Predicate semantics for effectful programs

Internship report
Inria - Paris, Prosecco team
Under the direction of Catalin Hritcu

Van Muylder Antoine
M2 LMFI - Paris Diderot

1 September 2019

This document presents the work done in the first four months of my internship at Inria Paris in the Prosecco team. It is mostly concerned with *program logics* and particular categorical semantics for these logics: *predicate semantics*.

In the first part of the report we introduce a semantic framework [MAA+19] ("the dm4all framework) allowing to certify effectful programs in a principled way. The main notion appearing is that of *effect observation*, and one of my contributions was to help showing this notion is equivalent (in a precise sense) to the more computational-friendly notion of *Dijkstra monad*.

Next, we discuss a variation of this framework for *relational reasoning*. Instead of checking that a single effectful program meets its specification, the goal is to relate two effectful programs at a relational specification. A significant amount of time in the internship was dedicated to embedding the standard relational program logic RHL ([Ben04]) within the developped semantic setting. Parts of this joint work notably led to a submission for the POPL Symposium [MHRM19].

Finally we compare the dm4all framework to an alternative functorial-flavored predicate semantics studied in [Has15] and observe that the dm4all framework is strictly more expressive than the latter. Aside from the new categorical logic formulation of the dm4all framework, this comparison gives a clue to understand how static ressource analyses can be handled by the dm4all framework.

Contents

1 Introduction

Program logics are deduction systems for proving desirable properties of programs. The user of such a system wishes to prove that the behavior of a program is correct relative to a specification. In the formal verification lingo it is said that the program *meets its specifications* whenever the proof is completed. Floyd-Hoare logic ([Hoa69], [Flo93]) is one of the first program logics, and was designed for the verification of imperative programs in a simple while language. Its basic constituent is the so-called *Hoare-triple* of the form $\vdash \{P\}C\{Q\}$ which is intuitively true if the execution of the program C in an initial state satisfying the properly P either yields a final state satisfying the property Q or diverges. This *specification* is expressed as a pair of formulas, a precondition P and a postcondition Q and is supposed to be provided by the user.

In this document we aim to build formal semantics for various program logics $\mathcal{H}$, that is to assign a meaning $\models S$ to each sequent $\vdash S$ of the logic. One way to achieve this is to first define a *predicate semantics* for each of the programs that are considered by the logic. A semantics for the sequents of $\mathcal{H}$ is then defined on top of this predicate semantics. In the case of Floyd-Hoare logic, Dijkstra himself [Dij78] proposes to consider a *weakest precondition semantics* $\theta(C) = \mathrm{wp}(C, -)$ for each program C. Specifically $\theta(C)$ is called a *predicate transformer semantics* because it converts in this case postconditions into preconditions. The assigned precondition answers to the question: "What is the weakest sufficient condition needed before the execution of C so that the considered postcondition holds after the execution of C?". Once such a predicate semantics for programs θ is fixed we can wonder if a given program meets a user-provided specification (P, Q) relative to θ. In other words it is possible to define a semantics for Hoare logic sequents $\vdash \{P\}C\{Q\}$ on top of the considered predicate semantics θ:

$$\models \{P\}C\{Q\} \quad \text{iff} \quad P \to \mathrm{wp}(C, Q)$$

Hoare logic deals with imperative, standard while programs. However when it comes to verification, it is more advantageous to write our programs in a purely functional programming language, that is a type theory, instead of an imperative one. The first reason is the inherent categorical nature of type theory. More precisely it is often possible to design models of purely functional languages by interpreting them in (structured) categories: think of the simply typed lambda calculus and its categorical counterpart, the cartesian closed categories (CCC). Hence the categorical semantics is greatly simplified and more principled if we switch to a functional language. The second reason to choose a functional language is that it is the place where the Curry-Howard correspondance applies: not only can we express programs in our language but also proofs, because proofs are programs and propositions are types. If the picked type theory is expressive enough (dependent types, inductive types, ...) we can even embed program logics themselve in it, together with their categorical semantics. This embedding allows in turn the formally verified study of meta-properties of the logic and its semantics such as soundness and completeness.

Yet how can we be sure that we don't loose any expressiveness by switching to a purely functional language (efficiency is another issue not discussed here)? Is there a simple way to express imperative programs in a functional language? The answer is yes, with *monads*. Monads are a functional programming concept and allow to express not only state manipulating imperative programs but also a wide range of other *computational effects*: generic recursion, failing computations, exception handling, nondeterministic and probabilistic computations, interactive input/output, continuations, ... Monads admit a categorical interpretation, which is of course the categorical notion of monad.

The dm4all framework. A significant part of this internship was dedicated to the study of a general predicate semantics framework for interpreting arbitrary effectful programs [MAA+19].

From now one we will refer to this semantic framework as "the dm4all framework". As in the case of Hoare logic and Dijkstra wp semantics, it is possible to use this predicate semantics in order to confer semantics to effectful program logics. The main contribution of the dm4all framework is that it abstracts away the effect at hand, wheras existing program logics such as Hoare logic focus on particular sets of effects, and particular verification methodologies. In the dm4all framework, effectful programs are considered as points of a monad: if A is a type and M is a monad accounting for a certain effect, then $M\,A$ is the type of effectful computations returning a value of type A. The key idea of [MAA+19], originating from [NMB08, NBG13, NMS+08, DN13, SWS+13, SHK+16, AHM+17] is that the various properties one can express about monadic programs live in a monad too: a *specification monad*, usually called W. For instance $W\,A$ can be the space of pre- and postconditions of a computation returning a value of type A. Once M and W are fixed, in order to give a predicate semantics to our effectful programs, we fix in a manner similar to that of Dijkstra discussed above what is called an *effect observation* [Kat14], which is nothing more than a monad morphism $\theta : M \to W$. One of my contributions in this internship was to help complete the proof of a result from a recently published paper [MAA+19] showing that these monad morphisms are an equivalent formulation of the more computational-friendly notion of Dijkstra monad. The use of Dijkstra monads for verifying effectful programs is discussed in section §3.

Relational reasoning. The dm4all framework allows to certify that one single program meets a given specification. Numerous more advanced program logics, called *relational program logics*, can establish than two different programs are related according to a relational specification. For instance, such logics can be used to prove that two implementations of a given interface are observationally equivalent, meaning that on same inputs they always return the same outputs [cCLRR16, KTL09, GS10, BAF08, CCcCK16, TSKB18, WDLC18, Yan07, HDNV12, HNDV14]. Non-interference [NBG13, CS10, SM03, AGH+17, BEG+19, SD16, BNN16], cost analyses [ÇBG+17, QGG19, RBG+18] and many other verification tasks can be handled by relational program logics. In section 4 we discuss a relational variation of the dm4all framework. This relational framework is defined in [MHRM19] and allows to confer predicate semantics to relational effectful program logics. Like the dm4all framework, the main contribution of this relational framework is to keep the computational effect abstract. Moreover it theoretically enables the comparison of two programs having two different effects.

Part of the internship consisted in showing that this relational framework is expressive enough to interpret a variant of the standard *Relational Hoare Logic* (RHL, [Ben04]). RHL is a logic reasonning about while-programs and allowing to verify that various program transformations (dead code eliminiation, program slicing, ..) do not change the behaviour of the intial program.

Towards a functorial semantics. The last part of the report 5 is concerned with the comparison of two semantic frameworks for unary verification. On one hand, the dm4all framework fully discussed in 3, and on the other hand a framework originating from Jacobs papers [Jac12], [Jac13], [Jac14] and further studied in [Has15] by Hasuo. The latter is called "Hasuo's framework" throughout this document. One of my contribution was to show that Hasuo's frameork is in fact equivalent to a small fragment of the dm4all framework. Moreover Hasuo's framework is formulated following a categorical logic principle, namely "interpretations are functors". Thus the comparison between the two frameworks confer a functorial-flavored formulation of the predicate semantics designed in the dm4all framework.

Categorical logic and functorial semantics [Jac99], [Shu17] bring a principled perspective on the relation between syntax and semantics. Consider for example the relation between the simply typed lambda calculus and CCC's. Let $\mathcal{L}$ denote the category of simply typed lambda calculi. One object of $\mathcal{L}$ is a type theory completely determined by its set of ground types.

Similarly Let $\mathcal{M}$ denote the category of CCC's. Briefly speaking, categorical logic studies the existing adjunction between syntax and semantics :

$$\mathcal{L} \underset{\text{Lg}}{\overset{\text{Syn}}{\rightleftarrows}} \mathcal{M}$$

The left adjoint $\text{Syn}(T)$ builds the *syntactic category* out of a theory T, i.e. a complete categorical account of the syntax described by T. The right adjoint $\text{Lg}(\mathcal{C})$ extracts the *internal language* from a structured category $\mathcal{C}$, in our case a CCC. This language is a type theory or equivalently a logic and can be used to proof categorical facts about $\mathcal{C}$ in a synthetic way. Consider the adjunction property :

$$\frac{\text{Syn}(T) \to \mathcal{C} : \mathcal{M}}{T \to \text{Lg}(\mathcal{C}) : \mathcal{L}}$$

The structured functor appearing above the line is precisely a functorial semantics of T and is also called a model of T or an interpretation of T. It maps syntactic constructs of the theory T (for example types) into categorical constructs in C. In short categorical logic and functorial semantics state that "interpretations are models, models are functors". There is a nice survey about this relationship on the nlab: [nla].

2 Basic material

This section discusses several topics required for a complete understanding of the report. It can safely be skipped by the expert reader, partly or entirely. All the code is given in a pseudo dependant type theory similar to Coq. We refer the reader to [BHM00], [PP02] and [Mog90] for more details about monads and effects.

2.1 Computational monads

Functional programing proposes to identify the notions of functions and programs. Consequently a program p taking its parameters in a type X and returning values in a type Y is simply described as a member of the function type, i.e. $p : X \to Y$. But sometimes programs do more than returning raw values: they can have *effects*. The simplest example of effect is probably the "failure" effect. Programs subject to failures either return a value if their execution went well or fail. To modelize this particular effect in our functional programming language, we tweak the codomain of $p : X \to Y$ by adding a value $\perp$ interpreted as the failure of the execution. We then have $p : X \to Y + 1$.

