The Next 700 Relational Program Logics

Kenji Maillard, Catalin Hritcu, Exequiel Rivas, Antoine van Muylder

The Next 700 Relational Program Logics

KENJI MAILLARD, Inria Paris and ENS Paris, France
CĂTĂLIN HRIȚCU, Inria Paris, France
EXEQUIEL RIVAS, Inria Paris, France
ANTOINE VAN MUYLDER, Inria Paris and Université de Paris, France

We propose the first framework for defining relational program logics for arbitrary monadic effects. The framework is embedded within a relational dependent type theory and is highly expressive. At the semantic level, we provide an algebraic presentation of relational specifications as a class of relative monads, and link computations and specifications by introducing relational effect observations, which map pairs of monadic computations to relational specifications in a way that respects the algebraic structure. For an arbitrary relational effect observation, we generically define the core of a sound relational program logic, and explain how to complete it to a full-fledged logic for the monadic effect at hand. We show that this generic framework can be used to define relational program logics for effects as diverse as state, input-output, nondeterminism, and discrete probabilities. We, moreover, show that by instantiating our framework with state and unbounded iteration we can embed a variant of Benton's Relational Hoare Logic, and also sketch how to reconstruct Relational Hoare Type Theory. Finally, we identify and overcome conceptual challenges that prevented previous relational program logics from properly dealing with control effects, and are the first to provide a relational program logic for exceptions.

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

Additional Key Words and Phrases: program verification, relational program logics, side-effects, monads, state, I/O, nondeterminism, probabilities, exceptions, dependent types, semantics, relative monads, foundations

ACM Reference Format:
Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. 2020. The Next 700 Relational Program Logics. *Proc. ACM Program. Lang.* 4, POPL, Article 4 (January 2020), 33 pages. https://doi.org/10.1145/3371072

1 INTRODUCTION

Generalizing unary properties, which describe single program runs, *relational properties* describe relations between multiple runs of one or more programs [Abate et al. 2019; Clarkson and Schneider 2010]. Formally verifying relational properties has a broad range of practical applications. For instance, one might be interested in proving that the observable behaviors of two programs are related, showing for instance that the programs are *equivalent* [Blanchet et al. 2008; Chadha et al. 2016; Ştefan Ciobâcă et al. 2016; Godlin and Strichman 2010; Hur et al. 2012, 2014; Kundu et al. 2009; Timany et al. 2018; Wang et al. 2018; Yang 2007], or that one *refines* the other [Timany and Birkedal 2019]. In other cases, one might be interested in relating two runs of a single program, but, as soon as the control flow can differ between the two runs, the compositional verification problem becomes the same as relating two different programs. This is for instance the case for *noninterference*, which

Authors' addresses: Kenji Maillard, Inria Paris, Paris, ENS Paris, Paris, France; Cătălin Hriţcu, Inria Paris, Paris, France; Exequiel Rivas, Inria Paris, Paris, France; Antoine Van Muylder, Inria Paris, Paris, Université de Paris, Paris, France.

2475-1421/2020/1-ART4
https://doi.org/10.1145/3371072

requires that a program's public outputs are independent of its private inputs [Antonopoulos et al. 2017; Banerjee et al. 2016; Barthe et al. 2019; Clarkson and Schneider 2010; Nanevski et al. 2013; Sabelfeld and Myers 2003; Sousa and Dillig 2016]. The list of practical applications of relational verification is, however, much longer, including verified program transformations [Benton 2004], cost analysis [Çiçek et al. 2017; Qu et al. 2019; Radicek et al. 2018], program approximation [Carbin et al. 2012; He et al. 2018], semantic diffing [Girka et al. 2015, 2017; Lahiri et al. 2012; Wang et al. 2018], cryptographic proofs [Barthe et al. 2009, 2013a, 2014; Petcher and Morrisett 2015; Unruh 2019], differential privacy [Barthe et al. 2013b, 2015b; Gavazzo 2018; Zhang and Kifer 2017], and even machine learning [Sato et al. 2019].

As such, many different relational verification tools have been proposed, making different tradeoffs, for instance between automation and expressiveness (see §6 for further discussion). In this paper we focus on *relational program logics*, which are a popular formal foundation for various relational verification tools. Relational program logics are proof systems whose rules can be used to prove that a pair of programs meets a rich relational specification. As such they are very expressive, and can in particular handle situations in which verifying the desired relational properties requires showing the full functional correctness of certain pieces of code. Yet they can often greatly simplify reasoning by leveraging the syntactic similarities between the programs one relates. Since Benton's [2004] seminal Relational Hoare Logic, many relational program logics have been proposed [Aguirre et al. 2017; Banerjee et al. 2016; Barthe et al. 2013b, 2014, 2015b, 2016; Carbin et al. 2012; Nanevski et al. 2013; Petcher and Morrisett 2015; Qu et al. 2019; Radicek et al. 2018; Sato et al. 2019; Sousa and Dillig 2016; Unruh 2019; Yang 2007; Zhang and Kifer 2017]. However, each of these logics is specific to a particular combination of side-effects that is completely fixed by the programming language and verification framework; the most popular side-effects these logics bake in are mutable state, recursion, cost, and probabilities.

The goal of this paper is to distil the generic relational reasoning principles that work for a broad class of side-effects and that underlie most relational program logics. We do this by introducing the first framework for defining program logics for *arbitrary* monadic effects. Our generic framework is embedded within a dependent type theory, e.g., Coq, which makes it highly expressive and simpler to describe.

Syntactic rules. To factor out the fully generic parts, the rules of the relational program logics derived in our framework are divided into three categories, following the syntactic shape of the monadic programs on which they operate:

- R1 rules for pure language constructs, derived from the ambient dependent type theory (these rules target the elimination constructs for positive types, like if-then-else for booleans, recursors for inductive types, etc.);
- R2 rules for the generic monadic constructs return and bind; and
- R3 rules for effect-specific operations (e.g., get and put for the state monad, or throw and catch for the exception monad).

This organization allows us to clearly separate not only the generic parts (R1&R2) from the effect-specific ones (R3), but also the effect-irrelevant parts (R1) from the effect-relevant ones (R2&R3).

In its simplest form (§2), the judgment of the relational program logics of our framework has the shape: $\vdash c_1 \sim c_2 \ \{ w \}$, where $c_1 : \mathrm{M}_1\ A_1$ is a computation in monad M_1 producing results of type A_1, where $c_2 : M_2\ A_2$ is a computation in monad M_2 producing results of type A_2, and where w is a relational specification of computations c_1 and c_2 drawn from the type $\mathrm{W}_{\mathrm{rel}}(A_1, A_2)$. Here M_1 and M_2 are two arbitrary and *potentially distinct* computation monads (e.g., the state monad $\mathrm{St}\ A = S \longrightarrow A \times S$ and the exception monad $\mathrm{Exc}\ A = A + E$), while w could, for instance, be a pair of a relational precondition and a relational postcondition, or a relational predicate transformer—in

this introduction we will use relational weakest preconditions. For instance, for relating two state monads on states S_1 and S_2, we often use relational specifications drawn from

$$\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}(A_1, A_2) = ((A_1 \times S_1) \times (A_2 \times S_2) \to \mathbb{P}) \to S_1 \times S_2 \to \mathbb{P}$$

which are predicate transformers mapping postconditions relating two pairs of a result value and a final state to a precondition relating two initial states (here $\mathbb{P}$ stands for the type of propositions of our dependent type theory). As an example of the judgment above, consider the programs $c_1 = \mathtt{bind}^{\mathrm{St}}\ (\mathtt{get}\ ())\ (\lambda x\,.\,\mathtt{put}\ (x + k))$, which increments the content of a memory cell by k, and $c_2 = \mathtt{ret}^{\mathrm{St}}\ ()$, which does nothing. These two programs are related by the specification $w = \lambda\varphi\,(s_1, s_2).\ \varphi\,(((), s_1 + k), ((), s_2)) : \mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}(\mathbb{1}, \mathbb{1})$ saying that for the postcondition φ to hold for the final states of c_1 and c_2, it is enough for it to hold for $s_1 + k$ and s_2, where s_1 and s_2 are the computation's initial states. Note that since c_1, c_2, and w are terms of our ambient type theory, free variables (like k) are handled directly by the type theory, which saves the simple judgment from an explicit context.

For pure language constructs R1, we try to use the reasoning principles of our ambient dependent type theory as directly as possible. For instance, our framework (again in its simplest incarnation from §2) provides the following rule for the if-then-else construct:

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

In order to prove that c_1 and c_2 satisfy the relational specification $\mathtt{if}\ b\ \mathtt{then}\ w_{\top}\ \mathtt{else}\ w_{\bot}$, it is enough to prove that c_1 and c_2 satisfy both branches of the conditional in a context extended with the value of b. Interestingly, this rule does not make any assumption on the shape of c_1 and c_2. Relational program logics often classify each rule depending on whether it considers a syntactic construct that appears on both sides (synchronous), or only on one side (asynchronous). In the rule above, taking c_1 to be of the shape $\mathtt{if}\ b\ \mathtt{then}\ c_1^{\top}\ \mathtt{else}\ c_1^{\bot}$ and c_2 to be independent of b, we can simplify the premise according to the possible values of b to derive an asynchronous variant of the rule:

$$\frac{\vdash c_1^{\top} \sim c_2\ \left\{\ w^{\top}\ \right\} \qquad \vdash c_1^{\bot} \sim c_2\ \left\{\ w^{\bot}\ \right\}}{\vdash \mathtt{if}\ b\ \mathtt{then}\ c_1^{\top}\ \mathtt{else}\ c_1^{\bot} \sim c_2\ \left\{\ \mathtt{if}\ b\ \mathtt{then}\ w^{\top}\ \mathtt{else}\ w^{\bot}\ \right\}} \tag{1}$$

By requiring that both commands are conditionals, we can also derive the synchronous rule:

$$\frac{\vdash c_1^{\top} \sim c_2^{\top}\ \left\{\ w^{\top}\ \right\} \qquad \vdash c_1^{\bot} \sim c_2^{\bot}\ \left\{\ w^{\bot}\ \right\}}{\vdash \mathtt{if}\ b_1\ \mathtt{then}\ c_1^{\top}\ \mathtt{else}\ c_1^{\bot} \sim \mathtt{if}\ b_2\ \mathtt{then}\ c_2^{\top}\ \mathtt{else}\ c_2^{\bot}\ \{\ w^{\bullet}\ \}} \tag{2}$$

where the relational specification $w^{\bullet} = \lambda\varphi\, s_{12}.\ (b \Leftrightarrow b_1) \wedge (b \Leftrightarrow b_2) \wedge \mathtt{if}\ b\ \mathtt{then}\ w^{\top}\ \varphi\ s_{12}\ \mathtt{else}\ w^{\bot}\ \varphi\ s_{12}$ ensures that the booleans b_1 and b_2 controlling the choice of the branch in each computation share the same value b.

For the monadic constructs R2, the challenge is in lifting the binds and returns of the two computation monads M_1 and M_2 to the specification level. For instance, in a synchronous rule one would relate $\mathtt{bind}^{\mathrm{M}_1}\ m_1\ f_1$ to $\mathtt{bind}^{M_2}\ m_2\ f_2$ by first relating computations m_1 and m_2, say via relational specification w^m, and then one would relate the two functions f_1 and f_2 pointwise via a function w^f mapping arguments in $A_1 \times A_2$ to relational specifications:

$$\frac{\vdash m_1 \sim m_2\ \{\ w^m\ \} \qquad \forall a_1, a_2 \vdash f_1\ a_1 \sim f_2\ a_2\ \left\{\ w^f\ (a_1, a_2)\ \right\}}{\vdash \mathtt{bind}^{\mathrm{M}_1}\ m_1\ f_1 \sim \mathtt{bind}^{\mathrm{M}_2}\ m_2\ f_2\ \left\{\ \mathtt{bind}^{\mathrm{W}_{\mathrm{rel}}}\ w^m\ w^f\ \right\}} \tag{3}$$

In the conclusion of this rule, we need a way to compose $w : \mathrm{W}_{\mathrm{rel}}(A_1, A_2)$ and $w^f : A_1 \times A_2 \to \mathrm{W}_{\mathrm{rel}}(B_1, B_2)$ to obtain a relational specification for the two binds. We do this via a bind-like construct:

$$\mathtt{bind}^{\mathrm{W}_{\mathrm{rel}}} : \mathrm{W}_{\mathrm{rel}}(A_1, A_2) \to (A_1 \times A_2 \to \mathrm{W}_{\mathrm{rel}}(B_1, B_2)) \to \mathrm{W}_{\mathrm{rel}}(B_1, B_2) \tag{4}$$

For the concrete case of $\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}$, this bind-like construct takes the form

$$\mathtt{bind}^{\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}}\ w^m\ w^f = \lambda\varphi\ (s_1, s_2).\ w^m\ (\lambda\ ((a_1, s_1'), (a_2, s_2'))).\ w^f\ (a_1, a_2)\ (s_1', s_2')\ \varphi)\ (s_1, s_2).$$

This construct is written in continuation passing style: the specification of the continuation w^f maps a postcondition $\varphi : (B_1 \times S_1) \times (B_2 \times S_2) \to \mathbb{P}$ to an intermediate postcondition $(A_1 \times S_1) \times (A_2 \times S_2) \to \mathbb{P}$, then w^m turns it into a precondition for the whole computation.

