arguments to `call` are lower than the *top-level* argument. More precisely, given an invariant $inv : (a : A) \to W^{\mathrm{Pure}}\,(B\,a)$ for f, we define a family of effect observations $\theta_a : \mathrm{GenRec} \to W^{\mathrm{Pure}}$ as described above, i.e., such that $\theta_a(\texttt{call}\ a')$ strengthens $inv\,a'$ with the precondition $a' \prec a$. From these θ_as, we obtain a Dijkstra monad GenREC together with a handling construct $\texttt{fix} : ((a : A) \to \mathrm{GenREC}\,(B\,a)\,(inv\,a)) \to (a : A) \to \mathrm{PURE}\,(B\,a)\,(inv\,a)$. We have used this treatment of general recursion to define and verify a simple Fibonacci example.

Compared to the "specify at handling time" approach above, this "specify upfront" approach to effect handlers has the advantage that it works for algebraic effects that involve resumptions. However, there remain several unresolved questions with this approach, including handling stateful computations and whether or not it is possible to program and verify the handler clauses simultaneously to be more in keeping with the general methodology of Dijkstra monads.

7 IMPLEMENTATION AND FORMALIZATION IN F* AND COQ

Dijkstra monads in F*. We have extended the effect definition mechanism of F* to support our more general approach to Dijkstra monads, in addition to the previous *DM4Free* one. F* users can now also define Dijkstra monads by providing both a computational and a specification monad, along with an effect observation or monadic relation between them, which provides more freedom in the choice of specifications. The SM language is not yet implemented in F*. Nevertheless, this extension enables the verification of the examples of §3.4, for which effects such as nondeterminism and IO were previously out of reach. Once a Dijkstra monad is defined, the F* type-checker computes weakest preconditions exactly as before and uses an SMT solver to discharge them. While internally F* only uses weakest preconditions as specification monads, it is customary for users to write Hoare-style pre- and postconditions, for which F* leverages the adjunction from §4.1.

Dijkstra monads in Coq. We have also embedded Dijkstra monads in Coq, showing that the concept is applicable in languages beyond F*. As with the F* implementation, programmers can supply their own computational and specification monads, with an effect observation or monadic relation between them. We implemented the base specification monads of §4.1 and the construction of effect observations from algebras of monads from §4.4, thus providing a convenient way to build a specification monad and effect observation at the same time. The Coq development also constructs Dijkstra monads from effect observations and proves their laws hold. Therefore, the examples from §3.4 are verified in Coq "all the way down". Verification in Coq follows the general pattern of (a) writing the specification; (b) writing the program in monadic style; and then (c) proving the resulting verification conditions using tactic proofs. The Dijkstra monad setup automatically takes care of the derivation of the weakest precondition transformer for the program.

Formalization of SM in Coq. We have formalized the SM language of §4.2 in Coq, taking Gallina as the base language $\mathcal{L}$ and providing an implementation of the denotation of SM terms and logical relation. SM is implemented using higher-order abstract syntax (HOAS) for the $\lambda x.\ t$ binders and De Bruijn indices for the $\lambda^\circ x.\ t$ ones. We build the functional version of the logical relation for a covariant type C, but omit the linear type system. Instead, the Coq version of Theorem 2 assumes

a semantic hypothesis requiring that the denotation of bind is homomorphic, and using which it then derives the full monad transformer (including all the laws). A paper proof that our syntactic linearity condition entails the semantic hypothesis can be found in the online appendix.

8 RELATED WORK

This work directly builds on prior work on Dijkstra monads in F* [Swamy et al. 2013, 2016], in particular the *DM4Free* approach [Ahman et al. 2017], which we discussed in detail in §1 and §2.2. Our generic framework has important advantages: (1) it removes the previous restrictions on the computational monad; (2) it gives much more flexibility in choosing the specification monad and effect observation; (3) it builds upon a generic dependent type theory, not on F* in particular.