However this simple design gives raise to another problem, of compositional nature. What if we want to use the results of p in another program $q : Y \to Z$ that expects a value of type Y to run? This situation happens a lot when it comes to error managing and can lead to terrible code designs if not treated carefully. For example we could rewrite q and change its domain to be $Y + 1$, treating the error case inside q. This is obviously a horrible choice because it contradicts the basic principle of code-reuse. The procedure q already does the job, we don't want to rewrite it. Besides, the amount of time and confusion arising from this adaptation might be ridiculous; think of q being in an external library. The right way to proceed is to *abstract effectful function composition*, and that is precisely what computational monads allow. For the situation above, the error monad $\text{Err}(A) = A + 1$ will let us compose the two procedures p and q, magically managing the eventual errors by itself:

$$X \xrightarrow{p} Y + 1 \quad ; \quad Y \xrightarrow{q} Z \xrightarrow{\eta_Z} Z + 1$$

The operator ; appearing at the center is called the Kleisli composition (sometimes called "fish" and sometimes written `>=>`) and it expects two effectful parametric programs, also called "Kleisli arrows". Of course there is an underlying notion of Kleisli category, which we will discuss later. Since q does not raise errors we need to embed its codomain by using the "unit" η_Z of the error monad at type Z. The unit of the monad is also called the return operator and is also noted `return`. or just `ret`. The unit η at type Z simply maps values from Z to the same values in $Z + 1$. Here is the code for the Kleisli composition of the error monad, where we annotate the parameter types with names for more clarity:

```
Definition KLcomp: (a: X -> Y+1) -> (b: Y -> Z+1) -> X -> Z+1
:= fun a b x =>
   let firstRes = a x in
   match firstRes with
     |Some y => b y
     |None => None
   end.
```

In short when executed, the Kleisli composition of two arrows run the programs normally until an error is raised. In this case it propagates the error to the final result. The Kleisli composition operator of a monad helps us to understand how effects are managed, and is more closely related to the categorical notion of a monad. Nevertheless programmers tend to prefer composing their effectful programs by using another equivalent operator:

The bind operator. Imperative programers (and even functional programers) usually like to name the results of intermediate computations, as in the `let firstRes = a x in` line of code above. Since we are dealing with effectful computations, i.e. members m of $M\,A$ for a certain return type A, we would like to be able to name the results of those intermediate effectful computations as well via a certain layer of abstraction. In the case of the error monad, $M\,A$ would be $A+1$ and that abstraction layer would allow to momentarily *forget* about the fact that a certain effectful computation $m : M\,Y$ might fail. It would allow to reason only about the value y of that prone to error-computation. That is why there exists an equivalent formulation for the Kleisli operator called `bind`, sometimes written `>>=` or more explicitely `bind y:Y <- m in f` This operator is of the following type in general:

```
bind: (m: M Y) -> (f: Y -> M Z) -> M Z
```

The first parameter m is the intermediate effectful computation to whose resulting value we wish to give a name y in the rest of the program. The second parameter is the second Kleisli arrow, the continuation of the program wherein we want to refer to the result y of the previous effectful computation m. It intuitively takes y as a parameter when the binded effectful computation (in $M\,Z$) is run. Now a Kleisli composition of several Kleisli arrows

$$A \xrightarrow{f} M\,B \quad ; \quad B \xrightarrow{g} M\,C \quad ; \quad C \xrightarrow{h} M\,D$$

evaluated for some initial value $a : A$ can be rather rewritten in an imperative style thanks to the `bind` operator:

```
bind b:B <- f a in
bind c:C <- g b in
bind d:D <- h c in
return d
```

Since d is a value and since we need to return an effectful computation of type D, i.e. something of type $M\,D$, we use the unit η_D of the monad, also called `return` to coerce this value to a computation. We are now ready to define what is a computational monad.

Monads in functional programming. A computational monad is a triple (M,`return`,`bind`) where:

- `M: Type -> Type` is a type constructor. Given a type of values A, $M\,A$ can be taught of as the type of effectful computations returning a value of type A.
- `return_A: A -> M A` wraps a value into an "effectless effectful" computation. Indeed the program "`return a`" even if effectful in theory does not actually trigger any effects.
- `bind_AB: M A -> (A -> M B) -> MB` allows to compose effectful computations and to write our effectful programs in a neat imperative way.
- `return` must be neutral for bind, to the left and to the right:

$$\texttt{bind m return} = \texttt{m}$$
$$\texttt{bind return(a) f} = \texttt{f(a)}$$

- `bind` must be associative (since it describes a kind of composition):

$$\texttt{bind ma fun x => (bind f(x) g)} = \texttt{bind (bind ma f) g}$$

In addition to this minimal interface, instances of computational monads exibhit specific *operations* (see [PP03]). In the following we give several example of monads and operations.

The error monad. The definition of the `bind` operator for the error monad $\mathrm{Err}(A) = A + 1$ is of course similar to that of the Kleisli composition operator.

```
Definition bind: (ma: Err A) -> (f: A -> Err B) -> Err B
:= fun ma f =>
match ma with
  |Some a => f a
  |None => None
end.
```

Suppose you want to write the function $f : x \mapsto (x-1)/2/2$ going from $\mathbb{N}$ to $\mathrm{Err}\,\mathbb{N}$. The program `subs: nat -> Err nat` substracting 1 from its argument raises an error on the value 0, and the operation `div2: nat -> Err nat` dividing its argument by 2 raises an error on odd values. Using the `bind` operator (and even the `let` formulation) we can write f like this:

```
Definition f: nat -> Err nat:= fun x =>
bind y <- subs x in
bind yover2 <- div2 y in
bind yover4 <- div2 yover2 in
return yover4
```

Notice that we could refine the error monad so that it manages several errors. The obtained construction $\mathrm{Exc}(A) = A + E$ is still a monad because we can define a `bind` and a `ret` operator in a similar way, and Exc is called the *exception* monad.

The writer monad. This monad is defined by $\mathrm{Wri}(A) = A \times \texttt{string}$ and allows to keep track of various events (here encoded as strings) by inserting them in a log. The bind operator for this monad is of the following form:

```
Definition bind (ma: Wri A) -> (f: A -> Wri B) -> Wri B
:= fun ma f =>
let (a,previousLog) = ma in
let (b,newMsg) = f a in
(b , previousLog ++ newMsg)
```

It is used together with an operation `tell` defined as:

```
Definition tell: string -> Wri unit:= fun msg => (tt,msg)
```

Here is a concrete example:

```
Definition someComp: nat -> Wri nat:= fun x =>
  tell "incrementing." ;
  bind i <- return (x + 1) in (*return gives an empty log*)
  tell "doubling." ;
  bind d <- return (i * 2) in
  tell "decrementing." ;
  return (d - 1).
Compute someComp 7.
```

Notice that the `;` operator is no longer the Kleisli composition but some syntactic sugar for the monadic sequence : a bind operator that does not use the names of former computations. Only the effects are propagated trough this operator: here it just appends the two given logs. The last line of code gives `(15,"incrementing.doubling.decrementing.")`.

The state monad. It is possible to emulate the writer monad using what is called a state monad. The state monad allows to simulate the assignment of values to imperative variables and the reading of those variables in our functional world. The principal idea to come up with this monad is to consider state passing functions. As a first approximation, such functions can be written as: $f : X \times S \to Y \times S$, where S is the type of possible states. To fit this idea in a monadic perspective we then need to curry those functions into $f : X \to (S \to Y \times S)$. Hence the state monad is defined by $\mathrm{St}(A) = S \to (A \times S)$ and has the following `bind` operator:

```
Definition bind (ma: St A) -> (f: A -> St B) -> St B
:= fun ma f s0 =>
let (a,s1) = ma s0 in
let mb = f a in
mb s1.
```

Let us define `get: St S` by `fun s0 => (s0,s0)`. This operation does not change the current state but just coerces it into a value. Similarly, we define `put: S -> St unit` by

```
fun toWrite s0 => (tt,toWrite)
```

This operation simply overwrites the current state by some state passed in argument. We can now define `someComp` in terms of the state monad. In the following code the set S of states is taken to be `string` (intuitively meaning that we have at our disposal one single variable of type string):

```
Definition someComp: nat -> St nat:= fun x =>
  put "incrementing." ;
  bind i <- return (x + 1) in (*return doesnt change the state*)
  bind prviousLog <- get() in   (*prviousLog is now "incrementing."*)
  put prviousLog ++ "doubling." ;
  bind d <- return (i * 2) in
  bind prviousLog' <- get() in
  put prviousLog' ++ "decrementing." ;
  return (d - 1).
```

Here is some other concrete example, an emulation of an imperative program exchanging the two values of two variables. Since there are two variables the global state collection S is taken to be $\mathbb{N} \times \mathbb{N}$:

```
Definition swap: St unit:=
  bind tup <- get() in
  let (a,b) = tup in
  put (b,a).
```

The nondeterminism monad. This monad allows to emulate nondeterministic computations in our functional language. By nondeterministic computation we mean a computation that could return several values instead of one. Hence effectful parametric programs (i.e. Kleisli arrows) are here of the form $f : X \to \mathcal{P}(Y)$ and $\mathrm{Ndet}(A) = \mathcal{P}(A)$. The `bind` operator is defined in the following way (we simulate sets by lists in this pseudo code, even if there is a slight mismatch):

```
Fixpoint bind (ma: Ndet A) (f: A -> Ndet B): Ndet B
:= fun ma f =>
match ma with
  |[] => []  (*"errors" are propagated*)
  |hd::tl => f(hd) ++ bind tl f
end.
```

In short every possible outcome value a of the first computation m is replaced by the set $f(a)$. As a concrete example we write a program computing the cartesian products of two sets. The idea is that the monadic abstraction allows you to pick a generic element in each set and to return the pair. Since the elements a, b were not fixed it returns in fact the list of all pairs.

```
Fixpoint cartprod: list A -> list B -> list (A*B)
:= fun alist blist =>
  bind a <- alist in   (*a generic in A*)
  bind b <- blist in
  return (a,b).
```

The continuation monad. This monad is defined by $\mathrm{Cont}_R(A) = (A \to P) \to P$ for R a type representing "the type of the values returned by the continuations". We simply give the `return` and `bind` operators here. This kind of monads will be useful to describe various predicate semantics.

```
Definition return: A -> Cont A:= fun a =>
fun (k: A -> R) => k a.

Definition bind: (ma: Cont A) -> (f: A -> Cont B) -> Cont B:=
fun ma f => fun (k: B -> R) =>
  let (temp: A -> R) = (fun a => f a k) in
  ma temp
```

$\mathrm{Cont}_R(A)$ should be understood as the type of computations expecting a continuation of type $A \to R$ to finally yield a value of type A.

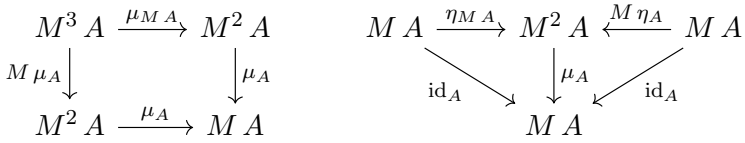
Monad morphisms. Let M and M' be two monads. A monad morphism is simply a function `theta_A: M A -> M' A` parameterized by a type A mapping the `return` operator of the first monad to the `return` operator of the second, and same for the `bind` operator.

$$\texttt{theta (return a)} = \texttt{return' a}$$
$$\texttt{theta (bind m f)} = \texttt{bind' (theta m) (fun a => theta (f a))}$$

2.2 Monads, categorically.

Monads were first introduced as a categorical concept. Computational monads were named monads because they can trivially be interpreted by categorical monads.

Let $\mathcal{C}$ be a category. A monad M over $\mathcal{C}$ is an endofunctor $\mathcal{C} \to \mathcal{C}$ together with two natural transformations. The first one is called the unit of the monad $\eta : \mathrm{Id}_{\mathcal{C}} \to M$, the second one is called the multiplication of the monad $\mu : M^2 \to M$. The following diagrams have to commute in $\mathcal{C}$:

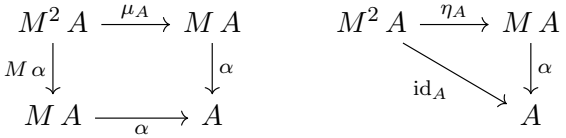


The names unit and multiplication come from the fact that a monad is precisely a monoid object in the monoidal category of endofunctors $[\mathcal{C}, \mathcal{C}]$ where the tensor operation $\otimes$ is instantiated by the composition operator $\circ$.

Monads over a fixed category $\mathcal{C}$ form themselve a category noted $\mathrm{Mon}_{\mathcal{C}}$. A morphism between two monads $\theta : M_1 \to M_2$ is defined as a natural transformation between the underlying functors, preserving the monadic structure as well. A complete definition can be found in [BHM00] for example.

Categories of algebras. Monads are 1-dimensional objects because they are functors. It is possible to recover part of the information contained in a monad into a 0-dimensional object, that is a single category. Categories of algebras play this role for monads.

Given an endofunctor $M : \mathcal{C} \to \mathcal{C}$, an algebra for this functor is an object A of $\mathcal{C}$ together with an arrow $\alpha : M\,A \to A$. An Eilenberg-Moore algebra is moreover required to satisfy the following laws:



These Eilenberg-Moore algebras form a category denoted $\mathcal{EM}(M)$ whose morphisms are explicited in [BHM00] as well.