Asynchronous rules for bind can be derived from the rule above, by taking m_1 to be $\mathtt{ret}^{\mathrm{M}_1}()$ or f_1 to be $\mathtt{ret}^{\mathrm{M}_1}$ above and using the monadic laws of M_1 (and symmetrically for M_2):

$$\frac{\vdash \mathtt{ret}^{\mathrm{M}_1}\ () \sim m_2\ \{\ w^m\ \} \qquad \forall a_2 \vdash c_1 \sim f_2\ a_2\ \left\{\ w^f\ a_2\ \right\}}{\vdash c_1 \sim \mathtt{bind}^{\mathrm{M}_2}\ m_2\ f_2\ \left\{\ \mathtt{bind}^{\mathrm{W}_{\mathrm{rel}}}\ w^m\ (\lambda((), a_2).\ w^f\ a_2)\ \right\}} \tag{5}$$

$$\frac{\vdash c_1 \sim m_2\ \{\ w^m\ \} \qquad \forall a_1, a_2 \vdash \mathtt{ret}^{\mathrm{M}_1}\ a_1 \sim f_2\ a_2\ \left\{\ w^f\ (a_1, a_2)\ \right\}}{\vdash c_1 \sim \mathtt{bind}^{\mathrm{M}_2}\ m_2\ f_2\ \left\{\ \mathtt{bind}^{\mathrm{W}_{\mathrm{rel}}}\ w^m\ w^f\ \right\}} \tag{6}$$

Finally, for the effect-specific operations R3, we provide a recipe for writing rules guided by our framework. For state, we introduce the following asynchronous rules for any a_1, a_2 and s:

$$\frac{}{\vdash \mathtt{get}\ () \sim \mathtt{ret}\ a_2\ \left\{\ w_{\mathtt{get}^l}\ \right\}} \qquad \frac{}{\vdash \mathtt{ret}\ a_1 \sim \mathtt{get}\ ()\ \left\{\ w_{\mathtt{get}^r}\ \right\}} \tag{7}$$

$$\frac{}{\vdash \mathtt{put}\ s \sim \mathtt{ret}\ a_2\ \left\{\ w_{\mathtt{put}^l}\ \right\}} \qquad \frac{}{\vdash \mathtt{ret}\ a_1 \sim \mathtt{put}\ s\ \left\{\ w_{\mathtt{put}^r}\ \right\}} \tag{8}$$

where $w_{\mathtt{get}^l} = \lambda\varphi\ (s_1, s_2).\ \varphi\ ((s_1, s_1), (a_2, s_2))$, $w_{\mathtt{get}^r} = \lambda\varphi\ (s_1, s_2).\ \varphi\ ((a_1, s_1), (s_2, s_2))$, $w_{\mathtt{put}^l} = \lambda\varphi\ (s_1, s_2).\ \varphi\ (((), s), (a_2, s_2))$ and $w_{\mathtt{put}^r} = \lambda\varphi\ (s_1, s_2).\ \varphi\ ((a_1, s_1), ((), s))$. Each of these rules describes at the specification level the action of a basic stateful operation (`get`, `put`) from either the left or the right computations, namely returning the current state for `get` or updating it for `put`. From these rules, we can derive two synchronous rules:

$$\frac{}{\vdash \mathtt{get}\ () \sim \mathtt{get}\ ()\ \left\{\ w_{\mathtt{get}}\ \right\}} \qquad \frac{}{\vdash \mathtt{put}\ s \sim \mathtt{put}\ s'\ \left\{\ w_{\mathtt{put}}\ \right\}}$$

where $w_{\mathtt{get}} = \lambda\varphi\ s_1\ s_2.\ \varphi\ ((s_1, s_1), (s_2, s_2))$ and $w_{\mathtt{put}} = \lambda\varphi\ s_1\ s_2.\ \varphi\ (((), s), ((), s'))$. These rules can be derived from the rule for $\mathtt{bind}^{\mathrm{W}_{\mathrm{rel}}}$, since by the monadic equations we can replace for instance $\vdash \mathtt{get}\ () \sim \mathtt{get}\ ()\ \left\{\ w_{\mathtt{get}}\ \right\}$ by the following derivation

$$\frac{\vdash \mathtt{ret}\ () \sim \mathtt{get}\ ()\ \left\{\ w_{\mathtt{get}^l}\ \right\} \qquad \forall u : \mathbb{1}, s_2 : S_2 \quad \vdash \mathtt{get}\ u \sim \mathtt{ret}\ s_2\ \left\{\ w_{\mathtt{get}^r}\ \right\}}{\vdash \mathtt{bind}^{\mathrm{St}s_1}\ (\mathtt{ret}\ ())\ \mathtt{get} \sim \mathtt{bind}^{\mathrm{St}s_2}\ (\mathtt{get}\ ())\ \mathtt{ret}^{\mathrm{St}s_2}\ \left\{\ \mathtt{bind}^{\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}}\ w_{\mathtt{get}^l}\ (\lambda(u, s_2).\ w_{\mathtt{get}^r})\ \right\}}$$

where the last specification reduces to $w_{\mathtt{get}}$ using the definition of $\mathtt{bind}^{\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}}$.

Simple semantics. To define a semantics for the $\vdash$ judgment above, we generalize recent work on (non-relational) effect observations [Maillard et al. 2019] to the relational setting, which raises significant challenges though. We start from two ideas from the non-relational setting: (1) specifications are drawn from a monad, ordered by precision [Ahman et al. 2017; Delbianco and Nanevski 2013; Maillard et al. 2019; Nanevski et al. 2008a,b, 2013; Swamy et al. 2013, 2016] and (2) one can link any computation with its specification by defining a monad morphism, i.e., a mapping between two monads that respects their monadic structure. In the case of state, an example monad morphism is $\theta^{\mathrm{St}}(c) = \lambda\varphi\ s.\ \varphi\ (c\ s) : \mathrm{St}\ A \to \mathrm{W}^{\mathrm{St}}\ A$, mapping a stateful computation to the unary specification monad $\mathrm{W}^{\mathrm{St}}\ A = (A \times S \to \mathbb{P}) \to S \to \mathbb{P}$, by running the computation and then checking whether the postcondition holds of the result. Inspired by Katsumata [2014], Maillard et al. [2019] call such monad morphisms *effect observations* and use them to decouple the computational monad from the specification monad, which brings great flexibility in choosing the specification monad and

verification style most suitable for the verification task at hand. Intuitively, an effect observation accounts for the various choices available when specifying computations with a particular effect, for instance total or partial correctness, angelic or demonic nondeterminism, ghost state, etc. In this paper we bring this flexibility to the relational verification world.

For this, we observe that even though $\mathrm{W}_{\mathrm{rel}}(A_1, A_2)$ is not a monad, it is a *relative monad* [Altenkirch et al. 2015] over the product $(A_1, A_2) \mapsto A_1 \times A_2$, as illustrated by the type of $\mathtt{bind}^{\mathrm{W}_{\mathrm{rel}}}$ above (4), where the continuation specification is passed a pair of results from the first specification. Similarly, we generalize monad morphisms to relative monads and observe that a relative monad morphism $\theta_{\mathrm{rel}} : \mathrm{M}_1\, A_1 \times \mathrm{M}_2\, A_2 \to \mathrm{W}_{\mathrm{rel}}(A_1, A_2)$ can immediately give us a semantics for the judgment above:

$$\vDash_{\theta_{\mathrm{rel}}} c_1 \sim c_2 \ \{\ w\ \} = \theta_{\mathrm{rel}}(c_1, c_2) \leq w,$$

by asking that the specification obtained by θ_{rel} is more precise than the user-provided specification w. In the case of state, $\theta^{\mathrm{St}}_{\mathrm{rel}}(c_1, c_2) = \lambda\varphi\ (s_1, s_2).\ \varphi\ (c_1\ s_1, c_2\ s_2)$ simply runs the two computations and passes the results to the postcondition. If we unfold this, and the definition of precision for $\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}$

$$w' \leq^{\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}} w = \forall\varphi\, s_1\, s_2.\ w\, \varphi\, (s_1, s_2) \Rightarrow w'\, \varphi\, (s_1, s_2), \tag{9}$$

we obtain the standard semantics of a relational program logic for stateful computations (but without other side-effects):

$$\vDash_{\theta^{\mathrm{St}}_{\mathrm{rel}}} c_1 \sim c_2 \ \{\ w\ \} = \forall\varphi\, s_1\, s_2.\ w\, \varphi\, (s_1, s_2) \Rightarrow \varphi\, (c_1\, s_1, c_2\, s_2)$$

Another important point is that the relational effect observation can help us in deriving simple effect-specific rules, such as the ones for `get` (7) and `put` (8) above. For deriving such rules, one first has to choose c_1 and c_2 (and we hope that the product programs of §5 can provide guidance on this in the future) and then one can simply compute the specification using θ. For instance, $w_{\mathtt{get}^I} = \lambda\varphi\ (s_1, s_2).\ \varphi\ ((s_1, s_1), (a_2, s_2))$ in the first get rule (7) really is just $\theta(\mathtt{get}\ (), \mathtt{ret}\ a_2)$. This idea is illustrated for various other effects in §2.6.

Finally, for probabilities and sometimes nondeterminism, one has to relax the definition of relational effect observations to account for the fact that, in the relational setting, modular verification can have a precision cost compared to whole-program verification. While for relative monad morphisms the following bind law has to hold with equality (and analogously for returns):

$$\theta_{\mathrm{rel}}\, (\mathtt{bind}^{\mathrm{M}_1}\, m_1\, f_1, \mathtt{bind}^{\mathrm{M}_2}\, m_2\, f_2) = \mathtt{bind}^{\mathrm{W}_{\mathrm{rel}}}\, (\theta_{\mathrm{rel}}\, (m_1, m_2))\, (\theta_{\mathrm{rel}} \circ (f_1, f_2))$$

we introduce a notion of *lax* relative monad morphism that allows the left-hand-side (i.e., less modular verification) to be more precise than the right-hand-side (i.e., more modular verification):

$$\theta_{\mathrm{rel}}\, (\mathtt{bind}^{\mathrm{M}_1}\, m_1\, f_1, \mathtt{bind}^{\mathrm{M}_2}\, m_2\, f_2) \leq \mathtt{bind}^{\mathrm{W}_{\mathrm{rel}}}\, (\theta_{\mathrm{rel}}\, (m_1, m_2))\, (\theta_{\mathrm{rel}} \circ (f_1, f_2)).$$

For instance, the *refinement* relational effect observation $\theta^{\forall\exists}$ interprets two nondeterministic computations $c_1 : \mathrm{Nd}\, A_1$ and $c_2 : \mathrm{Nd}\, A_2$ (represented as finite sets of possible outcomes) into the relational specification monad $\mathrm{W}^{\mathrm{Pure}}_{\mathrm{rel}}(A_1, A_2) = (A_1 \times A_2 \to \mathbb{P}) \to \mathbb{P}$ as follows:

$$\theta^{\forall\exists}_{\mathrm{rel}}\, (c_1, c_2) = \lambda\varphi.\ \forall a_1 \in c_1.\ \exists a_2 \in c_2.\ \varphi(a_1, a_2)$$

This interpretation is a natural generalization of subset inclusion (i.e., refinement of nondeterminism) to arbitrary relational postconditions φ, but only satisfies the lax monad morphism law relating $\mathtt{bind}^{\mathrm{Nd}}\ m\ f = \bigcup_{a \in m} f\ a$ and $\mathtt{bind}^{\mathrm{W}^{\mathrm{Pure}}_{\mathrm{rel}}}\ w\ f = \lambda\varphi.\ w\ (\lambda(a_1, a_2).\ f\ (a_1, a_2)\ \varphi)$:

$$\begin{array}{rcl} \theta^{\forall\exists}_{\mathrm{rel}}\, (\mathtt{bind}^{\mathrm{Nd}}\, m_1\, f_1, \mathtt{bind}^{\mathrm{Nd}}\, m_2\, f_2) & = & \lambda\varphi.\ \forall a_1 \in m_1.\ \forall b_1 \in f_1\ a_1.\ \exists a_2 \in m_2.\ \exists b_2 \in f_2\ a_2.\ \varphi(b_1, b_2) \\ & \leq & \\ \mathtt{bind}^{\mathrm{W}^{\mathrm{Pure}}_{\mathrm{rel}}}\, (\theta^{\forall\exists}_{\mathrm{rel}}\, (m_1, m_2))\, (\theta^{\forall\exists}_{\mathrm{rel}} \circ (f_1, f_2)) & = & \lambda\varphi.\ \forall a_1 \in m_1.\ \exists a_2 \in m_2.\ \forall b_1 \in f_1\ a_1.\ \exists b_2 \in f_2\ a_2.\ \varphi(b_1, b_2) \end{array}$$

On the left-hand-side one can choose a different a_2 for every b_1, while on the right-hand-side a single a_2 has to work for every b_1, so the two preconditions are not logically equivalent. Supporting such lax relational effect observations when needed is still relatively simple, even if deriving useful effect specific rules is generally more challenging in this case.