Jacobs [2015] studies adjunctions between state transformers and predicate transformers, obtaining a class of specification monads from the state monad transformer and an abstract notion of logical structures. He gives abstract conditions for the existence of such specification monads and of effect observations. Hasuo [2015] builds on the state-predicate adjunction of Jacobs to provide algebra-based effect observations (in the style of §4.4) for various computation and specification monads. Our work takes inspiration from this, but provides a more concrete account focused on covering the use of Dijkstra monads for program verification. In particular, we provide concrete recipes for building specification monads useful for practical verification (§4). Finally, we show that our Dijkstra monads are equivalent to the monad morphisms built in these earlier works.

Katsumata [2014] uses graded monads to give semantics to type-and-effect systems, introduces effect observations as monad morphisms, and constructs graded monads out of effect observations by restricting the specification monads to their value at $\mathbb{1}$. We extend his construction to Dijkstra monads, showing that they are equivalent to effect observations, and unify Katsumata's two notions of algebraic operation. A graded monad can intuitively be seen as a non-dependent version of a Dijkstra monad (a monad-like structure indexed by a monoid rather than a monad) but providing a unifying formal account is not completely straightforward. The framework of Kaposi and Kovács [2019] is a promising candidate for such a unifying account that might provide an abstract proof of the results of §5 (see the online appendix); we leave a full investigation as future work.

Katsumata [2013] gives a semantic account of Lindley and Stark [2005]'s $\top\top$-lifting, a generic way of lifting relations on values to relations on monadic computations, parameterized by a basic notion of relatedness at a fixed type. Monad morphisms $MA \to ((A \to \mathbb{P}) \to \mathbb{P})$, as used to generate Dijkstra monads in §3.4, are also unary relational liftings $(A \to \mathbb{P}) \to (MA \to \mathbb{P})$, and could be generated by $\top\top$-lifting. Further, binary relational liftings could be used to generate monadic relations that yield Dijkstra monads by the construction in §5. In both cases, what is specifiable about the underlying computation would be controlled by the chosen basic notion of relatedness.

Rauch et al. [2016] provide a generic verification framework for *first-order* monadic programs. Their work is quite different from ours, even beyond the restriction to first-order programs, since their specifications are "innocent" effectful programs, which can observe the computational context (e.g., state), but not change it. This introduces a tight coupling between computations and specifications, while we provide much greater flexibility through effect observations. In fact, we can embed their work into ours, since their notion of weakest precondition gives rise to an effect observation.

Generic reasoning about computational monads dates back to Moggi's [1989] seminal work, who proposes an embedding of his computational metalanguage into higher-order logic. Pitts & Moggi's evaluation logic [Moggi 1995; Pitts 1991] later introduces modalities to reason about the result(s) of computations, but not about the computational context. Plotkin and Pretnar [2008] propose a generic logic for algebraic effects that encompasses Moggi's computational λ-calculus, evaluation logic, and Hennesy-Milner logic, but does not extend to Hoare-style reasoning for state.

Simpson and Voorneveld [2018] and Matache and Staton [2019] explore logics for algebraic effects by specifying the effectful behaviour of algebraic operations using a collection of effect-specific modalities instead of equations. Their modalities are closely related to how we derive effect observations $\theta : M \to W^{\mathrm{Pure}}$ and thus program specifications from M-algebras on $\mathbb{P}$ in §4.4, as intuitively the conditions they impose on their modalities ensure that these can be collectively treated as an M-algebra on $\mathbb{P}$. In recent work concurrent to ours, Voorneveld [2019] studies a logic based on quantitative modalities by considering truth objects richer than $\mathbb{P}$, including $S \to \mathbb{P}$ for stateful and $[0, 1]$ for probabilistic computation. While the state case we already briefly discussed in the context of deriving effect observations in §4.4, it could be interesting to see if these ideas can be used to enable Dijkstra monads to be also used for reasoning about probabilistic programs.