Besides, one often considers a "smaller" category of algberas, the Kleisli category $\mathcal{K}\ell(M) = \mathcal{K}\ell_M$ of the monad M. This category is not directly defined in terms of algebras of functors but rather as:

- Objects of $\mathcal{K}\ell(M)$ are objects of $\mathcal{C}$.
- Arrows are given by $\mathcal{K}\ell_M(X, Y) = \mathcal{C}(X, M\,Y)$.

Relation between monads and adjunctions. The source of an adjunction $F \dashv G$ is by definition the domain of the left adjoint. It is always possible to build a monad out of an adjunction, by considering the composition $M = GF$, which is an endofunctor over the source, say $\mathcal{C}$. The *unit* of the adjunction is conveniently the same natural transformation $\eta : \mathrm{Id}_{\mathcal{C}} \to M$ as the unit of the built monad M. Let $\mathrm{Adj}_{\mathcal{C}}$ denote the category of adjunctions of source $\mathcal{C}$, i.e. the objects of this category are adjunctions of the following form:

$$\mathcal{C} \underset{G}{\overset{F}{\rightleftarrows}} \mathcal{D}$$

We just described a functor $\mathrm{Mkm} : \mathrm{Adj}_{\mathcal{C}} \to \mathrm{Mon}_{\mathcal{C}}$ that "makes" monads out of adjunctions. The categories of algebras previoulsy described allow to construct adjoints of this functor. Indeed the Kleisli and Eilenberg-Moore categories come with an adjunction structure with source $\mathcal{C}$:

$$\mathcal{C} \underset{G}{\overset{F}{\rightleftarrows}} \mathcal{K}\ell_M \text{ or } \mathcal{EM}_M$$

To visualize this situation we can think of Mkm as being a fibration:

$$\begin{array}{ccc} \mathrm{Mkm}^{-1}(M) & \hookrightarrow & \mathrm{Adj}_{\mathcal{C}} \\ & & \downarrow{\scriptstyle \mathrm{Mkm}} \\ M : & & \mathrm{Mon}_{\mathcal{C}} \end{array}$$

Its fiber over a monad M are the adjunctions $F \dashv G$ of source $\mathcal{C}$ having $M = GF$ as monad. In fact the Kleisli-adjunction for M is initial in the fiber $\mathrm{Mkm}^{-1}(M)$ wheras the Eilenberg-Moore adjunction is terminal in this fiber. So under this point if view, the Kleisli-adjunction functor $\mathcal{K}\ell : \mathrm{Mon}_{\mathcal{C}} \to \mathrm{Adj}_{\mathcal{C}}$ should be understood as some kind of lowest section of the fibration, whereas the Eilenberg-Moore-adjunction functor $\mathcal{EM} : \mathrm{Mon}_{\mathcal{C}} \to \mathrm{Adj}_{\mathcal{C}}$ should be understood as a greatest section.

Relative monads. Relative monads are like monads, but they are not endofunctors. Let $J : \mathcal{I} \to \mathcal{C}$ be a functor. A relative monad M over J is also a functor from $\mathcal{I}$ to $\mathcal{C}$ equipped with

- a relative return operator, i.e. a natural transformation $\mathrm{ret} : J \to M$
- a bind operator $\mathrm{bind}_{AB} : M\,A \to (J\,A \to M\,B) \to M\,B$

satisfying the expected monadic laws (see [ACU15] for more details). A relative monad morphism between two relative monads $M_1 : \mathcal{I}_1 \to \mathcal{C}_1$ and $M_2 : \mathcal{I}_2 \to \mathcal{C}_2$ maps the domain categories via a functor $\mathcal{F}^{\mathcal{I}} : \mathcal{I}_1 \to \mathcal{I}_2$ and the codomain categories via a functor $\mathcal{F}^{\mathcal{C}} : \mathcal{C}_1 \to \mathcal{C}_2$, in a commuting way. More precisely there is a natural transformation filling the following square from top-right to bottom-left:

$$\begin{array}{ccc} \mathcal{I}_1 & \overset{J_1}{\underset{M_1}{\rightrightarrows}} & \mathcal{C}_1 \\ {\scriptstyle \mathcal{F}^I}\downarrow & & \downarrow{\scriptstyle \mathcal{F}^C} \\ \mathcal{I}_2 & \underset{J_2}{\overset{M_2}{\rightrightarrows}} & \mathcal{C}_2 \end{array}$$

Additional commutations are needed to get the right notion of morphisms but we do not delve into such details here, see [ACU15].

As expected every monad is a relative monad with a trivial base functor.

2.3 A bit of domain theory.

Domain theory [AJ94] is concerned with the study of a some classes of orders meant to represent the available information a program can have about a value. It was in particular widely used to interpret fixed-point constructs of functionnal programming languages.

First, we consider complete lattices, i.e. orders with all suprema and infima. The Knaster-Tarski fix-point theorem states that if $f : X \to X$ is an order-preserving transformation of a complete lattice, then the set of fixed points of f forms itself a complete lattice. In particular it is not empty, since it contains a minimum and a maximum (possibly the same element).

Second, there exists a more constructive version of this theorem if the considered transformation f preserves suprema. This version only requires the base set X to be a *pointed directed complete partial order* (dcpo).

- A subset $D \subseteq X$ of an order is said to be *directed* if it is not empty and any two elements $d_1, d_2 \in D$ have an upper bound in D.
- Then a dcpo is an order such that all its directed subsets have a supremum. It is pointed if it has a minimum element $\bot$.

- A Scott continuous function between two dcpo's is a function mapping directed subsets to directed subsets and such that $f(\sup D) = \sup f(D)$ for every directed subset D.

The Kleene fixed point theorem states that if $f : X \to X$ is a Scott continuous function, then it has a fixpoint p. Moreover p is given by the supremum of the follwing ascending chain in X:

$$\bot \leq f(\bot) \leq f^2(\bot) \leq ... \leq f^n(\bot) \leq ...$$

2.4 Total and partial correctness.

Two kinds of program logics are generally identified in the literature. On one hand, program logics achieving *partial correctness* are logics allowing to certify a program m with respect to a given specification w *assuming* the termination of m. On the other hand, program logics achieving *total correctness* are logics allowing to prove that w correctly specifies m *and* that m terminates.

3 Unary verification

This section discusses the *dm4all* framework ([MAA+19]), which is a categorical semantic framework in which effectful programs are interpreted by predicate transformers. Once an interpretation for effectful programs is fixed it can be used to confer semantics to unary effectful program logics, i.e. deduction systems used to certify the behaviour of effectful programs.

A key idea [NMB08, NBG13, NMS+08, DN13, SWS+13, SHK+16, AHM+17, MAA+19] is that the possible specifications of an effectful program form themselve a monad: a *specification monad*, usually called W. Hence it is natural to wonder in what cases this connection between computations on one side, modelled by a computational monad M, and specifications on the other preserves the monadic structure. These considerations give raise to the notion of *effect observation*, which is nothing more than a monad morphism $\theta : M \to W$.

Effects observations turn out to be an alternative formalism for speaking about the more computational-friendly notion of Dijsktra monad. A Dijkstra monad is a monad-like structure $\mathcal{D}\, A\, w$ parametrized by a type A and a specification w. Its points $m : \mathcal{D}\, A\, w$ should be understood as the effectful computations *correctly specified* by w. Below we explain the precise link between the two notions of effect observation and Dijkstra monads.

3.1 Specification monads

A specification monad W is a monad such that for every type A, the collection $W\, A$ of effectful specifications is equipped with a preorder $\leq$. This preorder is meant to model the logical strength of the various predicates. Moreover the `bind` operation of such a monad has to be monotonic in both arguments. In other words if $w_1, w_2 : W\, A$ are specifications and $f_1, f_2 : A \to W\, B$ are Kleisli arrows, such that $w_1 \leq w_2$ and $\forall a : A, f_1(a) \leq f_2(a)$ then

$$\texttt{bind}\, w_1\, f_1 \leq \texttt{bind}\, w_2\, f_2$$

In the following we discuss some classes of examples.

The predicate monad. This basic specification monad is defined by $\mathcal{P}\mathrm{red}(A) = A \to \mathbb{P}$ and can be understood as the (covariant) powerset monad. It is not very expressive as it only allows to specify postconditions of computations. The `return` operator builds the singleton predicate: `return a = fun x => (x = a)`. This predicate can be thought of as a postcondition holding after the execution of an effectless computation solely returning a. The `bind` operator is defined by:

```
Definition bind (p: A -> Prop) (f: A -> B -> Prop):=
fun b => exists (a: A), p a /\ f a b.
```

Here the parameter p should be understood as a postcondition for a computation m returning a value a of type A. Similarly, `f a` is a postcondition holding after the execution of some computation $g(a)$. The bind of these two specifications is the postcondition that should hold after the execution of both m and $g(a)$. As for the preorder structure, it is simply given by the pointwise implication:

$$w_1 \leq w_2 \quad \text{iff} \quad \forall (a : A), w_1(a) \to w_2(a)$$

Of course we would like to be able to express preconditions for programs as well. In order to do so we introduce the following monad.

The pre/post monad. We define $\mathcal{P}\mathrm{rePost}(A) = \mathbb{P} \times (A \to \mathbb{P})$. A specification is now a pair (p, q) of predicates. The `return` operation of the monad yields a pair (p, q) specifying an effectless computation.

```
Definition return: A -> PrePost A:= fun a => (True,fun a' => a' = a)
```

The first component describes a precondition sufficient for an effectless computation returning a to run, hence it is trivial. The second component is similar to the `return` operator of the predicate monad. It describes a predicate on return values that should hold after the execution of an effectless computation... solely returning a. For the bind operator we have:

```
Definition bind: PrePost A -> (A -> PrePost B) -> PrePost B:=
fun w f =>
  let (prew,postw) = w in
  let (fun a => (pref a,postf a)) = f in  (*pseudo eta-expansion*)
  let preRes = prew /\ forall (a:A), postw a -> pref a in
  let postRes = fun b => exists (a:A), postw a /\ postf a b in
  (preRes,postRes)
```

Assume w is a specification assigned to a computation m (in the sense: properly describing the behaviour of m) and m returns a value a. Assume $f(a)$ is a specification assigned to a computation $g(a)$. What could be a specification assigned to the composition `bind m g`? Note that the word *assigment* will gain a formal sense in the following section.

- The precondition `preRes` should be a property sufficient to ensure the proper execution of the whole computation `bind m g`. In particular in order to run m we know that at least `prew` has to hold. Moreover we want g to run afterwards so we ask that if a is a result value of m (thus something respecting `postw`), then $g(a)$ can safely be executed, i.e. `pref a` holds.
- The postcondition `postRes` of the overall computation `bind m g` describes what we know about values returned by it. In particular we know that m ran and returned some value a satisfying `postw a`. Moreover, b is the value returned by $g(a)$ so it satisfies `postf a b`.

The preorder structure is given by pointwise implication for postconditions, and *reversed* pointwise implication for preconditions:

$$(p_1, q_1) \leq (p_2, q_2) \quad \text{iff} \quad \forall (a : A), q_1(a) \to q_2(a) \wedge p_2 \to p_1$$

The intuition is that a strong specification is difficult to prove, hence it carries either a weak precondition (few hypotheses) or a strong postcondition.

Eventhough pre/postconditions are intuitive objects, one often rather considers weakest preconditions (or equivalently strongest postconditions, which are not discussed here). Compare to $\mathcal{P}$rePost, the following monad is strictly more expressive and has better computational properties, namely thanks to the absence of existential quantification in its `bind` operator. See [Lei05].