Exceptions, and why the simple semantics is not enough. The simple construction we described so far is a natural extension of the solution we previously proposed in the unary setting [Maillard et al. 2019]. It works well for defining relational program logics for state and nondeterminism (and also input-output and probabilities), but it hits a limit when we try to incorporate exceptions. Indeed, defining a relational program logic for exceptions was an open research problem, and our proposed solution depends on solving several non-trivial technical challenges. Here we begin with an analysis of the main obstruction of applying the simple construction above to exceptions, and how that guides us to a generic construction that can be made to work.

For relating computations that can raise exceptions, we often need to use expressive specifications that can tell whether an exception was raised or not in each of the computations. For instance, such relational specifications could be drawn from:

$$\mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}(A_1, A_2) = ((A_1 + E_1) \times (A_2 + E_2) \to \mathbb{P}) \to \mathbb{P}.$$

A predicate transformer $w : \mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}(A_1, A_2)$ maps an exception-aware postcondition $\varphi : (A_1 + E_1) \times (A_2 + E_2) \to \mathbb{P}$ to a precondition, which is just a proposition in $\mathbb{P}$. However, more work is needed to obtain a *compositional* proof system. Indeed, suppose we have derivations for $\vdash m_1 \sim m_2 \ \{ \ w^m \ \}$ and $\forall a_1, a_2, \vdash f_1\ a_1 \sim f_2\ a_2 \ \left\{ \ w^f\ (a_1, a_2) \ \right\}$ with specifications w^m and $w^f\ (a_1, a_2)$ drawn from $\mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}$. In order to build a composite proof relating $c_1 = \mathtt{bind}^{\mathrm{Exc}}\ m_1\ f_1$ and $c_2 = \mathtt{bind}^{\mathrm{Exc}}\ m_2\ f_2$ we need compose w^m and w^f in some way. If w^m ensures that m_1 and m_2 terminate both normally returning values we can compose with w^f and if they both throw exceptions we can pass the exceptions to the final postcondition. Otherwise, a computation, say m_1, returns a value and the other, m_2, raises an exception. In this situation, the specification relating c_1 and c_2 needs a specification for the continuation f_1 of m_1, but this cannot be extracted out of w^f alone. In terms of the constructs of $\mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}$, this failure is an obstruction to complete the following tentative definition of $\mathtt{bind}^{\mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}}$:

$$
\begin{array}{l}
\mathtt{let}\ \mathtt{bind}^{\mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}}\ w^m\ (w^f : A_1 \times A_2 \to (((B_1 + E_1) \times (B_2 + E_2)) \to \mathbb{P}) \to \mathbb{P})\ (\varphi : (B_1 + E_1) \times (B_2 + E_2) \to \mathbb{P}) = \\
\quad w^m\ (\lambda x : (A_1 + E_1) \times (A_2 + E_2). \\
\qquad \mathtt{match}\ x\ \mathtt{with} \\
\qquad \mid \mathrm{Inl}\ a_1, \mathrm{Inl}\ a_2 \to w^f\ a_1\ a_2\ \varphi \\
\qquad \mid \mathrm{Inr}\ e_1, \mathrm{Inr}\ e_2 \to \varphi\ (\mathrm{Inr}\ e_1, \mathrm{Inr}\ e_2) \\
\qquad \mid \_ \to ???\ )
\end{array}
$$

Our solution is to pass in two independent *unary* (i.e., non-relational) specifications for the continuations f_1 and f_2 as additional arguments for bind:

$$
\begin{array}{l}
\mathtt{let}\ \mathtt{bind}^{\mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}}\ w^m\ (w^{f_1} : A_1 \to ((B_1 + E_1) \to \mathbb{P}) \to \mathbb{P})\ (w^{f_2} : A_2 \to ((B_2 + E_2) \to \mathbb{P}) \to \mathbb{P})\ w^f\ \varphi = \\
\quad w^m\ (\lambda x : (A_1 + E_1) \times (A_2 + E_2). \\
\qquad \mathtt{match}\ x\ \mathtt{with} \\
\qquad ... \\
\qquad \mid \mathrm{Inl}\ a_1, \mathrm{Inr}\ e_2 \to w^{f_1}\ a_1\ (\lambda be.\ \varphi\ be\ (\mathrm{Inr}\ e_2)) \\
\qquad \mid \mathrm{Inr}\ e_1, \mathrm{Inl}\ a_2 \to w^{f_2}\ a_2\ (\lambda be.\ \varphi\ (\mathrm{Inr}\ e_1)\ be)\ )
\end{array}
$$

The first new case corresponds to when m_2 terminated with an exception whereas m_1 returned a value normally. In this situation, we use the unary specification w^{f_2} to further evaluate the first computation, independently of the second one, which already terminated. The key observation is that the operation $\mathtt{bind}^{\mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}}$ can still be used to define a relative monad, but in a more complex relational setting that we introduce in §3. As a consequence of moving to this more complex setting our relational judgment needs to also keep track of unary specifications, and its semantics also becomes more complex. We tame this complexity by working this out internally to a *relational* dependent type theory [Tonelli 2013]. These two novel conceptual ideas (combining unary and

binary specifications, and embedding inside a relational dependent type theory for dealing with contexts) are fundamental pieces that make the generic framework work. In practice we can still implement this relational dependent type theory inside our ambient type theory and continue using the same tools for verification. We use Coq for developing a proof of concept implementation: we represent the types in the relational type theory using relations, and implement a set of combinators that account for type formers such as Π-types.

This paper makes the following **contributions**:

- We introduce the first generic framework for deriving relational program logics for arbitrary monadic effects, distilling the essence of previous relational program logics for specific effects. The proposed framework is highly expressive, and not only allows one to prove arbitrary relations between two programs with different monadic effects, but it also inherits the features of dependent type theory (higher-order logic, dependent types, polymorphism, lambdas, etc).
- We provide a generic semantics for these relational program logics based on the novel observations that (1) the algebraic structure of relational specifications can be captured by particular relative monads, and (2) the two considered computations can be mapped to their specifications by particular relative monad morphisms we call relational effect observations. Our framework provides great flexibility in choosing the kind of relational specifications and the effect observation best suited for the verification task at hand. Finally, our generic rules are proven sound for any specification monad and any effect observation.
- We show that this generic framework can be used to define relational program logics for effects as diverse as state, input-output, nondeterminism, and discrete probabilities. Moreover, we show that by instantiating our framework with state and unbounded iteration, we obtain a logic expressive enough to encode a variant of Benton's [2004] Relational Hoare Logic (RHL) (§4.1). Finally, we also sketch how Nanevski et al.'s [2013] Relational Hoare Type Theory (RHTT) can be reconstructed on top of our framework (§4.2).
- We identify and overcome conceptual challenges that prevented previous relational program logics from dealing with control effects such as exceptions [Barthe et al. 2016]. We provide a proper semantic account of exceptions and the first relational program logic for this effect.
- We propose a monadic notion of product programs, and illustrate it for the state effect.

Outline. After recalling how computational monads can express a wide range of effects, §2 introduces relational specification monads and effect observations, on top of which we build a simplified variant of our relational reasoning framework, which we illustrate for state, input-output, nondeterminism, discrete probabilities, and unbounded iteration, and also with proofs of noninterference. §3 then extends this simplified setting to account for effects including exceptions, based on a relational dependent type theory and also using relative monads as a unifying tool for the two settings. §4 explains how to embed RHL and the connection to RHTT. In §5 we discuss product programs, before reviewing related work in §6 and concluding in §7. The ideas of this paper are supported by an accompanying Coq development providing a proof of concept implementation (available at https://gitlab.inria.fr/kmaillar/dijkstra-monads-for-all/tree/relational) that includes both the simplified and generic frameworks.

2 SIMPLIFIED FRAMEWORK

In this section we introduce a simple framework for relational reasoning about monadic programs based on (1) relational specification monads, capturing relations between monadic programs, and (2) relational effect observations, lifting a pair of computations to their specification. By instantiating this framework with specific effects, we show how the specific rules of previous relational program logics can be recovered in a principled way and illustrate by example how these rules can be used

to prove relational properties of monadic programs, such as noninterference. But first, we recall the monadic presentation of a few effects such as state, exceptions, and nondeterminism.

2.1 From Effects to Monads

The seminal work of Moggi [1989] proposes using computational monads to encapsulate effects. A monad is a parametrized type $\mathrm{M}\,A$ equipped with two operations $\mathtt{ret}^{\mathrm{M}} : A \to \mathrm{M}\,A$, sending a value to an effectful context, and $\mathtt{bind}^{\mathrm{M}} : \mathrm{M}\,A \to (A \to \mathrm{M}\,B) \to \mathrm{M}\,B$, sequentially composes an effectful computation returning values in A with a continuation returning values in B, resulting in a B-valued computation. Crucially, these operations obey 3 laws – unitality of $\mathtt{ret}$ with respect to $\mathtt{bind}$ and associativity of $\mathtt{bind}$ – ensuring that any combinations of $\mathtt{ret}^{\mathrm{M}}$ and $\mathtt{bind}^{\mathrm{M}}$ can be seen as a linear sequence of computations.

$$\mathtt{bind}^{\mathrm{M}}\,(\mathtt{ret}^{\mathrm{M}}\,a)\,f = f\,a \qquad\qquad \mathtt{bind}^{\mathrm{M}}\,m\,\mathtt{ret}^{\mathrm{M}} = m$$

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

A considerable number of effects are captured by monads, including stateful computations, exceptions, interactive input-output, nontermination, nondeterminism, and continuations [Benton et al. 2000]. Each monad comes with specific *operations* [Plotkin and Power 2002] that allow the computation to perform the actual effects that the monad provides. To fix notation, we recall the basic monads corresponding to the effects that we will use in the rest of the paper.

Stateful computations. State passing functions $\mathrm{St}\,A = S \to A \times S$ are used to model state, where S is the type of the state. The functions $\mathtt{ret}^{\mathrm{St}}$ and $\mathtt{bind}^{\mathrm{St}}$ are defined as

$$\mathtt{let}\ \mathtt{ret}^{\mathrm{St}}\ a : \mathrm{St}\ A = \lambda s.\ (a,s) \qquad \mathtt{let}\ \mathtt{bind}^{\mathrm{St}}\ (m{:}\mathrm{St}\ A)\ (f{:}A \to \mathrm{St}\ B) : \mathrm{St}\ B = \lambda s.\ \mathtt{let}\ (a,s') = m\ s\ \mathtt{in}\ f\ a\ s'$$

This monad comes with two operations

$$\mathtt{let}\ \mathtt{get} : \mathrm{St}\ S = \lambda s.\ (s,s) \qquad\qquad \mathtt{let}\ \mathtt{put}\ (s{:}S) : \mathrm{St}\ \mathbb{1} = \lambda s_0.\ ((),\ s)$$

that permit reading and updating the state. A particular case of state are stores with many locations of a particular type $\mathcal{V}\mathrm{al}$. If $\mathcal{L}$ is a set of locations, then a computations with a store of type $S = \mathcal{L} \to \mathcal{V}\mathrm{al}$ are expressed by monad St_S. In this case, we have custom operations that are parameterized by the location which we are accessing in the store:

$$\mathtt{let}\ \mathtt{get}^{\mathcal{L}}(l{:}\mathcal{L}) : \mathrm{St}\ \mathcal{V}\mathrm{al} = \lambda s.\ (s\ l,s) \qquad\qquad \mathtt{let}\ \mathtt{put}^{\mathcal{L}}(l : \mathcal{L})\ (v : \mathcal{V}\mathrm{al}) : \mathrm{St}\ \mathbb{1} = \lambda s.\ ((),\ \mathit{upd}\ s\ l\ v)$$

where $\mathtt{let}\ \mathit{upd}\ s\ l_1\ v = \lambda l_2.\ \mathtt{if}\ l_2 = l_1\ \mathtt{then}\ v\ \mathtt{else}\ s\ l_2$.