In another recent concurrent work, Swierstra and Baanen [2019] study the predicate transformer semantics of monadic programs with exceptions, state, non-determinism, and general recursion. Their predicate transformer semantics appears closely related to our effect observations, and their compositionality lemmas are similar to our monad morphism laws. We believe that some of their examples of performing verification directly using the effect observation (instead of our Dijkstra monads), could be easily ported to our framework. Their goal, however, is to start from a specification and incrementally write a program that satisfies it, in the style of the refinement calculus [Morgan 1994]. It could be an interesting future work direction to build a unified framework for both verification and refinement, putting together the ideas of both works.

9 CONCLUSION AND FUTURE WORK

This work proposes a general semantic framework for verifying programs with arbitrary monadic effects using Dijkstra monads obtained from effect observations, which are monad morphisms from a computation to a specification monad. This loose coupling between the computation and the specification monad provides great flexibility in choosing the effect observation most suitable for the verification task at hand. We show that our ideas are general by applying them to both Coq and F*, and we believe that they could also be applied to other dependently-typed languages.

In the future, we plan to apply our framework to further computational effects, such as probability [Giry 1982]. It would also be interesting to investigate richer specification monads, for instance instrumenting W^{St} with information about framing, in the style of separation logic. Another interesting direction is to extend Dijkstra monads and our semantic framework to relational reasoning, in order to obtain principled semi-automated verification techniques for properties of multiple program executions (e.g., noninterference) or of multiple programs (e.g., program equivalence). As a first step, we plan to investigate switching from (ordered) monads to (ordered) relative monads for our specifications, by making return and bind work on pairs of values.

Finally, the SM language provides a general way to obtain correct-by-construction monad transformers, which could be useful in many other settings, especially within proof assistants. Categorical intuitions also suggest potential extensions of SM, e.g., some form of refinement types.

ACKNOWLEDGMENTS

We thank Nikhil Swamy and the anonymous reviewers for their feedback. This work was, in part, supported by the ERC under ERC Starting Grant SECOMP (715753). Guido Martínez' work was done, in part, during an internship at Inria Paris funded by the Microsoft Research-Inria Joint Centre. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