Backward predicate transformers. As discussed in §1 it is possible to specify programs by means of predicate transformers instead of pre/postconditions. Let us consider the continuation monad $\mathrm{Cont}_{\mathbb{P}}(A) = (A \to \mathbb{P}) \to \mathbb{P}$ discussed in §2.1. The type $\mathbb{P}$ is an impredicative universe for propositions (in Coq it is noted `Prop`). A point $w : W\,A$ of such a monad sends postconditions to preconditions, hence the name "backward predicate transformer". However $Cont_{\mathbb{P}}$ does not form a specification monad as is. Indeed its `bind` operator is not monotonic. In order to recover monotonicity we restrict ourselves to monotonic predicate transformers, i.e. the points $w : Cont_{\mathbb{P}}(A)$ preserving the pointwise implication. This restricted variant is called $\mathrm{MonCont}_{\mathbb{P}}$ or also W^{Pure} and is abusively identified with $\mathrm{Cont}_{\mathbb{P}}$ in what follows. Regarding the preorder structure, since backward predicate transformers yield preconditions, we take the order to be the reversed pointwise implication. In other words for every $w_1, w_2 : W\,A$,

$$w_1 \leq w_2 \quad \text{iff} \quad \forall (p : A \to \mathbb{P}), w_2(p) \to w_1(p)$$

3.2 Effect observations

The dm4all framework is based on the observation that specifications of effectful programs form a monad, like effectful programs do. Since both computations and specifications can be composed via the `bind` operators of the respective monads, it is natural to consider interpretations of effectful programs into specifications (that is, predicate semantics) compatible with the composition. More precisely, an effect observation between a computational monad M and a specification monad W is simply defined to be a monad morphism $\theta : M \to W$. Specifically, θ has to preserve the `return` operator and the `bind` operator (see end of §2.1).

Semantics for effectful program logics. Once a concrete instance $\theta : M \to W$ of the framework is fixed, it is possible to confer semantics to program logics dealing with the effect modelled by M. In general a (effectful) program logic is a syntactic tool for verifying that a certain (effectful) program m meets its specification w. The framework translates this condition into the following simple inequality:

$$\theta(m) \leq w$$

Concretely w is provided by the user and is the expected specification of the program whereas $\theta(m)$ is some kind of "most precise" specification, a complete logical description of the program m.

For instance, consider the case of a Hoare-like logic $\mathcal{H}$ with basic judgments $\vdash \{P\}m\{Q\}$. In order to interpret those judgments using the dm4all famework, we first choose the specification monad to be $W = \mathcal{P}\mathrm{rePost}$. We then fix a map $\theta : M \to \mathcal{P}\mathrm{rePost}$ describing a verification methodology and write $\theta(m) = (\mathrm{pre}_\theta(m), \mathrm{post}_\theta(m))$. Let $w = (P, Q)$ be a user-provided specification and m the effectful program the user whishes to certify. The framework provides the following semantics for $\mathcal{H}$:

$$\begin{aligned} \models \{P\}m\{Q\} \quad &\text{iff} \quad \theta(m) \leq w \\ &\text{iff} \quad P \to \mathrm{pre}_\theta(m) \wedge \mathrm{post}_\theta(m) \to Q \end{aligned}$$

On one hand this semantics for $\mathcal{H}$ is complete precisely when the sequents $\vdash \{\mathrm{pre}_\theta(m)\}m\{\mathrm{post}_\theta(m)\}$ are provable in $\mathcal{H}$ for every m. On the other hand program logics such as $\mathcal{H}$ are precisely designed in order to be sound with respect to the following operational-flavored semantics:

$$\models_{\mathrm{op}} \{P\}m\{Q\} \quad \text{iff} \quad \text{"if P holds and m runs then Q holds"}$$

In general the semantics provided by the dm4all framework for program logics will be complete, or will be forced to be complete by adding an extra completeness axiom in $\mathcal{H}$. In other words

sequents of the form $\vdash \{\mathrm{pre}_\theta(m)\}m\{\mathrm{post}_\theta(m)\}$ will hold provably, or axiomatically. Hence thanks to the soundness of $\mathcal{H}$ w.r.t $\models_{\mathrm{op}}$ and the completeness of the θ semantics for $\mathcal{H}$ we will be in position to assume that "if $\mathrm{pre}_\theta(m)$ holds and m runs then $\mathrm{post}_\theta(m)$ holds". This observation formally reflects the fact that we designed the framework thinking "operationally". In particular the various `bind` operations of the specification monads defined in §3.1 are of course inspired by an operational point of view.

Effect observations for exceptional programs. Recall that the computational monad $\mathrm{Exc}(A) = A + E$ allows to model programs raising exceptions, see §2.1. There are several possiblities for assigning to such effecful programs a predicate semantics within the dm4all framework.

The first solution is to consider the specification monad $W^{\mathrm{Exc}}(A) = ((A + E) \to \mathbb{P}) \to \mathbb{P}$ together with the effect observation $\theta^{\mathrm{Exc}} : \mathrm{Exc} \to W^{\mathrm{Exc}}$ defined by `theta m:= fun post => post m`. The monad W^{Exc} is expressive enough to let the user express properties about the potential exceptions arising in the program.

As a simple-minded example, think of the following program:

```
Definition m: nat -> nat -> Exc nat:= fun a b =>
  if b != 0 then a/b
  else raise DivisionError.
```

We would like to certify the (trivial) fact that if $b = 0$ then this programs raises a `DivisionError`. In terms of pre/post conditions we wish to prove that m meets the specification w given by `(b = 0,fun m' => m'=raise DivisionError)`. Specifications in W^{Exc} are not expressed as pre/postconditions though and we first need to perform a translation from pre/post into backward predicate transformers $w : W^{\mathrm{Exc}}$, translation described in [MAA+19]. Hence w can be written as:

```
Definition (w: WExc A):= fun (p: A+E -> P) =>
  b = 0 /\ (forall (m': A+E), m' = raise DivisionError ->p m').
```

Recall from the last paragraph that m meets its specification w according to the verification methodology θ precisely when $\theta(m) \leq w$. This amounts to show the following fact for every predicate $p : A + E \to \mathbb{P}$:

$$w\,p \implies \theta(m)\,p \quad \text{i.e.}$$
$$(b = 0) \wedge (\forall (m' : A + E), m' = \texttt{raise DivisionError} \to p(m')) \implies p(m)$$

which is easy. Indeed suppose the hypothesis$w\,p$. We know that $b = 0$ so our program m simplifies into `raise DivisionError`. But the second part of $w\,p$ states precisely that p holds for this program. And so p is true and the verification condition is discharged : $\theta(m) \leq w$.

Another possibility is to consider the specification monad $\mathrm{MonCont}_{\mathbb{P}}(A) = (A \to \mathbb{P}) \to \mathbb{P}$ together with the effect observation $\theta^{\perp} : \mathrm{Exc} \to \mathrm{MonCont}_{\mathbb{P}}$ defined as $\theta^{\perp}$ `(Some a) = fun p => p a` and $\theta^{\perp}$ `(raise e) = fun p => False`. This effect observation provides semantics for program logics achieving total correctness (see 2.4). Intuitively the precondition always ensures the correct execution of the program because it is false precisely when the program fails. Dually we can define $\theta^{\top}$ by changing `theta (raise e) = fun p => True` and this effect observation is morally a model of partial correctness program logics for exceptional computations.

Effect observations for nondeterminism. A similar story holds for the nondeterministic computations modelled by the powerset monad Ndet. It is indeed possible to define two dual effect observations $\theta^{\forall}, \theta^{\exists} : \mathrm{Ndet} \to \mathrm{MonCont}_{\mathbb{P}}$. The first one $\theta^{\forall}$ models "demonic verification" of nondeterministic programs wheras the second models "angelic verification". Here is the definition of $\theta^{\forall}$:

```
Definition θ∀: Ndet A -> MonCont A:= fun m =>
match m with
  |[] => (fun p => True)
  |hd::tl => (fun p => p hd /\ theta tl p)
end.
```

Effect observations for stateful programs. In this case we consider a specification monad of the form $W^{\mathrm{St}}(A) = (A \times S \to \mathbb{P}) \to S \to \mathbb{P}$. This monad allows postconditions to mention return values $a : A$ and final states $s_1 : S$, wheras preconditions can depend on initial states $s_0 : S$. An effect observation $\theta^{\mathrm{St}} : \mathrm{St} \to W^{\mathrm{St}}$ is given by:

```
Definition theta: St A -> WSt A:= fun m p s0 =>p (m s0).
```

Using this effect observation, it is for example possible to certify the behaviour of the `swap` program described in §2.1.

3.3 Dijkstra monads

One of the achievements of the dm4all framework is to decouple effectful computations on one side from their specifications on the other. This loose coupling allows to choose the best kind of specifications and interpretation for the verification problem at hand: think of the angelic/demonic effect observations for the nondeterminism effect for instance (see §3.1). Nevertheless an alternative intrinsic point of view was originally first developed: Dijkstra monads. .

Given a specification monad W, a Dijkstra monad is a family of types $\mathcal{D}\,A\,w$ indexed by a value type A and a specification w living in W. It exhibits a similar interface to a regular computational monad, namely:

- It posseses a `return` operator, of type `retD: (a:A) -> D A (retW a)` for any type A.
- It has of course a `bind` operator of the following type

  ```
  bindD: (c: D A wc) -> (f: (x:A) -> D B wf(x)) -> D B (bindW wc wf)
  ```

  Notice that this definition relies heavily on the dependant nature of the type theory at hand (in our case somtehing similar to Coq). Let c be a computation correctly specified by w_c. Let f be a computation depending on $x : A$ correctly specified by w_f. The bind operator allows you to compose those two effectful computations into a computation correctly specified by $\mathrm{bind}^W\, w_c\, w_f$.
- These two operators have to obey the equivalent of the usual monadic laws.
- Besides the usual monadic machinery replicated here along the specifications, Dijkstra monad also posses a weakening operator: `wkn: (w1 <= w2) -> (D A w1) -> (D A w2)` obeying various laws described in [MAA+19].

3.4 The equivalence

We define here the category of Dijkstra monads together with the category of monadic relations. There is an adjunction between these two notions, and restricting this adjunction to monad morphisms instead of monadic relations yield an equivalence of categories.

The category $\mathcal{DM}$on. The objects of this category are defined to be pairs $(W, \mathcal{D})$ such that the specification monad of $\mathcal{D}$ is W. A Dijkstra monad morphism $\Theta^{\mathcal{D}} : \mathcal{D}_1 \to \mathcal{D}_2$ over a specification monad morphism $\Theta^W : W_1 \to W_2$ is a collection of maps

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

compatible with the return and bind operators.

The adjunction. We give an idea of the correspondance described in [MAA+19].

$$\mathcal{DM}\text{on} \underset{\text{fib}}{\overset{\int}{\leftrightarrows}} \mathcal{M}\text{on}\mathcal{R}\text{el}$$

First start with a monadic relation $\theta : M \to W$. The only difference between effect observations and monadic relations is that monadic relations can lack functionality. We define a Dijkstra monad $\mathcal{D}$ over the specification monad W by the following:

$$\mathcal{D}\, A\, w = \{m : M\, A \mid \theta(m) \leq w\}$$

As stated earlier, $\mathcal{D}\, A\, w$ is indeed the type of all effectful computations correctly specified by w.

In the other direction let $\mathcal{D}$ be a Dijkstra monad over some specification monad W. We first need to build a computational monad M. Define M as the following sigma type: $M\, A = \sum_{w:W\, A} \mathcal{D}\, A\, w$. The first projection maps each element to its index w and defines in fact a monad morphism $M \to W$, that is, an effect observation.

4 Relational verification

The dm4all framework developped in the last section allows to interpret arbitrary effectful programs $m : M\, A$ by predicates $w : W\, A$. On top of this semantics, it is then possible to build semantics for a wide range of program logics as explained in 3.2.

Those program logics are tools to check that a given single program meets its specification. In fact there exists in the literature an alternative class of program logics known as *relational program logics*. Instead of analysing one single program, their purpose is to compare two different programs and to show that they are related. For instance the user of such logics could desire to prove that two implementations of a given interface are observationally equivalent [cCLRR16, KTL09, GS10, BAF08, CCcCK16, TSKB18, WDLC18, Yan07, HDNV12, HNDV14], meaning that on same inputs they always return the same outputs, regardless of perfomance issues. *Non-interfence* is another problem tackled by relational program logics [NBG13, CS10, SM03, AGH+17, BEG+19, SD16, BNN16]. In this case the goal is to show that private, secret inputs will not interfer with public outputs of a program otherwise leaking some private information. Amongst many other applications, relational logics also allow to perform cost analyses [ÇBG+17, QGG19, RBG+18], i.e. to compare the ressource usage of the two considered programs.