Exceptions. Computations potentially throwing exceptions of type E are captured by the type constructor $\mathrm{Exc}\,A = A + E$. The monadic operations are

$$\mathtt{let}\ \mathtt{ret}^{\mathrm{Exc}}\ a : \mathrm{Exc}\ A = \mathrm{Inl}\ a$$
$$\mathtt{let}\ \mathtt{bind}^{\mathrm{Exc}}\ (m{:}\mathrm{Exc}\ A)\ (f{:}A \to \mathrm{Exc}\ B) : \mathrm{Exc}\ B = \mathtt{match}\ m\ \mathtt{with}\ |\ \mathrm{Inl}\ a \to f\ a\ |\ \mathrm{Inr}\ e \to \mathrm{Inr}\ e$$

The operations provided are throwing and catching exceptions[1]:

$$\mathtt{let}\ \mathtt{throw}\,(e{:}E) : \mathrm{Exc}\ \mathbb{0} = \mathrm{Inr}\ e$$
$$\mathtt{let}\ \mathtt{catch}\,(m{:}\mathrm{Exc}\ A)\ (\mathit{mexc} : E \to \mathrm{Exc}\ A) : \mathrm{Exc}\ A = \mathtt{match}\ m\ \mathtt{with}\ |\ \mathrm{Inl}\ a \to \mathrm{Inl}\ a\ |\ \mathrm{Inr}\ e \to \mathit{mexc}\ e$$

Interactive Input-output. Computations doing interactive input of type I and output of type O are captured using monads as well. The type constructor has a tree-like form

$$\mathtt{type}\ \mathrm{IO}\ A = |\ \mathrm{Ret} : A \to \mathrm{IO}\ A\ |\ \mathrm{Input} : (I \to \mathrm{IO}\ A) \to \mathrm{IO}\ A\ |\ \mathrm{Output} : O \to \mathrm{IO}\ A \to \mathrm{IO}\ A$$

[1] Catching exceptions is the primary example of a *handler* [Plotkin and Pretnar 2009]; we use here the term operation in a wide sense englobing both *algebraic operations* (that we present as *generic effects* [Plotkin and Power 2002]) and handlers.

which consists of three possible cases: either we are done with a return value (Ret), or we expect a new input and then continue (Input), or we output and the continue (Output). The monadic function $\mathtt{ret}^{\mathrm{IO}}$ constructs a unique leaf tree using Ret and $\mathtt{bind}^{\mathrm{IO}}$ does tree grafting. The operations perform input and output, and they are directly captured using the corresponding constructors.

$$\mathtt{let\ input} : \mathrm{IO}\ I = \mathrm{Input}\ (\lambda i\,.\,\mathtt{ret}^{\mathrm{IO}}\ i) \qquad \mathtt{let\ output}\ (o : O) : \mathrm{IO}\ \mathbb{1} = \mathrm{Output}\ o\ (\mathtt{ret}^{\mathrm{IO}}\ ())$$

We call this monad the input-output monad on (I, O).

Nondeterminism. The finite powerset $\mathrm{Nd}\,X = \mathcal{P}_{fin}(X)$ models nondeterministic computations as a set of possible outcomes. The return operation maps a value v to the singleton $\{v\}$, and the bind operation uses union to collect all results, i.e., $\mathtt{bind}^{\mathrm{Nd}}\ m\ f = \bigcup_{v \in m} f\ v$. The operation $\mathtt{pick} = \{\mathtt{tt}, \mathtt{ff}\} : \mathrm{Nd}\,\mathbb{B}$ nondeterministically select a boolean value, whereas the operation $\mathtt{fail} : \mathrm{Nd}\,\mathbb{0}$ does not return any value. Repeating this operation, we can nondeterministically $\mathtt{choose} : (n : \mathbb{N}) \to \mathtt{fin}\,n$ an element of a finite set $\mathtt{fin}\,n$.

Imp-like effect. To capture the syntax of simple imperative programs, manipulating state and unbounded iteration, we introduce the Imp monad:

$$\begin{array}{ll} \mathtt{type}\ \mathrm{Imp}\ A = \mid \mathrm{Ret} : A \to \mathrm{Imp}\ A & \mid \mathrm{DoWhile} : \mathrm{Imp}\ \mathbb{B} \to \mathrm{Imp}\ A \to \mathrm{Imp}\ A \\ \qquad\qquad \mid \mathrm{Get} : (S \to \mathrm{Imp}\ A) \to \mathrm{Imp}\ A & \mid \mathrm{Put} : S \to \mathrm{Imp}\ A \to \mathrm{Imp}\ A \end{array}$$

Besides the monadic operations and the stateful ones, the Imp monad is built to offer an operation

$$\mathtt{let\ do\_while}\ (\mathit{body} : \mathrm{Imp}\ \mathbb{B}) : \mathrm{Imp}\ \mathbb{1} = \mathrm{DoWhile}\ \mathit{body}\ (\mathrm{Ret}\ ())$$

The expected semantics of this operation is to take a computation *body* and to iterate *body* as long as it returns true, so that the following equation – which does not hold in Imp – is satisfied

$$\mathtt{do\_while}\ \mathit{body} \quad = \quad \mathtt{bind}^{\mathrm{Imp}}\ \mathit{body}\ (\lambda b\,.\,\mathtt{if}\ b\ \mathtt{then}\ \mathtt{do\_while}\ \mathit{body}\ \mathtt{else}\ \mathtt{ret}^{\mathrm{Imp}}\ ())$$

When defining functions out of Imp, we will thus make sure that it holds in the target.

Probabilities. A probabilistic computation is a sub-probability distribution on possible outcomes, i.e., for a countable type A, $\mathrm{Prob}\,A$ represents functions $f : A \to \mathbb{I}$ (where we write $\mathbb{I}$ for the unit interval $[0; 1]$) such that $\sum_{a \in A} f\ a \leq 1$. Restricting our attention to discrete probabilities, the monad structure on Prob is known as the *Giry monad* [Giry 1982]. The Dirac distribution at v assigning weight 1 to v and 0 to any other value implements returns. Binding a distribution $m : \mathrm{Prob}\,A$ to a function $f : A \to \mathrm{Prob}\,B$ amounts to computing the distribution on B given by $\lambda y.\ \Sigma_{x \in \mathrm{supp}(m)} f\ x\ y$. We can consider various basic distributions on countable spaces as operations, for instance $\mathtt{flip} : \mathbb{I} \to \mathrm{Prob}\,\mathbb{B}$ provides a Bernoulli distribution $\mathcal{B}_p$ on booleans (with parameter $p \in \mathbb{I}$).

2.2 Specifications as (Relative) Monads

An important idea in the non-relational verification setting is to encapsulate the specification of a monadic computation inside a monad [Ahman et al. 2017; Delbianco and Nanevski 2013; Maillard et al. 2019; Nanevski et al. 2008a,b, 2013; Swamy et al. 2013, 2016], giving the same algebraic footing to both computations and specifications. For instance, stateful computations returning values in A are elements of a state monad $\mathrm{St}\,A = S \to (A \times S)$ and can be given specifications drawn from the monad $\mathrm{W}^{\mathrm{St}}\,A = (A \times S \to \mathbb{P}) \to (S \to \mathbb{P})$ equipped with the monad structure given by

$$\mathtt{let}\ \mathtt{ret}^{\mathrm{W^{St}}}\ (a{:}A) : \mathrm{W^{St}}\,A = \lambda\varphi\ s.\ \varphi\ (a,s)$$
$$\mathtt{let}\ \mathtt{bind}^{\mathrm{W^{St}}}\ (wm : \mathrm{W^{St}}\,A)\ (wf{:}\ A \to \mathrm{W^{St}}\,B) : \mathrm{W^{St}}\,B = \lambda\varphi\ s.\ wm\ (\lambda\ a.\ wf\ a\ \varphi)\ s$$

Intuitively, a specification $w : \mathrm{W^{St}}\,A$ is a predicate transformer mapping postconditions, which are predicates on the return value and final state, to preconditions, which are predicates on the initial

state. The monadic structure on W^{St} provides a canonical way to describe the monadic rules of a non-relational program logic, i.e.,

$$\frac{\vdash v : A}{\vdash \mathtt{ret}^{\mathrm{St}}\, v : \mathrm{St}\, A\, \{\, \mathtt{ret}^{\mathrm{W^{St}}}\, v \,\}} \qquad \frac{\vdash m : \mathrm{St}\, A\, \{\, w^m \,\} \qquad a : A \vdash f\, a : \mathrm{St}\, B\, \{\, w^f \,\}}{\vdash \mathtt{bind}^{\mathrm{St}}\, m\, f : \mathrm{St}\, B\, \{\, \mathtt{bind}^{\mathrm{W^{St}}}\, w^m\, w^f \,\}} \tag{10}$$

This is, in fact, the main idea behind Dijkstra monads [Ahman et al. 2017; Jacobs 2015; Maillard et al. 2019; Swamy et al. 2013, 2016], which additionally internalize $\mathrm{St}\, A\, \{w\}$ as a computation type.

Now returning to the relational setting, a relational specification for a pair of stateful computations $c_1 : \mathrm{St}_{S_1}\, A_1$ and $c_2 : \mathrm{St}_{S_2}\, A_2$ consists of a predicate transformer w mapping postconditions relating two pairs of a result value and a final state to a precondition relating two initial states, i.e.,

$$\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}(A_1, A_2) = ((A_1 \times S_1) \times (A_2 \times S_2) \to \mathbb{P}) \to S_1 \times S_2 \to \mathbb{P}. \tag{11}$$

$\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}$ does not posses the monad structure its unary variant has. To begin with it is not even an endofunctor: it takes *two* types as input and produces one. However, in order to derive a relational program logic, we need operations playing the role of $\mathtt{ret}^{\mathrm{W^{St}}}$ and $\mathtt{bind}^{\mathrm{W^{St}}}$ in the unary rules (10). In detail, we need a specification covering the case of two returns, as well as a combinator producing a specification for a pair of $\mathtt{bind}^{\mathrm{St}}$ out of specifications for the subcomputations. In the particular case of $\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}$, the monadic operations of the unary variant W^{St} can be naturally extended to the relational setting providing such combinators:

$$\mathtt{let}\ \mathtt{ret}^{\mathrm{W^{St}_{rel}}}\ (a_1{,}a_2){:}A_1 \times A_2 : \mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}(A_1{,}A_2) = \lambda\varphi\ (s_1,\, s_2).\ \varphi\ ((a_1{,}s_1),\, (a_2{,}s_2))$$

$$\mathtt{let}\ \mathtt{bind}^{\mathrm{W^{St}_{rel}}}\ (wm : \mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}(A_1{,}A_2))\ (wf{:}A_1 \times A_2 \to \mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}(B_1{,}B_2)) : \mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}(B_1{,}B_2) =$$
$$\lambda\varphi\ (s_1{,}s_2).\ wm\ (\lambda\ ((a_1{,}s_1'){,}(a_2{,}s_2')).\ wf\,(a_1,\, a_2)\ \varphi\ (s_1'{,}s_2'))$$

These operations satisfy equations analogous to the monadic ones and are part of a relative monad structure in the sense of Altenkirch et al. [2015]. The relational specifications for state $\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}$ are also naturally ordered by $\leq^{\mathrm{W^{St}_{rel}}}$ (see (9) in §1) and this ordering is compatible with the relative monad structure, as long as we restrict our attention to *monotonic* predicate transformers, a condition that we will assume from now on for all monads on predicate transformers. We call such a monad-like structure equipped with a compatible ordering a *simple relational specification monad*.