REFERENCES

J. Adámek, S. Milius, N. Bowler, and P. B. Levy. Coproducts of monads on set. *LICS*. 2012.

D. Ahman. Handling fibred algebraic effects. *PACMPL*, 2(POPL):7:1–7:29, 2018.
D. Ahman and T. Uustalu. Update monads: Cointerpreting directed containers. *TYPES*, 2013.
D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. *POPL*. 2017.
N. Benton, J. Hughes, and E. Moggi. Monads and effects. *APPSEM*. 2000.
N. Bowler, S. Goncharov, P. B. Levy, and L. Schröder. Exploring the boundaries of monad tensorability on set. *Logical Methods in Computer Science*, 9(3), 2013.
P. Clairambault and P. Dybjer. The biequivalence of locally cartesian closed categories and Martin-Löf type theories. *Mathematical Structures in Computer Science*, 24(6), 2014.
T. Coquand and C. Paulin. Inductively defined types. *COLOG*, 1988.
G. A. Delbianco and A. Nanevski. Hoare-style reasoning with (algebraic) continuations. *ICFP*. 2013.
E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, 1975.
J. Egger, R. E. Møgelberg, and A. Simpson. The enriched effect calculus: syntax and semantics. *LogCom*, 24(3):615–654, 2014.
R. W. Floyd. Nondeterministic algorithms. *J. ACM*, 14(4):636–644, 1967.
M. Giry. A categorical approach to probability theory. *Categorical Aspects of Topology and Analysis*. 1982.
I. Hasuo. Generic weakest precondition semantics from monads enriched with order. *Theor. Comput. Sci.*, 604:2–29, 2015.
C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
M. Hyland, P. B. Levy, G. D. Plotkin, and J. Power. Combining algebraic effects with continuations. *Theor. Comput. Sci.*, 375 (1-3):20–40, 2007.
B. Jacobs. Comprehension categories and the semantics of type dependency. *Theor. Comput. Sci.*, 107(2):169–207, 1993.
B. Jacobs. Dijkstra monads in monadic computation. *CMCS*, 2014.
B. Jacobs. Dijkstra and Hoare monads in monadic computation. *Theor. Comput. Sci.*, 604:30–45, 2015.
M. Jaskelioff and E. Moggi. Monad transformers as monoid transformers. *Theor. Comput. Sci.*, 411(51-52):4441–4466, 2010.
A. Kaposi and A. Kovács. Signatures and induction principles for higher inductive-inductive types. *arXiv:1902.00297*, 2019.
S. Katsumata. Relating computational effects by $\top\top$-lifting. *Inf. Comput.*, 222:228–246, 2013.
S. Katsumata. Parametric effect monads and semantics of effect systems. *POPL*. 2014.
K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.
K. R. M. Leino and J. L. A. van de Snepscheut. Semantics of exceptions. *PROCOMET*. 1994.
S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. *POPL*. 1995.
S. Lindley and I. Stark. Reducibility and $\top\top$-lifting for computation types. *TLCA*. 2005.
C. Lüth and N. Ghani. Composing monads using coproducts. *ICFP*. 2002.
G. Malecha, G. Morrisett, and R. Wisnesky. Trace-based verification of imperative programs with I/O. *J. Symb. Comput.*, 46 (2):95–118, 2011.
C. Matache and S. Staton. A sound and complete logic for algebraic effects. *FoSSaCS*. 2019.
C. McBride. Turing-completeness totally free. *MPC*. 2015.
E. Moggi. Computational lambda-calculus and monads. *LICS*. 1989.
E. Moggi. A semantics for evaluation logic. *Fundam. Inform.*, 22(1/2):117–152, 1995.
C. Morgan. *Programming from Specifications (2nd Ed.)*. Prentice Hall, Hertfordshire, UK, 1994.
G. Munch-Maccagnoni. *Syntax and Models of a non-Associative Composition of Programs and Proofs. (Syntaxe et modèles d'une composition non-associative des programmes et des preuves)*. PhD thesis, Paris Diderot University, France, 2013.
A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. *ICFP*. 2008a.
A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *JFP*, 18(5-6):865–911, 2008b.
A. Nanevski, A. Banerjee, and D. Garg. Dependent type theory for verification of information flow and access control policies. *ACM TOPLAS*, 35(2):6, 2013.
S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *CACM*, 19(5):279–285, 1976.
A. M. Pitts. Evaluation logic. In *IV Higher Order Workshop, Banff 1990*. Springer, 1991.
G. D. Plotkin and J. Power. Notions of computation determine monads. *FOSSACS*, 2002.
G. D. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.
G. D. Plotkin and M. Pretnar. A logic for algebraic effects. In *LICS*. 2008.
G. D. Plotkin and M. Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.
J. Protzenko and B. Parno. EverCrypt cryptographic provider offers developers greater security assurances. Microsoft Research Blog, 2019.
C. Rauch, S. Goncharov, and L. Schröder. Generic hoare logic for order-enriched effects with exceptions. *WADT*, 2016.
E. Sekerinski. Exceptions for dependability. In *Dependability and Computer Engineering: Concepts for Software-Intensive Systems*, pages 11–35. IGI Global, 2012.
A. Simpson and N. F. W. Voorneveld. Behavioural equivalence via modalities for algebraic effects. *ESOP*. 2018.

N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. *PLDI*, 2013.

N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.

W. Swierstra and T. Baanen. A predicate transformer semantics for effects, 2019.

N. Voorneveld. Quantitative logics for equivalence of effectful programs. *MFPS*. 2019. To appear.