The relational framework designed in [MHRM19] aims to recover the many insights gained in unary verification via dm4all framework, in a relational perspective. In particular this relational framework keeps the effect abstract, wheras existing program logics always focus on a given set of effects and provide a single verification methodology. Besides, the framework theoretically enables the comparison of programs with different effects.

In this section we introduce the above-mentionned semantic relational framework and show that it is possible to interpret Relational Hoare Logic (RHL) in it [Ben04]. RHL is one of the first relational program logic to have emerge and is a tool for performing various static analyses such as sound optimizing transformations for imperative programs (dead code elimination, program slicing, ...)

4.1 The relational framework

In its simplest form the relational framework is similar to the dm4all framework. The main difference is that monads become relative monads (see 2.2).

Relational effectful computations. Let M_1 and M_2 be monads on $\mathcal{C}$, accounting for the effects of the first and second program we wish to compare (equivalently the left and right programs). The category $\mathcal{C}$ can be tought of as `Type`. We obtain a (relative) monad by applying the product:

$$M_1 \times M_2 : \mathcal{C}^2 \to \mathcal{C}^2$$

This monad simply sends a pair of type (A_1, A_2) to the pair of types $(M_1(A_1), M_2(A_2))$. Intuitively a point of this relative monad is a pair of effectful programs (m_1, m_2).

Relational specifications. On another hand we modelize relational specifications by an ordered relative monad $W : \mathcal{C}^2 \to \mathcal{C}$ over the product functor. We call such relative monads relational specification monads. By definition they have

- a preorder structure on $W(A, B)$ for every types A, B.
- a return operator $\text{ret}_{AB} : A \times B \to W(A, B)$.
- a bind operator $\text{bind}_{AB} : W(A_1, B_1) \to (A_1 \times B_1 \to W(A_2, B_2)) \to W(B_1, B_2)$ wich is monotonic in both arguments.

For instance, the relational equivalent of W^{Pure} is $W^{\text{Pure}}_{\text{rel}}(A, B) = (A \times B \to \mathbb{P}) \to \mathbb{P}$. Similarly $W^{\text{St}}_{\text{rel}}(A, B) = (A \times B \times S^2 \to \mathbb{P}) \to S^2 \to \mathbb{P}$ is the relational equivalent of the specification monad W^{St} used to verify stateful computations in the unary case. A point $w : W^{\text{St}}_{\text{rel}}(A, B)$ is a relational backward predicate transformer. It maps postconditions depending on result values and final states, to preconditions depending on two initial states.

Relational effect observations. A relational effect observation is defined to be a relative monad morphism between a relational computation monad $M_1 \times M_2$ and a relational specification monad W. Specificall it is a natural transformation filling the following square:

$$\begin{array}{ccc} \mathcal{C}^2 & \overset{Id}{\underset{M_1 \times M_2}{\rightrightarrows}} & \mathcal{C}^2 \\ {\scriptstyle Id}\downarrow & & \downarrow{\scriptstyle \text{prod}} \\ \mathcal{C}^2 & \underset{\text{prod}}{\overset{W}{\rightrightarrows}} & \mathcal{C} \end{array}$$

Equivalently a relational effect observation is a collection of maps $\theta_{AB} : M_1\, A \times M_2\, B \to W(A, B)$ preserving the return and the bind operations of the two relative monads.

Semantics for relational program logics. In the very same way effect observations allow the dm4all framework to provide semantics for (unary) program logics 3.2, our relational framework can provide semantics for relational program logics, such as RHL as we will investigate. The idea is the following. Let m_1, m_2 in M_1, M_2 be two effectful programs we wish to compare. Let W be a suited relational specification monad and $\theta_{\text{rel}} : M_1 \times M_2 \to W$ a relational effect observation. Let $w : W(A, B)$ a user-provided relational specification. Intuitively w encodes the expected relation between the two programs (for example "they yield the same output"...). The two programs m_1 and m_2 are related at w if

$$\theta_{\text{rel}}(m_1, m_2) \leq w$$

The next subsections are dedicated to embed RHL into our relational framework. We proceed in several steps :

- We first devise a *unary* effect observation to interpret monadic while programs. This requires the definition of a monad accouting for programs with while loops. The effect observation is then defined via domain-theoretic arguments.

  The unary effect observation allows in turn the construction of a relational effect observation θ_{rel} to compare monadic while programs (like in RHL). This work is done in 4.2.

- Next in 4.3 we sketch a relational program logic $\mathcal{H}_{\text{rel}}$ sound and complete with respect to the relational semantics defined in 4.2. Obviously the obtained logic is similar to RHL.

- This similarity is captured by a syntactic translation τ from Benton's RHL sequents to sequents of the devised sound logic $\mathcal{H}_{\text{rel}}$. The translation is defined in 4.4

- Finally we prove that τ is an embedding, meaning that translated demonstrable sequents of RHL are demonstable sequents of $\mathcal{H}_{\text{rel}}$. Of course for this it suffices to show that translated RHL rules are admissible in $\mathcal{H}_{\text{rel}}$. This work is done in 4.5.

Schematically we work on the following bottom and right implications:

$$\begin{array}{ccc} \models_{\text{RHL}} - & & \models_{\theta_{\text{rel}}} \tau(-) \\ \text{sound}\uparrow & & \uparrow\text{sound} \\ \vdash_{\text{RHL}} - & \xrightarrow[\text{embedding}]{} & \vdash_{\mathcal{H}_{\text{rel}}} \tau(-) \end{array}$$

Since $\mathcal{H}_{\text{rel}}$ is sound with respect to our semantic framework, and τ is an embedding, we obtain the soundness of RHL. In other words RHL is interpretable in the relational semantic framework. Interestingly the semantics obtained via θ_{rel} is a partial correctness one wheras the original RHL semantics mixes partial and total correctness (that is why there is no top arrow in the above diagram).

4.2 Semantics for monadic while programs

We begin by studying a monadic syntax and a predicate semantics for while programs. This allows a partial correctness verification of while programs.

The Imp monad. A while program can read and write a state $s : S$ and can repeat instructions via while loops. Here we choose do-while loops because they ease the definition of the subsequent effect observation.

```
Inductive Imp (A:Type) :=
  |Ret : A -> Imp A
  |DoWhile : Imp bool -> Imp A -> Imp A
  |Get : (S -> Imp A) -> Imp A
  |Put : S -> Imp A -> Imp A.
```

Let us unformally explain the intended meaning of this syntax. A formal semantics will be given right after, under the form of a (unary) effect observation. The first argument of the `DoWhile` constructor is meant to be the body of the loop. Since we want to work with do-while loops, this body is executed at least once. Then its returned boolean value determines wheter it should be executed more than once. The second argument of `DoWhile` is supposed to be the continuation, i.e. the while-program executed after the loop, if the loop finishes. The `Get` constructor has one single argument, say k. To execute a `Get` instruction one has to fetch the state s and branch on it with k, that is to run $k\,s$. Finally the `Put` constructor expects a state in order to overwrite the current state, and a continuation which is executed after the overwriting.

A unary effect observation. Now that we have defined the `Imp` monad, we can devise a partial correctness interpretation $\theta^{\mathrm{Part}} : \mathrm{Imp} \to W^{\mathrm{St}}$ into stateful specifications. Here is its definition:

```
Fixpoint theta (ma : Imp A) : WSt A := match ma with
  |Ret a => retWSt a
  |Put (s':S) (k:Imp A) => (fun p s => theta k p s')
  |Get (k : S-> Imp A) => (fun p s => theta (k s) p s)
  |DoWhile (body : Imp bool) (k : Imp A) =>
    fix( loopBody ) ; theta k
```

The function `loopBody` used in the `DoWhile` case is a nested function defined as follows:

```
loopBody : WSt bool -> WSt bool := fun w =>
  bind b <- theta body in
  if b then w else (retWSt false)
```

This function intuitively executes the `body` once and then run its parameter if the resulting boolean is true. It returns false otherwise to avoid further loops.

How is defined the fixpoint combinator appearing in the last line of the definition of θ^{Part}? Why does it exist ? We are here making use of the pointwise domain structure of $W^{\mathrm{St}}\, A = (A \times S \to \mathbb{P}) \to S \to \mathbb{P}$. Recall that its minimum is the predicate transformer noted $\dot{\top}$ mapping every postcondition and initial state to `True`, and of course its maximum is $\dot{\bot}$. Since $W^{\mathrm{St}}\mathbb{B}$ is a dcpo (2.3), it is equipped with a constructive fixpoint combinator mapping *Scott-continuous* functionals $F : W^{\mathrm{St}}\mathbb{B} \to W^{\mathrm{St}}\mathbb{B}$ to their least fixpoint computed by

$$\mu\, F = \sup_{n \in \mathbb{N}} F^n(\dot{\top})$$

So for theta to be well defined, `loopBody` should be Scott-continuous. In fact we can prove that

- For every $(c : \mathrm{Imp}\, A)$, if $\theta^{\mathrm{Part}}(c)$ is a Scott-continuous predicate transformer (preserving suprema on its postcondition parameter) then loop_c is Scott-continuous as well.
- And in fact $\theta(c)$ is always Scott continuous.

Thus loop_c is Scott continous and θ^{Part} is well defined. Moreover we can reason by induction in order to show that θ^{Part} is indeed a monad morphism.

θ^{Part} is a partial correctness interpretation, that is, we can morally assume the termination of programs when verifying them according to θ^{Part}. In particular the verification of non-terminating programs should be straightforward (up to determining if the considered computation terminates of course). To illustrate this observation, we analyse the specification returned by θ^{Part} when it is evaluated in some infinite loop. Define `ma := DoWhile (ret true) (ret tt)` where `tt` stands for the unique inhabitant of the unit type. The nested function `loopBody` is just the indentity function on $W^{\mathrm{St}}\mathbb{B}$ in this case. Hence its least fixpoint is $\dot{\top}$ and we have $\theta^{\mathrm{Part}}(ma) = \dot{\top}; \texttt{ret ()} = \dot{\top}$. Now every specification $w : W^{\mathrm{St}}\, 1$ correctly specifies ma because $\theta^{\mathrm{Part}}(ma) = \dot{\top} \leq w$.

Dually, choosing a greatest fixpoint instead of a least fixpoint leads to the definition of a total correctness interpretation θ^{Tot}. In this case the verification of programs implicitely entails their termination.