Definition 1. *A* simple relational specification monad *consist of*

- *for each pair of types* (A_1, A_2)*, a type* $\mathrm{W}_{rel}(A_1, A_2)$ *equipped with a preorder* $\leq^{\mathrm{W}_{rel}}$
- *an operation* $\mathtt{ret}^{\mathrm{W}_{rel}} : A_1 \times A_2 \to \mathrm{W}_{rel}(A_1, A_2)$
- *an operation* $\mathtt{bind}^{\mathrm{W}_{rel}} : \mathrm{W}_{rel}(A_1, A_2) \to (A_1 \times A_2 \to \mathrm{W}_{rel}(B_1, B_2)) \to \mathrm{W}_{rel}(B_1, B_2)$ *monotonic in both arguments*
- *satisfying the 3 following equations*

$$\mathtt{bind}^{\mathrm{W}_{rel}}\,(\mathtt{ret}^{\mathrm{W}_{rel}}\,(a_1, a_2))\ w^f = w^f\ (a_1, a_2) \qquad \mathtt{bind}^{\mathrm{W}_{rel}}\ w^m\ \mathtt{ret}^{\mathrm{W}_{rel}} = w^m$$

$$\mathtt{bind}^{\mathrm{W}_{rel}}\,(\mathtt{bind}^{\mathrm{W}_{rel}}\ w^m\ w^f)\ w^g = \mathtt{bind}^{\mathrm{W}_{rel}} w^m (\lambda x.\ \mathtt{bind}^{\mathrm{W}_{rel}}\,(w^f\ x)\ w^g)$$

for any $a_1 : A_1, a_2 : A_2, w^f : A_1 \times A_2 \to \mathrm{W}_{rel}(B_1, B_2), w^m : \mathrm{W}_{rel}(A_1, A_2), w^g : B_1 \times B_2 \to \mathrm{W}_{rel}(C_1, C_2)$.

A simple way to produce various examples of simple relational specification monads besides $\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}$ is to start from a non-relational specification monad W in the sense of Maillard et al. [2019], that is a monad equipped with a compatible order, and to compose it with the function $(A_1, A_2) \mapsto A_1 \times A_2$. A result of Altenkirch et al. [2015] (prop. 2.3.(1)) then ensures that $\mathrm{W}_{\mathrm{rel}}(A_1, A_2) = \mathrm{W}(A_1 \times A_2)$ is a simple relational specification monad. In the following paragraphs, we illustrate this construction with a few concrete instances showing the flexibility of this construction. Depending on the property we want to verify and the desired verification style, we can pick relational specification monads among

many different alternatives. For instance, choosing a simpler relational specification monad can often simplify verification, but also have less expressive power than more sophisticated variants. Similarly, relational weakest preconditions are better suited for (semi-)automated verification, but relational pre-/postconditions are more intuitive to humans and make the connection to established relational program logics more evident.

Backward predicate transformer. A stateless version of $\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}$ is the predicate transformer

$$\mathrm{W}^{\mathrm{Pure}}_{\mathrm{rel}}(A_1, A_2) = (A_1 \times A_2 \to \mathbb{P}) \to \mathbb{P}$$

equipped with monadic operations and order derived from the monotonic continuation monad. We call this simple relational specification monad Pure because it naturally applies to the relational verification of pure code, however it can also be useful to verify effectful code as we will see for nondeterministic computations in §2.6.

Pre-/postconditions. Specifications written in terms of pre-/postconditions are simpler to understand than their predicate transformer equivalents. We show that relational specifications written as pre-/postcondition also form a relational specification monad. The type constructor

$$\mathrm{PP}^{\mathrm{Pure}}_{\mathrm{rel}}(A_1, A_2) = \mathbb{P} \times (A_1 \times A_2 \to \mathbb{P})$$

models a pair consisting of a precondition in $\mathbb{P}$ and a postcondition, that is a relation on final values of two computations. There is a natural ordering between such pairs, namely

$$(pre_1, post_1) \leq^{\mathrm{PP}^{\mathrm{St}}_{\mathrm{rel}}} (pre_2, post_2) \quad \Longleftrightarrow \quad \begin{array}{l} pre_2 \Rightarrow pre_1 \ \wedge \\ \forall (a_1 : A_1)(a_2 : A_2).post_1(a_1, a_2) \Rightarrow post_2(a_1, a_2). \end{array}$$

The monadic structure is given by

$\texttt{let ret}^{\mathrm{PP}^{\mathrm{Pure}}_{\mathrm{rel}}}\ (a_1, a_2) = (\ \top,\ \lambda(a_1', a_2').\ a_1 = a_1' \wedge a_2 = a_2'\ )$

$\texttt{let bind}^{\mathrm{PP}^{\mathrm{Pure}}_{\mathrm{rel}}}\ (pre, post)\ f =$

$\quad \texttt{let}\ pre' = pre \wedge \forall a_1, a_2\ .\ post\ (a_1, a_2) \Longrightarrow \pi_1\ (f(a_1, a_2))\ \texttt{in}$

$\quad \texttt{let}\ post'\ (b_1, b_2) = \exists a_1, a_2\ .\ post\ (a_1, a_2) \wedge \pi_2\ (f(a_1, a_2))\ (b_1, b_2)\ \texttt{in}$

$\quad (pre', post')$

The return operation results in a trivial precondition and a postcondition holding exactly for the given arguments, whereas $\texttt{bind}^{\mathrm{PP}^{\mathrm{Pure}}_{\mathrm{rel}}}$ strengthens the precondition of its first argument so that the postcondition of the first computation entails the precondition of the continuation.

Stateful pre-/postconditions. Continuing on pre-/postconditions, we consider a stateful variant of $\mathrm{PP}^{\mathrm{Pure}}_{\mathrm{rel}}$:

$$\mathrm{PP}^{\mathrm{St}}_{\mathrm{rel}}(A_1, A_2) = (S_1 \times S_2 \to \mathbb{P}) \times ((S_1 \times A_1 \times S_1) \times (S_2 \times A_2 \times S_2) \to \mathbb{P})$$

These are pairs, where the first component consists of a precondition on a pair of initial states, one for each sides, while the second component is a postcondition formed by a relation on triples of an initial state, a final value and a final state.

The simple relational monadic specification structure is similar to the one of $\mathrm{PP}^{\mathrm{Pure}}_{\mathrm{rel}}$, threading in the state where necessary, and specifying that the initial state does not change for return:

$$\texttt{let ret}^{\mathrm{PP}^{\mathrm{St}}_{\mathrm{rel}}}\ (a_1,a_2) = (\lambda(s_1,\ s_2)\ .\ \top,\quad \lambda((s_1^i,\ a_1',\ s_1^f),(s_1^i,\ a_2',\ s_2^f))\ .\ a_1 = a_1' \wedge a_2 = a_2' \wedge s_1^i = s_1^f \wedge s_2^i = s_2^f).$$

There is a natural embedding of stateful pre-/postconditions $(pre, post) : \mathrm{PP}^{\mathrm{St}}_{\mathrm{rel}}(A_1, A_2)$ into stateful backward predicate transformers $\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}(A_1, A_2)$ given by

$$\lambda\varphi\ (s_1^i, s_2^i).\ pre(s_1^i, s_2^i) \wedge \forall a_1, a_2, s_1^f, s_2^f\ .post\ ((s_1^i, a_1, s_1^f), (s_2^i, a_2, s_2^f)) \Rightarrow \varphi\ ((a_1, s_1^f), (a_2, s_2^f)) : \mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}(A_1, A_2).$$

Errorful backward predicate transformer. While exceptions turn out to be complex in general, a coarse approach is still possible using the simple relational monad

$$\mathrm{W}^{\mathrm{Err}}_{\mathrm{rel}}(A_1, A_2) = ((A_1 \times A_2 + \mathbb{1}) \to \mathbb{P}) \to \mathbb{P}. \tag{12}$$

This construction represents a predicate transformer that works on either successful computations, or on an indication that at least one of the computations threw an exception, but losing the information of which of the two sides raised the exception. We can actually show that, under mild assumptions, no simple relational specification monad accounting for exceptions can distinguish the three situations where the left, the right, or both programs are raising exceptions. Intuitively, this is due to the fact that the two programs are supposed to run independently, but the simple relational specification monad impose some amount of synchronization. We return to $\mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}$ from §1 and solve this problem in §3, while previous relational program logics have generally been stuck with weak specification monads in the style of $\mathrm{W}^{\mathrm{Err}}_{\mathrm{rel}}$ above [Barthe et al. 2016].

Input-output backward predicate transformer. A relational specification monad similar to $\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}$ can be used to specify interactive I/O computations. For relating two computational monads on input-output sets (I_1, O_2) and (I_2, O_2), we use

$$\mathrm{W}^{\mathrm{IO}}_{\mathrm{rel}}(A_1, A_2) = (A_1 \times A_2 \to \mathrm{list}(\mathcal{E}_1) \times \mathrm{list}(\mathcal{E}_2) \to \mathbb{P}) \to \mathrm{list}(\mathcal{E}_1) \times \mathrm{list}(\mathcal{E}_2) \to \mathbb{P} \tag{13}$$

where $\mathcal{E}_1 = I_1 + O_1$ and $\mathcal{E}_2 = I_2 + O_2$ represent a log element of possible input-output behaviour. Intuitively, a specification of type $\mathrm{W}^{\mathrm{IO}}_{\mathrm{rel}}(A_1, A_2)$ is a backward predicate transformer that transforms a postcondition on the output results and the I/O history into a precondition describing the I/O history before running the computations. Alternative relational specification monads for input-output are easily defined following the discussion in Maillard et al. [2019].

Quantitative predicate transformers. The backward predicate transformer $\mathrm{W}^{\mathrm{Pure}}_{\mathrm{rel}}$ generalizes to quantitative settings were propositions are replaced by a notion of resource. We use a particular case of this generalization as the relational specification monad for probabilities, restricting to monotonic additive continuous[2] maps in the type [Audebaud and Paulin-Mohring 2006; Faissole and Spitters 2017]

$$\mathrm{W}^{\mathrm{Prob}}_{\mathrm{rel}}(A_1, A_2) = (A_1 \times A_2 \to \mathbb{I}) \to \mathbb{I}$$

2.3 Relational Semantics from Effect Observations

The relational judgment $\vdash\ c_1 \sim c_2\ \{\ w\ \}$ should assert that monadic computations $c_1 : \mathrm{M}_1 A_1$ and $c_2 : \mathrm{M}_2 A_2$ satisfy a relational specification $w : \mathrm{W}_{\mathrm{rel}}(A_1, A_2)$ drawn from a simple relational specification monad. What does this judgment mean in our semantic framework? Certainly it requires a specific connection between the computational monads M_1, M_2 and the simple relational specification monad $\mathrm{W}_{\mathrm{rel}}$. In the non-relational setting, this is accomplished by an *effect observation*, i.e., a monad morphism from the computational monad to the specification monad [Katsumata 2014; Maillard et al. 2019]. An effect observation accounts for the various choices available when specifying a particular effect, for instance total or partial correctness in the case of errors or recursion, angelic or demonic interpretations of nondeterministic computations, or connecting ghost state with actual state or with past IO events. In the relational setting, we introduce *relational effect observations*, families of functions respecting the monadic structure, defined here from first principles, but arising as an extension of monad morphisms as we will show in §3.4.

Definition 2. *A* simple lax relational effect observation θ_{rel} *from computational monads* $\mathrm{M}_1, \mathrm{M}_2$ *to a simple relational specification monad* W_{rel} *is given by*

- *for each pair of types* A_1, A_2 *a function* $\theta_{rel} : \mathrm{M}_1\, A_1 \times \mathrm{M}_2\, A_2 \to \mathrm{W}_{rel}(A_1, A_2)$
- *such that*

$$\theta_{rel}\,(\mathtt{ret}^{\mathrm{M}_1}\, a_1, \mathtt{ret}^{\mathrm{M}_2}\, a_2) \leq^{\mathrm{W}_{rel}} \mathtt{ret}^{\mathrm{W}_{rel}}\,(a_1, a_2)$$

$$\theta_{rel}\,(\mathtt{bind}^{\mathrm{M}_1}\, m_1\, f_1, \mathtt{bind}^{\mathrm{M}_2}\, m_2\, f_2) \leq^{\mathrm{W}_{rel}} \mathtt{bind}^{\mathrm{W}_{rel}}\,(\theta_{rel}\,(m_1, m_2))\,(\theta_{rel} \circ (f_1, f_2))$$

[2] as maps between ω-cpo

We say that θ_{rel} is a simple strict relational effect observation *if these two laws hold with equality.*

As explained in the introduction, for stateful computations a simple strict relational effect observation targeting $\mathrm{W}_{\mathrm{rel}}^{\mathrm{St}}$ runs the two computations and passes the results to the postcondition:

$$\theta_{\mathrm{rel}}^{\mathrm{St}}(c_1, c_2) = \lambda\varphi\,(s_1, s_2).\ \varphi(c_1\, s_1, c_2\, s_2). \tag{14}$$

A more interesting situation happens when interpreting nondeterministic computations $(c_1, c_2) : \mathrm{Nd}\, A_1 \times \mathrm{Nd}\, A_2$ into the relational specification monad $\mathrm{W}_{\mathrm{rel}}^{\mathrm{Pure}}(A_1, A_2)$. Two natural simple strict relational effect observations are given by

$$\theta_{\mathrm{rel}}^{\forall}(c_1, c_2) = \lambda\varphi.\ \forall a_1 \in c_1, a_2 \in c_2.\ \varphi(a_1, a_2), \quad \theta_{\mathrm{rel}}^{\exists}(c_1, c_2) = \lambda\varphi.\ \exists a_1 \in c_1, a_2 \in c_2.\ \varphi(a_1, a_2). \tag{15}$$

The first one $\theta_{\mathrm{rel}}^{\forall}$ prescribes that all possible results from the left and right computations have to satisfy the relational specification, corresponding to a demonic interpretation of nondeterminism, whereas the angelic $\theta_{\mathrm{rel}}^{\exists}$ requires at least one final value on each sides to satisfy the relation.

These examples are instances of the following theorem, which allows to lift unary effect observations to simple strict relational effect observations. To state it, we first recall that two monadic computations $c_1 : \mathrm{M}\, A_1$ and $c_2 : \mathrm{M}\, A_2$ *commute* [Bowler et al. 2013; Führmann 2002] when

$$\mathtt{bind}^{\mathrm{M}}\ c_1\ \left(\lambda a_1.\ \mathtt{bind}^{\mathrm{M}}\ c_2\ \left(\lambda a_2.\ \mathtt{ret}^{\mathrm{M}}(a_1, a_2)\right)\right) = \mathtt{bind}^{\mathrm{M}}\ c_2\ \left(\lambda a_2.\ \mathtt{bind}^{\mathrm{M}}\ c_1\ \left(\lambda a_1.\ \mathtt{ret}^{\mathrm{M}}(a_1, a_2)\right)\right).$$

The intuition is that executing c_1 and then c_2 is the same as executing c_2 and then c_1.

Theorem 1. *Let $\theta_1 : \mathrm{M}_1 \to \mathrm{W}$ and $\theta_2 : \mathrm{M}_2 \to \mathrm{W}$ be unary effect observations, where M_1 and M_2 are computational monads and W is a (unary) specification monad. We denote with $\mathrm{W}_{rel}(A_1, A_2) = \mathrm{W}\,(A_1 \times A_2)$ the simple strict relational specification monad derived from W (see §2.2). If for all $c_1 : \mathrm{M}_1\, A_1$ and $c_2 : \mathrm{M}_2\, A_2$, we have that $\theta_1(c_1)$ and $\theta_2(c_2)$ commute, then the following function $\theta_{rel} : \mathrm{M}_1\, A_1 \times \mathrm{M}_2\, A_2 \to \mathrm{W}_{rel}(A_1, A_2)$ is a simple relational effect observation*

$$\theta_{rel}(c_1, c_2) = \mathtt{bind}^{\mathrm{W}}\ \theta_1(c_1)\ \left(\lambda a_1.\ \mathtt{bind}^{\mathrm{W}}\ \theta_2(c_2)\ \left(\lambda a_2.\ \mathtt{ret}^{\mathrm{W}}(a_1, a_2)\right)\right).$$

Moreover, a partial converse for this theorem holds: given a simple relational effect observation $\theta_{\mathrm{rel}} : \mathrm{M}_1\, A_1 \times \mathrm{M}_2\, A_2 \to \mathrm{W}_{\mathrm{rel}}(A_1, A_2)$ where $\mathrm{W}_{\mathrm{rel}}$ is a lifting of a unary specification monad (i.e., $\mathrm{W}_{\mathrm{rel}}(A_1, A_2) = \mathrm{W}(A_1 \times A_2)$), then there exist commuting unary effect observations $\theta_1 : \mathrm{M}_1 \to \mathrm{W}$ and $\theta_2 : \mathrm{M}_2 \to \mathrm{W}$ such that θ_{rel} is equal to the simple relational effect observation obtained from applying Theorem 1 to these.

Another class of examples of *lax* effect observations, covering for instance the refinement observation for nondeterminism $\theta_{\mathrm{rel}}^{\forall\exists}(c_1, c_2) = \lambda\varphi.\ \forall a_1 \in c_1.\ \exists a_2 \in c_2.\ \varphi(a_1, a_2)$ from §1, is provided by the following theorem that connects *lax* effect observations to *relators* Γ over the monad M [Dal Lago et al. 2017; Gavazzo 2018] which lift relations on values to relations on monadic computations:

$$\Gamma \quad : \quad (A_1 \times A_2 \to \mathbb{P}) \longrightarrow MA_1 \times MA_2 \to \mathbb{P}.$$

Theorem 2. *A relator Γ over a monad M induces a simple lax relational effect observation of the form $\theta_{rel}^{\Gamma} : MA_1 \times MA_2 \to \mathrm{W}_{rel}^{\mathrm{Pure}}(A_1, A_2)$.*

Proof. The carrier of $\theta_{\mathrm{rel}}^{\Gamma} : M\, A_1 \times M\, A_2 \to (A_1 \times A_2 \to \mathbb{P}) \to \mathbb{P}$ is obtained by swapping the arguments of Γ, while the two inequalities are direct consequences of the compatibility of the relator Γ with the monad M. □

Relators provide interesting examples of relational effect observations for nondeterminism and for probabilities. Relational effect observation extend relators by providing the possibility of relating two different computational monads, as well as having more sophisticated specifications with ghost state or exceptional postconditions. Conversely, relators preserve – in a lax sense – identities and relational composition.

In general, given a simple lax relational effect observation $\theta_{\text{rel}} : \mathrm{M}_1, \mathrm{M}_2 \to \mathrm{W}_{\text{rel}}$, we define the semantic relational judgment by

$$\vDash_{\theta_{\text{rel}}} c_1 \sim c_2 \ \{ w \} \quad = \quad \theta_{\text{rel}}\,(c_1, c_2) \leq^{\mathrm{W}_{\text{rel}}} w, \tag{16}$$

where we make use of the preorder given by W_{rel}. The following 3 subsections explain how to derive sound rules for a relational logic parameterized by the computational monads $\mathrm{M}_1, \mathrm{M}_2$, the simple relational specification monad W_{rel}, and the simple lax relational effect observation θ_{rel}.

2.4 Pure Relational Rules

We start with rules coming from the ambient dependent type theory. Even though the semantics of the relational judgment depends on the choice of an effect observation, the soundness of the basic pure rules introduced in Figure 1 is independent from both the computational monads and effects observation. Indeed, the proof of soundness of these follows from applying the adequate dependent eliminator coming from the type theory.

$$\mathbb{B}\text{-Elim}\ \frac{\mathtt{if}\ b\ \mathtt{then} \vdash c_1 \sim c_2 \ \left\{ w^{\top} \right\}\ \mathtt{else} \vdash c_1 \sim c_2 \ \left\{ w^{\bot} \right\}}{\vdash c_1 \sim c_2 \ \left\{ \mathtt{if}\ b\ \mathtt{then}\ w^{\top}\ \mathtt{else}\ w^{\bot} \right\}} \qquad \mathbb{0}\text{-Elim}^{2}\ \frac{w \leq \bot}{\vdash c_1 \sim c_2 \ \{ w \}}$$

$$\mathbb{N}\text{-Elim}\ \frac{\begin{array}{c} n : \mathbb{N} \qquad w = \mathtt{elim}^{\mathbb{N}}\ w_0\ w_{suc} \qquad \vdash c_1[0/n] \sim c_2[0/n] \ \{ w_0 \} \\ \forall n : \mathbb{N}, \vdash c_1 \sim c_2 \ \{ w\, n \} \quad \Rightarrow \quad \vdash c_1[\mathtt{S}\, n/n] \sim c_2[\mathtt{S}\, n/n] \ \{ w_{suc}\, (w\, n) \} \end{array}}{\vdash c_1 \sim c_2 \ \{ w\ n \}}$$

Fig. 1. Pure relational rules

These rules can then be tailored as explained in the introduction to derive asynchronous (1) or synchronous (2) rules more suited for applications. For some of the derived rules, there is, however, an additional requirement on the simple relational specification monad, so that we can strengthen preconditions. This small mismatch in the theory, already present in the unary setting of Maillard et al. [2019] on top of which we work, could be solved by adopting a richer definition of specification monads, for instance taking inspiration in the work of Gavazzo [2018], and is left as future work.

2.5 Generic Monadic Rules

Given any computational monads $\mathrm{M}_1, \mathrm{M}_2$ and a simple relational specification monad W_{rel}, we introduce three rules governing the monadic part of a relational program logic (Figure 2). Each of

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

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

Fig. 2. Generic monadic rules in the simple framework

these rules directly corresponds to one aspect of the simple relational specification monad and are

[2] Assuming that W_{rel} contains a top element $\bot$ that entails falsity of the precondition; this is the case for all our examples.

all synchronous. As explained in the introduction (5), it is then possible to derive asynchronous variants using the monadic laws of the computational monads.

Theorem 3 (Soundness of generic monadic rules). *The relational rules in Figure 2 are sound with respect to any lax relational effect observation θ_{rel}, that is*

$$\vdash c_1 \sim c_2 \ \{ w \} \qquad \Rightarrow \qquad \forall \theta_{rel},\ \vDash_{\theta_{rel}} c_1 \sim c_2 \ \{ w \}\,.$$

Proof. For rules Ret and Bind, we need to prove that $\theta_{\text{rel}}(\mathtt{ret}^{\mathrm{M}_1}\, a_1, \mathtt{ret}^{\mathrm{M}_2} a_2) \leq \mathtt{ret}^{\mathrm{W}}(a_1, a_2)$ and $\theta_{\text{rel}}\,(\mathtt{bind}^{\mathrm{M}_1}\, m_1\, f_1, \mathtt{bind}^{\mathrm{M}_2}\, m_2\, f_2) \leq \mathtt{bind}^{\mathrm{W}}\,(\theta_{\text{rel}}\,(m_1, m_2))\,(\theta_{\text{rel}} \circ (f_1, f_2))$, which are exactly the laws of a lax relational effect observation. For Weaken, we need to show that $\theta_{\text{rel}}(c_1, c_2) \leq w'$ under the assumptions that $\theta_{\text{rel}}(c_1, c_2) \leq w$ and $w \leq w'$ so we conclude by transitivity. □

2.6 Effect-Specific Rules

The generic monadic rules together with the rules coming from the ambient type theory allow to derive relational judgments for the main structure of the programs. However, these rules are not enough to handle full programs written in the computational monads M_1 and M_2, as we also need rules to reason about the specific effectful operations that these monads provide. The soundness of effect-specific relational rules is established with respect to a *particular* choice of relational effect observation $\theta_{\text{rel}} : \mathrm{M}_1, \mathrm{M}_2 \longrightarrow \mathrm{W}_{\text{rel}}$. Consequently, we make essential use of θ_{rel} to devise effect-specific rules. The recipe was already illustrated for state in the introduction: first pick a pair of effectful *algebraic* operations (or `ret` for the asynchronous rules), unfold their definition, and then compute a sound-by-design relational specification for this pair by simply applying θ_{rel}. By following this recipe, we are decoupling the problem of choosing the computations on which these rules operate (e.g., synchronous vs. asynchronous rules to which we return in §5) from the problem of choosing sensible specifications, which is captured in the choice of θ_{rel}.