A relational effect observation. In order to define a relational effect observation $\theta^{\mathrm{Part}}_{\mathrm{rel}} : \mathrm{Imp} \times \mathrm{Imp} \to W^{\mathrm{St}}_{\mathrm{rel}}$ we first post-compose our unary effect observation θ^{Part} with two monad morphisms ν_1 and ν_2 to get $\theta^{\mathrm{Part}}_1 : \mathrm{Imp} \to W^{\mathrm{St}}_{\mathrm{rel}}(-, 1)$ and $\theta^{\mathrm{Part}}_2 : \mathrm{Imp} \to W^{\mathrm{St}}_{\mathrm{rel}}(1, -)$. Here is the definition of ν_1:

```
Definition nu1 : WSt A -> WrelSt A 1:= fun w  =>
  fun (p : A * S *S -> Prop) (s1 s2 : S) =>
  w (fun a s => p a s s2) s1
```

Then there are two possibilities for coercing a pair of specifications living in $W^{\mathrm{St}}_{\mathrm{rel}}(A,1) \times W^{\mathrm{St}}_{\mathrm{rel}}(1,B)$ into a single relational specification $W^{\mathrm{St}}_{\mathrm{rel}}(A,B)$. In one case `lrRun` morally executes the left program first and then the right program. The other map does the converse. Here is a diagram of the situation:

$$\begin{array}{ccc}
\mathrm{Imp}\,A \times \mathrm{Imp}\,B & \xrightarrow{\theta^{\mathrm{Part}} \times \theta^{\mathrm{Part}}} & W^{\mathrm{St}}\,A \times W^{\mathrm{St}}\,B \\
\downarrow{\scriptstyle \theta_{\mathrm{rel}}} & & \downarrow{\scriptstyle \nu_1^A \times \nu_2^B} \\
W^{\mathrm{St}}_{\mathrm{rel}}(A,B) & \xleftarrow{\mathrm{lrRun}} & W^{\mathrm{St}}_{\mathrm{rel}}(A,1) \times W^{\mathrm{St}}_{\mathrm{rel}}(1,B)
\end{array}$$

$$\mathrm{rlRun} : W^{\mathrm{St}}_{\mathrm{rel}}(A,1) \times W^{\mathrm{St}}_{\mathrm{rel}}(1,B) \to W^{\mathrm{St}}_{\mathrm{rel}}(A,B)$$

In fact this diagram commutes and defines a relational effect observation $\theta^{\mathrm{Part}}_{\mathrm{rel}}$ abbreviated θ_{rel}.

4.3 A sound relational program logic

We devise here a relational program logic $\mathcal{H}_{\mathrm{rel}}$ whose (contextless) sequents are of the form $\vdash c_1 \sim c_2 \{w\}$ for $c_1 : \mathrm{Imp}\,A$ and $c_2 : \mathrm{Imp}\,B$. This sound logic $\mathcal{H}_{\mathrm{rel}}$ is defined in order to ease the proof that RHL is a sound logic with respect to the relational semantic θ_{rel}. $\mathcal{H}_{\mathrm{rel}}$ is embedded in reasonnable type theory (Coq) meaning that $\mathcal{H}_{\mathrm{rel}}$ sequents can have a context, i.e. they can mention non constant terms. As explained in [MHRM19] it is possible to cook a sound relational program logic from a given relational effect observation. We use θ_{rel} as this effect observation and call $\mathcal{H}_{\mathrm{rel}}$ the resulting logic.

$\mathcal{H}_{\mathrm{rel}}$ is built upon the following principles:

- Several rules are dedicated to various constructs of the ambient type theory, such as the if-rule:

$$\frac{\vdash \mathtt{if}\ b\ \mathtt{then} \sim c_2 \ \left\{ w^{\top} \right\}\ \mathtt{else} \sim c_2 \ \left\{ w^{\bot} \right\}}{\vdash c_1 \sim c_2 \ \left\{ \mathtt{if}\ b\ \mathtt{then}\ w^{\top}\ \mathtt{else}\ w^{\bot} \right\}}$$

  The fact that `b = true` to the left of the premisse can be used to simplify the programs c_1 and c_2 accordingly.

- $\mathcal{H}_{\mathrm{rel}}$ exhibits three generic monadic rules. One for ret, one for bind and a weakening rule:

$$\textsc{Ret}\ \frac{a_1 : A_1 \qquad a_2 : A_2}{\vdash \mathtt{ret}^{\mathrm{M}_1}\, a_1 \sim \mathtt{ret}^{\mathrm{M}_2}\, a_2 \ \left\{ \mathtt{ret}^{\mathrm{W}}\,(a_1,a_2) \right\}} \qquad \textsc{Weaken}\ \frac{\vdash c_1 \sim c_2 \ \{ w \} \qquad w \leq w'}{\vdash c_1 \sim c_2 \ \left\{ w' \right\}}$$

$$\textsc{Bind}\ \frac{\vdash m_1 \sim m_2 \ \{ w^m \} \qquad \forall a_1, a_2 \vdash f_1\, a_1 \sim f_2\, a_2 \ \left\{ w^f\,(a_1,a_2) \right\}}{\vdash \mathtt{bind}^{\mathrm{M}_1}\, m_1\, f_1 \sim \mathtt{bind}^{\mathrm{M}_2}\, m_2\, f_2 \ \left\{ \mathtt{bind}^{\mathrm{W}_{\mathrm{rel}}}\, w^m\, w^f \right\}}$$

  The bind rule allows the user to break his monadic programs into simpler constituents, and continue the proof on those consituents.

- Some rules are specific to the effect at hand (here state + non-termination) and permit to relate two effectful operations such as two `Put`'s for example.

- Finally we add an add hoc completeness rule:

$$\vdash c_1 \sim c_2 \{\theta_{\mathrm{rel}}(c_1,c_2)\}$$

This rule artificially ensures the completeness of the θ_{rel} semantics for $\mathcal{H}_{\text{rel}}$. In other words, if $\theta_{\text{rel}}(c_1, c_2) \leq w$ then $\vdash c_1 \sim c_2 \{w\}$ is provable. We allow ourselves to impose such a coarse rule because we just see $\mathcal{H}_{\text{rel}}$ as a mean to guide the computation of the θ_{rel} semantics. In other words $\mathcal{H}_{\text{rel}}$ is a tool to prove that RHL is sound for the θ_{rel} semantics.

4.4 The translation

We first translate Benton's while-language constructs into our monadic while-language Imp 1. Indeed the programs described in [Ben04] change the global state, i.e. overwrite memory cases but do not return any value, hence the unit type 1. We translate full sequents of RHL afterwards. The following table describes the syntax of Benton's while programs:

$$\begin{aligned} \texttt{int exp} \ni E &:= n \mid X \mid E \,\texttt{iop}\, E \\ \texttt{bool exp} \ni B &:= b \mid E \,\texttt{bop}\, E \mid \texttt{not}\, B \mid B \,\texttt{lop}\, B \\ \texttt{com} \ni C &:= \texttt{skip} \mid X \texttt{:=} E \mid C; C \mid \texttt{if}\, B \,\texttt{then}\, C \,\texttt{else}\, C \mid \texttt{while}\, B \,\texttt{do}\, C \end{aligned}$$

n is an integer, X is an integer variable, `iop` is an integer operator such as $+$, b is a boolean value, `bop` is a comparison operator such as $=$ or $\leq$, and `lop` is a logical operator such as $\wedge$.

In the following, The set of states S is chosen to be the set of functions mapping integer variables to integer values (S is the collection of all possible valuations). For $s : S$ we note $s(X)$ the value attributed to X according to s, and $s[X := v]$ the global state uptated with the value v at location X.

Integers and boolean expressions. A constant integer n is translated to $\tau(n) = \text{ret}\, n : \text{Imp}\, \mathbb{N}$. A variable X is translated by `Get k where k = fun s => ret s(X).` Moreover $E_1 \,\texttt{iop}\, E_1$ is translated inductively into `bind e1 <- tau(E1) in bind e2 <- tau(E2) in e1 iop e2`. The translation of boolean expressions uses similar ideas. We also note $s(B)$ for the value of B according to the state s.

Commands. The translation τ maps commands to points of Imp 1. `skip` is translated into `ret ()`. An assignment $X\texttt{:=}E$ is translated by:

```
bind e <- tau(E) in
Get k where k = fun s =>
Put s[X:=e]
```

So "at run time", e is the value of E according to the current state and this current state is updated at location X as expected for an assignement. The sequence of two programs $C_1; C_2$ is translated into a monadic sequence $\tau(C_1); \tau(C_2)$. The if statement $\texttt{if}\, B \,\texttt{then}\, C_1 \,\texttt{else}\, C_2$ is translated into `bind b <- tau(B) in if b then C1 else C2` and finally the while command $\texttt{while}\, B \,\texttt{do}\, C$ is translated into

```
DoWhile
(bind b <- tau(B) in if b then C else ret false)
ret ()      (*trivial continuation*)
```

Sequents. RHL sequents are of the form $\vdash C \sim C' : \phi \Rightarrow \psi$, where C and C' are commands and ϕ and ψ are pre and postconditions. Since commands do not return values, those predicates depend solely on intial or final states. For example ϕ could be $X\langle 1 \rangle = Y\langle 2 \rangle$ relating a location of the left program C to a location of the right program C'. In order to translate Benton's specifications into relational backward predicate transformers, i.e. points of $W^{\text{St}}_{\text{rel}}$ we begin by identifying ϕ with a point of $S^2 \to \mathbb{P}$ and ψ with a point of $1 \times 1 \times S^2 \to Prop$ (a post-condition

usually mentions values). The latter type is itself identified with $S^2 \to \mathbb{P}$. Then we use a translation of pre/postconditions into backward predicate transformers wich maps $(pre, post)$ to

$$\lambda\varphi\ (s_1^i, s_2^i).\ pre(s_1^i, s_2^i) \wedge \forall a_1, a_2, s_1^f, s_2^f.post\ ((a_1, s_1^f), (a_2, s_2^f)) \Rightarrow \varphi\ ((a_1, s_1^f), (a_2, s_2^f))$$

This translation is inspired by the simpler translation of pre/postconditions into backward predicate transformers in the unary setting:

$$\lambda\varphi\, s^i.\ pre(s^i) \wedge \forall a, s^f.post(a, s^f) \Rightarrow \varphi(a, s^f)$$

In short the translation τ maps RHL sequents of the form $\vdash C \sim C' : \phi \Rightarrow \psi$ into $\mathcal{H}_{\text{rel}}$ sequents of the form $\vdash \tau(C) \sim \tau(C')\,\{\tau(\phi \Rightarrow \psi)\}$.

4.5 Admissiblity

In order to show that RHL is sound w.r.t the monadic predicate semantics θ_{rel}, the next step is to translate RHL rules into actual $\mathcal{H}_{\text{rel}}$ "goals". If we manage to show that those translated rules are admissible in $\mathcal{H}_{\text{rel}}$ (that is, τ is an embedding), since $\mathcal{H}_{\text{rel}}$ is sound with respect to our relational semantic framework we obtain soundness of RHL with respect to the monadic relational semantic framework (see the diagram at the end of 4.1). In what follows we explain the admissibility of a few tranlsated RHL rules.

The skip rule.

$$\vdash \texttt{skip} \sim \texttt{skip} : \phi \Rightarrow \phi$$

The translated axiom is proved in the following way in $\mathcal{H}_{\text{rel}}$:

$$\dfrac{\dfrac{}{\vdash \text{ret}\,() \sim \text{ret}\,()\,\{\text{ret}^W\,()\,()\}}\ \text{RET} \qquad \text{ret}^W\,()\,() \leq \tau(\phi \Rightarrow \phi)}{\vdash \text{ret}\,() \sim \text{ret}\,()\,\{\tau(\phi \Rightarrow \phi)\}}\ \text{WKN}$$

To discharge the right premisse, recall that both $\text{ret}^W\,()\,()$ and $\tau(\phi \Rightarrow \phi)$ are stateful backward predicate transformers, i.e. members of the monad $W^{\text{St}}_{\text{rel}}(1,1) = (S^2 \to \mathbb{P}) \to (S^2 \to \mathbb{P})$. Since the order is defined as the reversed pointwise implication, we start by instantiating both sides transformers and compare them afterward. Let $p : S^2 \to \mathbb{P}$ and $(s_1, s_2) : S^2$. On one hand we have $\text{ret}^W\,()\,()\,p\,(s_1, s_2) = p(s_1, s_2)$. On the other hand we have

$$\tau(\phi \Rightarrow \phi)\,p\,(s_1, s_2) = \phi(s_1, s_2) \wedge \forall(s_1^f, s_2^f).\phi(s_1^f, s_2^f) \to p(s_1^f, s_2^f)$$

The fact that $\text{ret}^W\,()\,() \leq \tau(\phi \Rightarrow \phi)$ in p, s_1, s_2 is now trivial. Destruct the conjunction, apply the right side and get p.

The if rule.

$$\dfrac{\vdash C \sim C' : \phi \wedge B\langle 1\rangle \wedge B'\langle 2\rangle \Rightarrow \phi' \qquad \vdash D \sim D' : \phi \wedge \texttt{not}(B\langle 1\rangle \vee B'\langle 2\rangle) \Rightarrow \phi'}{\vdash \texttt{if}\,B\,\texttt{then}\,C\,\texttt{else}\,D \sim \texttt{if}\,B'\,\texttt{then}\,C'\,\texttt{else}\,D' : \phi \wedge B\langle 1\rangle = B'\langle 2\rangle \Rightarrow \phi'}$$

Note $c_1 = \texttt{if}\,B\,\texttt{then}\,C\,\texttt{else}\,D$ and $c_2 = \texttt{if}\,B'\,\texttt{then}\,C'\,\texttt{else}\,D'$. We begin by observing that the translated command $\tau(c_i)$ is a bind in Imp. So we would like to weaken our specification accordingly into some $w \leq \tau(\phi \wedge B\langle 1\rangle \wedge B'\langle 2\rangle \Rightarrow \phi')$ such that $w = \text{bind}\,w_m\,w_f$ in $W^{\text{St}}_{\text{rel}}$. This would allow us to first apply the weakening rule, followed by a bind rule. We define w by

```
w :=
let b0,b0' = (fun p s1 s2 => p  s1(tau B)  s1 s2(tau B') s2) in
match b0,b0' with
  |true, true => tau(leftSpec)
  |false, false => tau(rightSpec)
  |true, false => wtf
  |false, true => wft
```

where `leftSpec` is $\tau(\phi \wedge B\langle 1\rangle \wedge B'\langle 2\rangle \Rightarrow \phi')$, and `rightSpec` is $\tau(\phi \wedge \texttt{not}(B\langle 1\rangle \vee B'\langle 2\rangle) \Rightarrow \phi')$. The predicate transformers `wtf` and `wft` are let as is for the moment. Why is w weaker than the final relational specification $\tau(\phi \wedge B\langle 1\rangle = B'\langle 2\rangle \Rightarrow \phi')$? Assume the latter holds in p, s_1, s_2. In particular $B\langle 1\rangle = B'\langle 2\rangle$ and we are thus in the two first branches of w. Moreover `tau(leftSpec) p s1 s2` and `tau(rightSpec) p s1 s2` simplify thanks to the branching and become trivial to show.

Since the specification w is written as a bind, we are in position to apply the $\mathcal{H}_{\text{rel}}$ bind rule. This gives us the two following premisses to discharge:

$$\frac{\begin{array}{c}\vdash \tau(B) \sim \tau(B')\,\{\lambda p\, s_1\, s2.\ p\, s_1(B)\, s_1\, s_2(B')\, s_2\} \\ \forall b_0, b_0' \vdash \texttt{if}\, b_0\, \texttt{then}\, \tau(C)\, \texttt{else}\, \tau(D) \sim \texttt{if}\, b_0'\, \texttt{then}\, \tau(C')\, \texttt{else}\, \tau(D')\,\{\texttt{if}\, b_0\, \texttt{then} \ldots \texttt{else} \ldots\}\end{array}}{\vdash \tau(c_1) \sim \tau(c_2)\,\{w\}}$$

In fact the first premisse is an instance of the completeness rule because $\theta_{\text{rel}}(\tau(B), \tau(B')) \leq \lambda p\, s_1\, s2.\ p\, s_1(B)\, s_1\, s_2(B')\, s_2$. For the second one, the idea is to destruct the two booleans present in the context. This yields four sequents to discharge. Two are discharged by the hypotheses (the translated premisses of the RHL if-rule). The two remaining sequents are discharged by the completeness-rule if we chose to set $w_{tf} = \theta_{\text{rel}}(\tau(C), \tau(D'))$ and $w_{ft} = \theta_{\text{rel}}(\tau(C'), \tau(D))$ in the definition of w above.

So the translated if-rule is indeed admissible in $\mathcal{H}_{\text{rel}}$.

The sequence rule.

$$\frac{\vdash C \sim C' : \phi \Rightarrow \phi' \qquad \vdash D \sim D' : \phi' \Rightarrow \phi''}{\vdash C; D \sim C'; D' : \phi \Rightarrow \phi''}$$

This rule translates into the following $\mathcal{H}_{\text{rel}}$ goal:

$$\frac{\vdash \tau(C) \sim \tau(C')\,\{\tau(\phi \Rightarrow \phi')\} \qquad \vdash \tau(D) \sim \tau(D')\,\{\tau(\phi' \Rightarrow \phi'')\}}{\vdash \tau(C); \tau(D) \sim \tau(C'); \tau(D')\,\{\tau(\phi \Rightarrow \phi'')\}}$$

In order provide a derivation for this goal, we proceed in a similar way. We begin by weakening the specification to match the computations. The translated conclusion of the rule exibhits two monadic sequences (bind-operators discarding the first resulting value). Thus we want to find a specification $w = w_c; w_d$ such that $w \leq \tau(\phi \Rightarrow \phi'')$. Of course we chose w to be $\tau(\phi \Rightarrow \phi'); \tau(\phi' \Rightarrow \phi'')$ in $W^{\text{St}}_{\text{rel}}$, and it satisfies the above inequality. We apply the weakening rule accordingly, and then the bind rule of $\mathcal{H}_{\text{rel}}$ to get

$$\frac{\vdash \tau(C) \sim \tau(C')\,\{\tau(\phi \Rightarrow \phi')\} \qquad \vdash \tau(D) \sim \tau(D')\,\{\tau(\phi' \Rightarrow \phi'')\}}{\vdash \tau(C); \tau(D) \sim \tau(C'); \tau(D')\,\{w\}}$$

The two premisses are our hypotheses, implying that the translated sequence rule is admissible in $\mathcal{H}_{\text{rel}}$.

Other rules. Following the same proof technique, the remaining translated RHL rules are also admissible within $\mathcal{H}_{\text{rel}}$. Note that the symmetry and transitivity rules of RHL have a peculiar status because their formulation heavily depends on the syntactic choices made in RHL. We did not consider these rules at all.

To sum up, a significant fragment of RHL can be interpreted in our monadic semantic framework, via the sound logic $\mathcal{H}_{\text{rel}}$.

5 An equivalent functorial semantics

This section is dedicated to the study of an alternative point of view for predicate semantics initially studied by Jacobs in [Jac12], [Jac13], [Jac14] and further studied in [Has15] by Hasuo. The perspective adopted on those works is that of categorical logic and its *functorial semantics*. Categorical logic stems from the observation that semantics, i.e. intepretations of syntactic constructs, can often be expressed by functors between proper categories (see discussion in 1). A predicate semantics for effectful programs modelized by a monad T is, under this perspective, simply a functor $\mathcal{K}\ell(T) \to \text{Poset}^{op}$, mapping Kleisli arrows to backward predicate transformers.

We study here the semantic framework defined in [Has15] ("Hasuo's framework") and analyse the relation between this framework and the dm4all framework trough the notion of *Kleisli liftings*. Hasuo's semantics can in fact be translated into a small fragement of the dm4all framework, specifically effect observations $M \to W^{\text{Pure}}$, wich demonstrates the power of our framework. Consequently we also freely obtain a categorical logic-flavored formulation of the dm4all framework for this kind of effect observations.

These lines of work were in fact initially started in order to study what kind of minimal interface should specification monads have. For some verification tasks such as ressource analyses, the specification needs to be able to express properties over quantities. It happens that the work done in [Jac12], [Jac13], [Jac14] partially tackles this problem.

5.1 Hasuo's framework

We focus our attention to functorial predicate semantics $\mathcal{K}\ell(T) \to \text{Poset}^{op}$ arrising from so-called "PT situations". In what follows it is good to view $T\,1$ as a collection of truth values. In fact the dm4all framework replaces this type by the propositional universe $\mathbb{P}$ and this is one of the reasons why it is more general than Hasuo's framework.

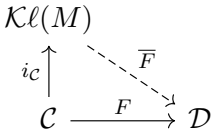
PT situations An order-enriched monad is a monad whose Kleisli category $\mathcal{K}\ell(T)$ is enriched over posets. In other words each hom set $\mathcal{K}\ell_T(X, Y)$ has a preorder structure and the Kleisli composition is monotonic on both arguments. Let T be an order enriched monad over a "type-like" category $\mathcal{C}$. A predicate transformer situation for T (PT situation) is an Eilenberg-Moore algebra $\tau : T^2\,1 \to T\,1$ over $T\,1$ satisfiyng some monotonicity conditions (see [Has15]).

Given such a PT situation it is easy to devise a functorial predicate semantics. We define the functor $\mathcal{K}\ell^{\tau}_T(-, 1) : \mathcal{K}\ell(T) \to \text{Poset}^{op}$ as follows:

- objects X of $\mathcal{C}$ are mapped to "predicates" $\mathcal{C}(X, T\,1)$.
- arrows $g : X \to T\,Y$ are mapped to backward predicate transformers $\mathcal{K}\ell^{\tau}_T(g, 1) : \mathcal{C}(Y, T\,1) \to \mathcal{C}(X, T\,1)$ defined as $\mathcal{K}\ell^{\tau}_T(g, 1)(q) = X \xrightarrow{g} T\,Y \xrightarrow{T(q)} T^2\,1 \xrightarrow{\tau} T\,1$.

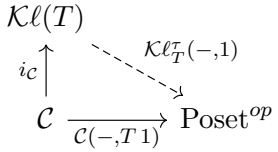
5.2 Kleisli liftings

Let $F : \mathcal{C} \to \mathcal{D}$ a functor. Asume having a monad M over the domain category $\mathcal{C}$. A Kleisli lifting of F is a functor $\overline{F} : \mathcal{K}\ell(M) \to \mathcal{D}$ making the following diagram commute:



In other words $\overline{F}$ has the same behaviour than F over effectless computations. The map $i_{\mathcal{C}}$ is the left adjoint of the Kleisli adjunction and just maps arrows $f : X \to Y$ into $\eta_Y \circ f : X \to M\,Y$. Next we observe that the two considered predicate semantics frameworks are equivalent to the datum of a Kleisli lifting of their predicate functor.

Hasuo's lifting. In this case the predicate functor is the contravariant functor $\mathcal{C}(-, T\,1) : \mathcal{C} \to \mathrm{Poset}^{op}$. The functorial predicate semantics we devised upon the PT situation τ is the functor $\mathcal{K}\ell^{\tau}_{T}(-, 1) : \mathcal{K}\ell(T) \to \mathrm{Poset}^{op}$. This functor is a lifting of the predicate functor $\mathcal{C}(-, T\,1)$.



Indeed take $f : X \to Y$ an arrow in $\mathcal{C}$. On one hand the predicate functor maps it to $- \circ f : \mathcal{C}(Y, T\,1) \to \mathcal{C}(X, T\,1)$. On the other hand $i_{\mathcal{C}}$ maps f to $\eta_Y \circ f$ wich is mapped to $\tau \circ T(-) \circ \eta_Y \circ f$. The latter simplifies into $\tau \circ \eta_{T\,1} \circ - \circ f$ thanks to the monadic equations. Since τ is an Eilenberg-Moore algebra the expression becomes $- \circ f$, making the diagram commute. So PT situations give rise to functorial predicate semantics wich are *liftings* of the predicate functor $\mathcal{C}(-, T\,1)$.