Nondeterministic computations. The two relational effect observations $\theta^{\forall}_{\text{rel}}$ and $\theta^{\exists}_{\text{rel}}$ provide different relational rules for the operation `pick`. As an example of how the recipe works, suppose that we want to come up with an asymmetric rule for nondeterministic computations that works on the left program, and which is sound with respect to $\theta^{\forall}_{\text{rel}}$. This means that the conclusion will be of the form $\vdash \mathtt{pick} \sim \mathtt{ret}\, a_2 \ \left\{ w_{\mathtt{pick}^l} \right\}$ for some $w_{\mathtt{pick}^l} : \mathrm{PP}^{\mathrm{Pure}}_{\text{rel}}$. To obtain $w_{\mathtt{pick}^l}$, we apply the effect observation to the two computations involved in the rule

$$w_{\mathtt{pick}^l} = \theta^{\forall}_{\text{rel}}\,(\mathtt{pick}, \mathtt{ret}\, a_2) = \lambda\varphi.\ \forall b \in \{\mathtt{tt}, \mathtt{ff}\}, a \in \{a_2\}.\ \varphi(b, a) = \lambda\varphi.\ \varphi(\mathtt{tt}, a_2) \wedge \varphi(\mathtt{ff}, a_2),$$

obtaining a rule that is trivially sound:

$$\textsc{DemonicPickLeft}\ \frac{}{\vdash \mathtt{pick} \sim \mathtt{ret}\, a_2 \ \{ \lambda\varphi.\ \varphi(\mathtt{tt}, a_2) \wedge \varphi(\mathtt{ff}, a_2) \}}\,.$$

Similarly for `fail`, we compute $w_{\mathtt{fail}^l}$ for $\vdash \mathtt{fail} \sim \mathtt{ret}\, a_2 \ \left\{ w_{\mathtt{fail}^l} \right\}$ as follows:

$$w_{\mathtt{fail}^l} = \theta^{\forall}_{\text{rel}}\,(\mathtt{fail}, \mathtt{ret}\, a_2) = \lambda\varphi.\ \forall b \in \{\mathtt{tt}, \mathtt{ff}\}, a \in \emptyset.\ \varphi(b, a) = \lambda\varphi.\ \top,$$

$$\textsc{DemonicFailLeft}\ \frac{}{\vdash \mathtt{fail} \sim \mathtt{ret}\, a_2 \ \{ \lambda\varphi.\ \top \}}\,.$$

Following the same approach, we can come up with an asymmetric rule on the right as well as a symmetric one. For concreteness, we show the symmetric rule for the effect observation $\theta^{\exists}_{\text{rel}}$:

$$\textsc{Angelic}\ \frac{}{\vdash \mathtt{pick} \sim \mathtt{pick} \ \{ \lambda\varphi.\ \varphi(\mathtt{tt}, \mathtt{tt}) \vee \varphi(\mathtt{tt}, \mathtt{ff}) \vee \varphi(\mathtt{ff}, \mathtt{tt}) \vee \varphi(\mathtt{ff}, \mathtt{ff}) \}}\,.$$

Taking inspiration from the sample rule in [Barthe et al. 2015a], we introduce a rule for the refinement effect observation $\theta^{\forall\exists}_{\text{rel}}$ using an auxilliary function to select the elements in correspondence:

$$\textsc{Refinement}\ \frac{h : \mathtt{fin}\, n \longrightarrow \mathtt{fin}\, m}{\vdash \mathtt{choose}\, n \sim \mathtt{choose}\, m \ \{ \lambda\varphi.\ \forall k. \varphi(k, h\, k) \}}\,.$$

Exceptions using $\mathrm{W}^{\mathrm{Err}}_{\mathrm{rel}}$. Taking M_1 and M_2 to be exception monads on exception sets E_1 and E_2, and the relational specification monad $\mathrm{W}^{\mathrm{Err}}_{\mathrm{rel}}$ (Equation 12 on page 11), we have an effect observation interpreting any thrown exception as a unique erroneous termination situation, that is

$$\begin{array}{l}\texttt{let}\ \theta^{\mathrm{Err}}_{\mathrm{rel}}\ ((c_1, c_2) : \mathrm{Exc}\ A_1 \times \mathrm{Exc}\ A_2) : \mathrm{W}^{\mathrm{Err}}_{\mathrm{rel}}(A_1, A_2) = \\ \quad \lambda\varphi.\ \texttt{match}\ c_1, c_2\ \texttt{with} \mid \mathrm{Inl}\ a_1, \mathrm{Inl}\ a_2 \to \varphi\ (\mathrm{Inl}\ (a_1, a_2)) \mid \_, \_ \to \varphi\ (\mathrm{Inr}\ ())\end{array}$$

Under this interpretation we can show the soundness of the following rules:

$$\textsc{ThrowL}\ \frac{}{\vdash \mathtt{throw}\, e_1 \sim \mathtt{ret}\, a_2\ \{\ \lambda\varphi.\ \varphi(\mathtt{Inr}\ ())\ \}} \qquad \textsc{ThrowR}\ \frac{}{\vdash \mathtt{ret}\, a_1 \sim \mathtt{throw}\, e_2\ \{\ \lambda\varphi.\ \varphi(\mathtt{Inr}\ ())\ \}}$$

$$\textsc{Catch}\ \frac{\vdash c_1 \sim c_2\ \{\ w\ \} \qquad \forall e_1\ e_2 \vdash c_1^{\maltese}\ e_1 \sim c_2^{\maltese}\ e_2\ \{\ w^{\maltese}\ \} \\ \forall e_1\ a_2 \vdash c_1^{\maltese}\ e_1 \sim \mathtt{ret}\, a_2\ \{\ w^{\maltese}\ \} \qquad \forall a_1\ e_2 \vdash \mathtt{ret}\, a_1 \sim c_2^{\maltese}\ e_2\ \{\ w^{\maltese}\ \}}{\vdash \mathtt{catch}\, c_1\, c_1^{\maltese} \sim \mathtt{catch}\, c_2\, c_2^{\maltese}\ \{\ \lambda\varphi.w(\lambda a_0.\ \mathtt{match}\ a_0\ \mathtt{with}\ \mathrm{Inl}\ a \to \varphi\ (\mathrm{Inl}\ a) \mid \mathrm{Inr}\ () \to w^{\maltese}\ \varphi)\ \}}$$

The rules ThrowL and ThrowR can be derived using the recipe above, but the exceptions have to be conflated to the same exceptional result $\mathtt{Inr}\ ()$, a situation that is forced by the choice of relational effect observation and a weak specification monad. As a consequence, the Catch rule considers one successful case and three exceptional cases. The specification in the conclusion takes a postcondition φ and computes a precondition by running the transformer w on a new postcondition that depends on the result of c_1 and c_2. If both computations were successful, then this new postcondition is simply the original φ. If an exception was thrown (in any of the sides or both), then the new postcondition is computed using the transformer $w^{\maltese}$, which specifies the three exceptional cases. The specification for Catch does not follow mechanically from $\theta^{\mathrm{Err}}_{\mathrm{rel}}$ using our recipe, since it is a handler and not an algebraic operation.

Input-output computations. Let M_1 and M_2 be the input-output monads on (I_1, O_1) and (I_2, O_2) respectively (§2.1). We want an effect observation on the relational specification monad $\mathrm{W}^{\mathrm{IO}}_{\mathrm{rel}}$ (Equation 13 on page 12):

$$\theta^{\mathrm{IO}}_{\mathrm{rel}} : \mathrm{M}_1\ A_1 \times \mathrm{M}_2\ A_2 \to \mathrm{W}^{\mathrm{IO}}_{\mathrm{rel}}(A_1, A_2)$$

Notice that $\mathrm{W}^{\mathrm{IO}}_{\mathrm{rel}}(A_1, A_2) = \mathrm{W}^{\mathrm{IO}}(A_1 \times A_2)$, where W^{IO} is a unary specification monad defined by

$$\mathrm{W}^{\mathrm{IO}}(A) = (A \to \mathrm{list}(\mathcal{E}_1) \times \mathrm{list}(\mathcal{E}_2) \to \mathbb{P}) \to \mathrm{list}(\mathcal{E}_1) \times \mathrm{list}(\mathcal{E}_2) \to \mathbb{P}$$

By applying Theorem 1 to unary effect observations $\theta^{\mathrm{IO}}_1 : \mathrm{M}_1 \to \mathrm{W}^{\mathrm{IO}}$ and $\theta^{\mathrm{IO}}_2 : \mathrm{M}_2 \to \mathrm{W}^{\mathrm{IO}}$, we obtain the desired relational effect observation $\theta^{\mathrm{IO}}_{\mathrm{rel}}$. The unary effect observation θ^{IO}_1 is defined by recursion on the computation trees (θ^{IO}_2 is analogous):

$$\begin{array}{l}\texttt{let rec}\ \theta^{\mathrm{IO}}_1\ (c : \mathrm{M}_1\ A) : \mathrm{W}^{\mathrm{IO}}\ A = \texttt{match}\ c\ \texttt{with} \\ \quad \mid \mathrm{Ret}\ x \to \mathtt{ret}^{\mathrm{W}^{\mathrm{IO}}}\ x \\ \quad \mid \mathrm{Input}\ k \to \mathtt{bind}^{\mathrm{W}^{\mathrm{IO}}}\ (\lambda\ \varphi\ (h_1, h_2)\ .\ \forall i, \varphi\ i\ (\mathrm{Inl}\ i :: h_1, h_2))\ (\lambda\ i\ .\ \theta^{\mathrm{IO}}_1\ (k\ i)) \\ \quad \mid \mathrm{Output}\ o\ k \to \mathtt{bind}^{\mathrm{W}^{\mathrm{IO}}}\ (\lambda\ \varphi\ (h_1, h_2)\ .\ \varphi\ ()\ (\mathrm{Inr}\ o :: h_1, h_2))\ (\lambda\ ()\ .\ \theta^{\mathrm{IO}}_1\ k)\end{array}$$

The relational rules we get by applying our recipe to `input` and `output` are the following:

$$\textsc{InputL}\ \frac{}{\vdash \mathtt{input} \sim \mathtt{ret}\, a_2\ \{\ \lambda\varphi, (h_1, h_2).\ \forall i_1 \in I_1, \varphi\ (i_1, a_2)\ (\mathtt{Inl}\ i_1 :: h_1, h_2)\ \}}$$

$$\textsc{OutputL}\ \frac{}{\vdash \mathtt{output}\ o_1 \sim \mathtt{ret}\, a_2\ \{\ \lambda\varphi, (h_1, h_2).\ \varphi\ ((), a_2)\ (\mathtt{Inr}\ o :: h_1, h_2)\ \}}$$

Unbounded iteration. Specifications for imperative programs as modeled by the Imp monad from §2.1 come in two flavors. This is reflected here by two unary effect observations: a first one for total correctness θ^{Tot} *ensuring* the termination of a program; and a second one for partial correctness θ^{Part} *assuming* the termination of a program. We explain how this situation extends to the relational setting, focusing on partial correctness, but the same methodology applies to total correctness. Concretely, we define a simple strict relational effect observation

$$\theta^{\mathrm{Part}}_{\mathrm{rel}} : \mathrm{Imp}\, A_1 \times \mathrm{Imp}\, A_2 \to \mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}(A_1, A_2)$$

by applying Theorem 1 to a unary effect observation θ^{Part} defined using the domain structure with which W^{St} is naturally endowed. From basic domain theoretic results, W^{St} can be endowed with a least fixpoint combinator $\mathtt{fix} : (\mathrm{W}^{\mathrm{St}}\,\mathbb{B} \to \mathrm{W}^{\mathrm{St}}\,\mathbb{B}) \to \mathrm{W}^{\mathrm{St}}\,\mathbb{B}$, used to define

$$\begin{array}{l}
\mathtt{let}\ \theta^{\mathrm{Part}}\ (c : \mathrm{Imp}\ A) : \mathrm{W}^{\mathrm{St}}\ A = \mathtt{match}\ c\ \mathtt{with} \\
\quad \mid \mathrm{Ret}\ x \to \mathtt{ret}^{\mathrm{W}^{\mathrm{St}}}\ x \qquad \mid \mathrm{Get}\ k \to \lambda \varphi\ s\ .\ \theta^{\mathrm{Part}}\ (k\ s)\ \varphi\ s \qquad \mid \mathrm{Put}\ s'\ k \to \lambda \varphi\ s\ .\ \theta^{\mathrm{Part}}\ k\ \varphi\ s' \\
\quad \mid \mathrm{DoWhile}\ body\ k \to \\
\qquad \mathtt{let}\ loop\ (w : \mathrm{W}^{\mathrm{St}}\ \mathbb{B}) = \mathtt{bind}^{\mathrm{W}^{\mathrm{St}}}\ (\theta^{\mathrm{Part}}\ body)\ (\lambda\ b.\ \mathtt{if}\ b\ \mathtt{then}\ w\ \mathtt{else}\ \mathtt{ret}^{\mathrm{W}^{\mathrm{St}}}\ \mathtt{ff})\ \mathtt{in} \\
\qquad \mathtt{bind}^{\mathrm{W}^{\mathrm{St}}}\ (\mathtt{fix}\ loop)\ (\lambda\ _\ .\ \theta^{\mathrm{Part}}\ k)
\end{array}$$

How does θ^{Part} work? In the first three cases, it trivially returns in the Ret branch, evaluates a continuation to the current state in the Get branch, and evaluates a continuation with an updated state in the Put branch. The interesting part is in the DoWhile branch, where the body is repeatedly run using `fix` as long as the guard returns `tt`. We proved by induction on c that θ^{Part} is a monad morphism. Theorem 1 asks for two monad morphisms whose images commute. We provided those morphisms by tweaking a bit the definition of θ^{Part}: we embed in a variation of W^{St} that accounts for a pair of states, the first $\theta^{\mathrm{Part}}_1 : \mathrm{Imp} \to \mathrm{W}^{\mathrm{St}}$ uses the left state and the second $\theta^{\mathrm{Part}}_2 : \mathrm{Imp} \to \mathrm{W}^{\mathrm{St}}$ uses the right state. Applying Theorem 1, we obtain the definition of $\theta^{\mathrm{Part}}_{\mathrm{rel}}$.