In the other direction it is possible to recover a PT situation starting with a lifting $\overline{\mathcal{C}(-, T\,1)}$ of the predicate functor. The exact formula is

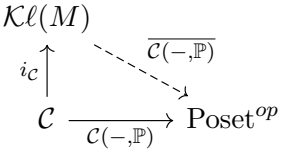
$$\tau = \overline{\mathcal{C}(T^2\,1 \xrightarrow{id} T(T\,1), T\,1)}(\mathrm{id}_{T\,1}) : T^2\,1 \to T\,1$$

The dm4all lifting. Let $\theta : M \to W^{\mathrm{Pure}}$ be an effect observation. Recall that $\mathcal{K}\ell$ can be viewed as a functor mapping monads to their Kleisli adjunction $\mathcal{K}\ell : \mathrm{Mon}_{\mathcal{C}} \to \mathrm{Adj}_{\mathcal{C}}$. Up to a forgetful functor we can type $\mathcal{K}\ell : \mathrm{Mon}_{\mathcal{C}} \to \mathrm{Cat}$. Now $\mathcal{K}\ell_{\theta}$ is a functor $\mathcal{K}\ell(M) \to \mathcal{K}\ell(W^{\mathrm{Pure}})$ and simply maps Kleisli arrows $g : X \to M\,Y$ to Kleisli arrows $\theta_Y \circ g$.

This time the predicate functor we wish to lift is $\mathcal{C}(-, \mathbb{P})$. In order to do so we can define $\overline{\mathcal{C}(-, \mathbb{P})} : \mathcal{K}\ell(M) \to \mathrm{Poset}^{op}$ as follows:

$$\overline{\mathcal{C}(X \xrightarrow{g} M\,Y, \mathbb{P})} = \lambda(p : Y \to \mathbb{P})(x : X).\ \mathcal{K}\ell_{\theta}(g)\,x\,p$$

As expected the following diagram commutes:



Indeed if $g = \eta_Y \circ f$ then we have $\mathcal{K}\ell_{\theta}(g)\,x\,p = \theta(\eta_Y(f\,x))\,p = \eta^{W}_{Y}(f\,x)\,p = p(f\,x)$ and it precisely the result of the bottom arrow $\mathcal{C}(f, \mathbb{P})\,p\,x$.

In the other direction assume having $\overline{\mathcal{C}(-, \mathbb{P})}$ a lifting of $\mathcal{C}(-, \mathbb{P})$. We need to recover an effect observation θ out of this Kleisli lifting. We define $\theta(my)$ by identifying $my : M\,Y$ with its selection arrow $1 \xrightarrow{my} M\,Y$ and computing

$$\theta_Y(my) = \overline{\mathcal{C}(1 \xrightarrow{my} M\,Y, \mathbb{P})} : \mathcal{C}(Y, \mathbb{P}) \to \mathcal{C}(1, \mathbb{P}) \simeq W^{\mathrm{Pure}}\,Y$$

Interestingly the fact that θ is a monad morphism relies on the two defining features of Kleilsi liftings. On one hand θ preserves the return operator because $\overline{\mathcal{C}(-,\mathbb{P})}$ is a lifting (the diagram commutes). On the other hand θ is compatible with the bind operator precisely because of the functoriality of $\overline{\mathcal{C}(-,\mathbb{P})}$.

In conclusion PT situations are in bijection with Kleisli liftings of the predicate functor $\mathcal{C}(-,T\,1)$ and effect observations $\theta : M \to W^{\mathrm{Pure}}$ are in bijection with Kleisli liftings of the predicate functor $\mathcal{C}(-,\mathbb{P})$.

5.3 Comparison

Now that we have a similar formulation for both frameworks it is possible to compare them. The rest of the section is a sketch of two different ideas to compare the two frameworks.

Via fibrations. Observe that there is a fibration

$$\begin{array}{ccc} [\mathcal{K}\ell(T), D] & & \{\overline{F_1}\} \overset{\lambda_*}{\circlearrowright} \{\overline{F_2}\} \\ {\scriptstyle \mathrm{MkPred}}\downarrow & & \\ [\mathcal{C}, \mathcal{D}] & & F_1 \xrightarrow{\ \lambda\ } F_2 \end{array}$$

Given a Kleisli lifting G, the fibration builds the associated predicate functor by $\mathrm{MkPred}(G)(X \xrightarrow{f} Y) = G(X \xrightarrow{f} Y \xrightarrow{\eta_Y} M\,Y)$. Let F_1 and F_2 two points of the base category, and λ an arrow between them. The fiber above F_i is the category of Kleisli liftings of F_i. We want to define a functor λ_* between those two categories. Consider the following diagram:

$$\begin{array}{ccc} F_2\,X & \overset{\overline{F_2}\,g\,?}{\dashrightarrow} & F_2\,Y \\ {\scriptstyle \lambda_X}\uparrow & \nearrow & \uparrow{\scriptstyle \lambda_Y} \\ F_1\,X & \xrightarrow[\overline{F_1}\,g]{} & F_1\,Y \end{array}$$

In order for $\lambda_Y \circ \overline{F_1}\,g$ to factor through λ_X we want it to be coarse enough compare to λ_X. In other words we intuitively wish for its kernel to be bigger than the kernel of λ_X. But this is always the case if we take a monic λ. Back to our concrete case, let us consider $\iota : T\,1 \to \mathbb{P}$ an injective function. For instance in the case of non-determinism, $T\,1 \simeq \mathbb{B}$ and ι is just an embedding of complete lattices. Via the Yonedda embedding $Y : \mathcal{C} \to [\mathcal{C}^{op}, \mathrm{Set}]$ we obtain a natural transformation between the two representable predicate functors $F_1 = \mathcal{C}(-,T\,1)$ and $F_2 = \mathcal{C}(-,\mathbb{P})$:

$$\lambda = Y(\iota) : \mathcal{C}(-,T\,1) \to \mathcal{C}(-,\mathbb{P})$$

Since λ has all its components injective, we can map any lifting $\overline{\mathcal{C}(-,T\,1)}$ to a lifting $\overline{\mathcal{C}(-,\mathbb{P})}$ of the dm4all predicate functor $\mathcal{C}(-,\mathbb{P})$, thanks to the above kernel analysis. But such liftings are known to be in bijective correspondance with effect observations $\theta : M \to W^{\mathrm{Pure}}$.

In short, under the existence of an injective map $\iota : T\,1 \to \mathbb{P}$, it is possible to build an effect observation out of a functorial predicate semantics à la Hasuo. Moreover this correspondance is injective meaning that the dm4all framework is more expressive in some sense.

Via modules. Monads are just monoid objects in the endofunctor category. Likewise, *modules over a monad* are monoid-action objects in the endofunctor category. More precisely a right M-module is a natural transformation $M \circ F \to F$ satisfying some action-like constraints.

In fact Kleisli liftings of functors $F : \mathcal{C} \to \mathcal{D}$ along the unit of a monad M are themselve in bijective correspondance with right M-module structures over the functor F. The latter result

is a variation of a theorem present in [Mul93]. We think that the right M-module obtained from the Hasuo Kleisli lifting $\overline{\mathcal{C}(-, T\,1)}$ embeds into the the right M-module obtained from the dm4all Kleisli lifting $\overline{\mathcal{C}(-, \mathbb{P})}$ but this claim has not been fully investigated yet.

In conclusion we are able to show that Hasuo's functorial predicate semantic framework is (under the existence of $\iota : T\,1 \to \mathbb{P}$) always a particular instance of the dm4all framework. Moreover the obtained effect observation θ maps effectful computations two points of W^{Pure}, one amongst many possible specification monads present in the dm4all framework.

References

[ACU15] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. *Logical Methods in Computer Science*, 11(1), 2015.

[AGH+17] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 362–375. ACM, 2017.

[AHM+17] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 515–529. ACM, January 2017.

[AJ94] Samson Abramsky and Achim Jung. Domain theory. 1994.

[BAF08] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *J. Log. Algebr. Program.*, 75(1):3–51, 2008.

[BEG+19] Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. Verifying relational properties using trace logic. Draft, June 2019.

[Ben04] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN Notices*, volume 39, pages 14–25. ACM, 2004.

[BHM00] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In *International Summer School on Applied Semantics*, pages 42–122. Springer, 2000.

[BNN16] Anindya Banerjee, David A. Naumann, and Mohammad Nikouei. Relational logic with framing and hypotheses. In Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen, editors, *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India*, volume 65 of *LIPIcs*, pages 11:1–11:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[ÇBG+17] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 316–329, 2017.

[CCcCK16] Rohit Chadha, Vincent Cheval, Ştefan Ciobâcă, and Steve Kremer. Automated verification of equivalence properties of cryptographic protocols. *ACM Trans. Comput. Log.*, 17(4):23:1–23:32, 2016.

[cCLRR16] Ştefan Ciobâcă, Dorel Lucanu, Vlad Rusu, and Grigore Rosu. A language-independent proof system for full program equivalence. *Formal Asp. Comput.*, 28(3):469–497, 2016.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010.

[Dij78] Edsger W Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. In *Programming Methodology*, pages 166–175. Springer, 1978.

[DN13] Germán Andrés Delbianco and Aleksandar Nanevski. Hoare-style reasoning with (algebraic) continuations. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 363–376. ACM, 2013.

[Flo93] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[GS10] Benny Godlin and Ofer Strichman. Inference rules for proving the equivalence of recursive procedures. In Zohar Manna and Doron A. Peled, editors, *Time for Verification, Essays in Memory of Amir Pnueli*, volume 6200 of *Lecture Notes in Computer Science*, pages 167–184. Springer, 2010.

[Has15] Ichiro Hasuo. Generic weakest precondition semantics from monads enriched with order. *Theor. Comput. Sci.*, 604:2–29, 2015.

[HDNV12] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. The marriage of bisimulations and kripke logical relations. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 59–72. ACM, 2012.

[HNDV14] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. A logical step forward in parametric bisimulations. Technical Report MPI-SWS-2014-003, 2014.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Jac99] Bart Jacobs. *Categorical logic and type theory*, volume 141. Elsevier, 1999.

[Jac12] Bart Jacobs. New directions in categorical logic, for classical, probabilistic and quantum logic. *arXiv preprint arXiv:1205.3940*, 2012.

[Jac13] Bart Jacobs. Measurable spaces and their effect logic. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 83–92. IEEE, 2013.

[Jac14] Bart Jacobs. Dijkstra monads in monadic computation. In *International Workshop on Coalgebraic Methods in Computer Science*, pages 135–150. Springer, 2014.

[Kat14] Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 633–646. ACM, 2014.

[KTL09] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 327–337. ACM, 2009.

[Lei05] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[MAA+19] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. To appear at ICFP, 2019.

[MHRM19] Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. The next 700 relational program logics. Submission to the 2020 POPL Symposium, 2019.

[Mog90] Eugenio Moggi. *An abstract view of programming languages.* University of Edinburgh. Department of Computer Science. Laboratory for . . . , 1990.

[Mul93] Philip S Mulry. Lifting theorems for kleisli categories. In *International Conference on Mathematical Foundations of Programming Semantics*, pages 304–319. Springer, 1993.

[NBG13] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Dependent type theory for verification of information flow and access control policies. *ACM Transactions on Programming Languages and Systems*, 35(2):6, 2013.

[nla] nlab. Relation between type theory and category theory.

[NMB08] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6):865–911, 2008.

[NMS+08] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot: dependent types for imperative programs. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 229–240. ACM, 2008.

[PP02] Gordon D. Plotkin and John Power. Notions of computation determine monads. In *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, pages 342–356, 2002.

[PP03] Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[QGG19] Weihao Qu, Marco Gaboardi, and Deepak Garg. Relational cost analysis for functional-imperative programs. To appear at ICFP, 2019.

[RBG+18] Ivan Radicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. Monadic refinements for relational cost analysis. *PACMPL*, 2(POPL):36:1–36:32, 2018.

[SD16] Marcelo Sousa and Isil Dillig. Cartesian Hoare logic for verifying k-safety properties. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 57–69. ACM, 2016.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.

[Shu17] Michael Shulman. Categorical logic from a categorical point of view. *Available on the web*, 2017.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[SWS+13] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398, 2013.

[TSKB18] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST. *PACMPL*, 2(POPL):64:1–64:28, 2018.

[WDLC18] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *PACMPL*, 2(POPL):56:1–56:29, 2018.

[Yan07] Hongseok Yang. Relational separation logic. *Theor. Comput. Sci.*, 375(1-3):308–334, 2007.