This simple relational effect observation $\theta^{\mathrm{Part}}_{\mathrm{rel}}$ captures partial correctness in the following sense: intuitively, $\vDash_{\theta^{\mathrm{Part}}_{\mathrm{rel}}} \{\ \psi\ \}\ c_1 \sim c_2\ \{\ \varphi\ \}$ implies that if $\psi(s_1, s_2)$ holds and the *two* programs c_1 and c_2 terminate on these initial states s_1, s_2, then the postcondition holds of the final states. This judgment using pre-/postconditions is expressed in terms of the usual judgment by applying the translation into stateful backward predicate transformers (see §2.2). On top of this $\theta^{\mathrm{Part}}_{\mathrm{rel}}$, we devise a rule for `do_while` using an invariant $\mathtt{inv}_{b_1,b_2} : S \times S \to \mathbb{P}$:

$$\frac{\vdash \{\ \mathtt{inv}_{\mathtt{tt},\mathtt{tt}}\ \}\ \mathtt{body}_1 \sim \mathtt{body}_2\ \{\ \lambda(\_, b_1, s_1)\,(\_, b_2, s_2).\ b_1 = b_2 \wedge \mathtt{inv}_{b_1,b_2}(s_1, s_2)\ \}}{\vdash \{\ \mathtt{inv}_{\mathtt{tt},\mathtt{tt}}\ \}\ \mathtt{do\_while}\,\mathtt{body}_1 \sim \mathtt{do\_while}\,\mathtt{body}_2\ \{\ \lambda(\_, (), s_1)\,(\_, (), s_2).\ \mathtt{inv}_{\mathtt{ff},\mathtt{ff}}(s_1, s_2)\ \}} \tag{17}$$

This rule is synchronous in the sense that the bodies always yield the same boolean values. Consequently the two loops run the same number of steps. The postcondition ensures that if the loop terminates, then the invariant $\mathtt{inv}_{\mathtt{ff},\mathtt{ff}}$ holds.

Probabilistic computations. For discrete probabilistic computations modeled by the monad Prob, a first idea would be to use a unary effect observation and appeal once again to Theorem 1. This simple strict relational effect observation however does not validate a rule correlating two `flip` operations with an arbitrary coupling between the Bernoulli distributions on each side [Barthe et al. 2009]. A posteriori this is not so surprising, since the commutation hypothesis of Theorem 1 implies that the effects on each side are observed in an independent fashion.

Hence we rely on a more sophisticated *lax* relational effect observation $\theta^{\mathrm{Prob}} : \mathrm{Prob}\, A_1 \times \mathrm{Prob}\, A_2 \to \mathrm{W}^{\mathrm{Prob}}(A_1, A_2)$ defined as

$$\theta^{\mathrm{Prob}}(c_1, c_2) = \lambda\varphi.\ \inf_{d \sim c_1, c_2} \sum_{a_1 : A_1,\, a_2 : A_2} d(a_1, a_2) \cdot \varphi(a_1, a_2)$$

where we write $d \sim c_1, c_2$ to specify a *coupling* d of the two distributions c_1 and c_2, i.e., a distribution on $A_1 \times A_2$ such that the marginals satisfy $\mathrm{Prob}(\pi_1)d = c_1$ and $\mathrm{Prob}(\pi_2)d = c_2$. Since we are taking the infimum over all such couplings, the resulting relational effect observation is necessarily lax and the conditions of linearity and continuity imposed on $\mathrm{W}^{\mathrm{Prob}}$ are needed to show the monadic inequalities. Using this relational effect observation we straightforwardly validate the following rule for correlating two sampling operations as in (×)pRHL [Barthe et al. 2009, 2017].

$$\frac{d \sim \mathcal{B}_p, \mathcal{B}_q}{\vdash \mathtt{flip}\, p \sim \mathtt{flip}\, q \ \left\{ \lambda\varphi.\ \sum_{b_1,b_2:\mathbb{B}} d(b_1,b_2) \cdot \varphi(b_1,b_2) \right\}}$$

2.7 Example: Noninterference

As a specific example of the simplified framework, we explore *noninterference*, a popular relational property for information flow control systems [Antonopoulos et al. 2017; Banerjee et al. 2016; Barthe et al. 2019; Clarkson and Schneider 2010; Nanevski et al. 2013; Sabelfeld and Myers 2003]. The noninterference property dictates that the public outputs of a program cannot depend on its private inputs. Formally, and in its most basic form, we can capture this property by classifying the store's locations by two security levels: *high* for private information and *low* for public information. By $s =_L s'$ we express that the two stores s and s' are equal for all low locations. We use $s \overset{p}{\rightsquigarrow} s'$ to denote that the execution of a program p on a store s ends in store s'. The noninterference property is then written as

$$\forall s_i, s_i', s_o, s_o'. \quad s_i =_L s_i' \wedge s_i \overset{p}{\rightsquigarrow} s_o \wedge s_i' \overset{p}{\rightsquigarrow} s_o' \implies s_o =_L s_o'$$

A typical solution for enforcing noninterference is to define a static type system which is capable of rejecting obviously interferent programs [Sabelfeld and Myers 2003]. For example, such a type system can rule out interferent programs such as

$$\mathtt{if}\ h > 0\ \mathtt{then}\ l := 1\ \mathtt{else}\ l := 0$$

where h is a high reference and l is a low one. However, the static nature of these type systems restricts the family of programs that we can show noninterferent. A characteristic example of this limitation is the following noninterferent program:

$$\mathtt{if}\ h = 1\ \mathtt{then}\ l := h\ \mathtt{else}\ l := 1$$

Relational program logics such as Benton's [2004] RHL provide a less restrictive framework for proving non-interference, as the proof can rely on information accumulated during the derivation steps. We follow the approach of relational program logics and show how noninterference proofs can be done in our framework. We restrict ourselves to programs with conditionals but without while-loops. In §4.1, we will show a complete embedding of RHL, including iteration. For now though, we assume that we are working with a memory consisting of locations $\mathcal{L} = \{\mathtt{l}, \mathtt{h}\}$ storing natural numbers, and consider the data in h to be private and the data in l to be public. As discussed in §2.1, these stateful computations can be captured using the monad St_S where $S = \mathcal{L} \to \mathbb{N}$. The program above can be represented using this monad as follows:

$$c = \mathtt{let}\ x = \mathtt{get}\ h\ \mathtt{in}\ \mathtt{if}\ x = 1\ \mathtt{then}\ \mathtt{put}\ l\ x\ \mathtt{else}\ \mathtt{put}\ l\ 1 : \mathrm{St}\ \mathbb{1}$$

We instantiate our framework with the computational monad St_S on both sides, and use the simple relational specification monad $\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}$ from §1. The judgment we establish to prove noninterference is

$$\vdash c \sim c \ \left\{ \lambda\varphi\ (s_1^i, s_2^i).\ s_1^i\ \mathtt{l} = s_2^i\ \mathtt{l} \wedge \forall\ s_1^f\ s_2^f . s_1^f\ \mathtt{l} = s_2^f\ \mathtt{l} \implies \varphi\ (((), s_1^f), ((), s_2^f)) \right\}$$

This weakest precondition transformer comes from taking the pre-/postcondition pair

$$\lambda(s_1, s_2).\ s_1\ \mathtt{l} = s_2\ \mathtt{l} : S \times S \to \mathbb{P} \qquad \lambda(s_1^i, (), s_1^f)\ (s_2^i, (), s_2^f).\ s_1^f\ \mathtt{l} = s_2^f\ \mathtt{l} : (S \times \mathbb{1} \times S) \times (S \times \mathbb{1} \times S) \to \mathbb{P}$$

and translating it to its predicate transformer form following the description in §2.2. The proof derivation consists of applying the Bind rule after a weakening, and later applying the asymmetric conditional rules (see page 3) for covering the four cases.

A similar example of noninterference can be done by changing the state monad St for the input-output monad IO described in §2.1. In this case, input and output channels are classified as high or low, and the noninterference policy is spelled out in terms of these by using the specification monad $\mathrm{W}_{\mathrm{rel}}^{\mathrm{IO}}$ or one of its variants.

Finally, an interesting characteristic of our framework is that we can easily adapt the setting to handle more than one effect at the same time. For example, if we are interested in modeling both IO and state with noninterference, then it is enough to apply the state monad transformer to the IO monad, and replace the relational specification monad $\mathrm{W}_{\mathrm{rel}}^{\mathrm{St}}$ by a monad which takes into account the input-output in the specifications as well.

3 GENERIC FRAMEWORK

While the simple framework works well for a variety of effects, it falls short of providing a convincing treatment of control effects such as exceptions. This limitation is due to the fact that simple relational specification monads merge tightly together the specification of two independent computations. We now explain how to overcome these limitations starting with the example of exceptions, and how it leads to working inside a relational dependent type theory. Informed by the generic constructions on relative monads underlying the simple setting, we derive a notion of relational specification monad and relational effect observation in this enriched setting. These relational specification monads require an important amount of operations so we introduce relational specification monad transformers for state and exceptions, simplifying the task of building complex relational specification monad from simpler ones. As a consequence, we can easily combine exceptions with any of the effects already handled by the simplified framework (e.g., state, nondeterminism, IO, and probabilities).

3.1 Exceptional Control Flow in Relational Reasoning

We explained in §2.6 how to prove relational properties of programs raising exceptions, as long as we give up on the knowledge of which program raised an exception at the level of relational specifications. This restriction prevents us from even stating natural specifications such as simulations: "if the left program raises, so does the right one".

In order to go beyond this unsatisfying state of affairs, we consider a type of relational specifications allowing to write specifications consisting of predicate transformers mapping a postcondition on pairs of either a value or an exceptional final state to a proposition:

$$\mathrm{W}_{\mathrm{rel}}^{\mathrm{Exc}}(A_1, A_2) = ((A_1 + E_1) \times (A_2 + E_2) \to \mathbb{P}) \to \mathbb{P}.$$

For instance, the specification of simulation above can be stated as

$$\lambda\varphi.\ \forall ae_1 ae_2.(\mathtt{Inr?}\ ae_1 \Rightarrow \mathtt{Inr?}\ ae_2) \Rightarrow \varphi(ae_1, ae_2) : \mathrm{W}_{\mathrm{rel}}^{\mathrm{Exc}}(A_1, A_2),$$

where $\mathtt{Inr?}\ ae = \mathtt{match}\ ae\ \mathtt{with}\ \mathtt{Inr}\ \_ \to \top | \_ \to \bot$.

As explained in the introduction, this type does not admit a monadic operation $\mathtt{bind}\ w^m\ w^f$ using only a continuation of type $w^f : A_1 \times A_2 \to \mathrm{W}_{\mathrm{rel}}^{\mathrm{Exc}}(B_1, B_2)$ due to the fact that w^m could result in an intermediate pair consisting of a normal value on one side and an exception on the other side. Our solution is to provide to $\mathtt{bind}^{\mathrm{W}_{\mathrm{rel}}^{\mathrm{Exc}}}$ the missing information it needs in such cases. To that purpose, we use the unary specification monads $\mathrm{W}_1^{\mathrm{Exc}} A_1 = (A_1 + E_1 \to \mathbb{P}) \to \mathbb{P}$ and $\mathrm{W}_2^{\mathrm{Exc}} A_2 = (A_2 + E_2 \to \mathbb{P}) \to \mathbb{P}$ to provide independent specifications of each program. With the addition of these, we can write a combinator that relies on the unary specifications when the results of the first computations differ (one raise an exception and the other returns).

$$\texttt{val bind}^{\mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}} : \mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}(A_1,A_2) \to (A_1 \to \mathrm{W}^{\mathrm{Exc}}_1 B_1) \to (A_2 \to \mathrm{W}^{\mathrm{Exc}}_2 B_2) \to (A_1 \times A_2 \to \mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}(B_1,B_2)) \to \mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}(B_1,B_2)$$

$$\begin{array}{l}\texttt{let bind}^{\mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}}\ wm\ (f_1 : A_1 \to ((B_1 + E_1) \to \mathbb{P}) \to \mathbb{P})\ (f_2 : A_2 \to ((B_2 + E_2) \to \mathbb{P}) \to \mathbb{P})\ f = \\ \quad \lambda(\varphi : (B_1 + E_1) \to \mathbb{P}). \\ \quad\quad wm\ (\lambda\ ae : (A_1 + E_1) \times (A_2 + E_2). \\ \quad\quad\quad \texttt{match}\ ae\ \texttt{with} \\ \quad\quad\quad \mid \mathrm{Inl}\ a1, \mathrm{Inl}\ a2 \to f\ a1\ a2\ \varphi \qquad \mid \mathrm{Inl}\ a_1, \mathrm{Inr}\ e_2 \to f_1\ a_1\ (\lambda\ be \to \varphi\ be\ (\mathrm{Inr}\ e_2)) \\ \quad\quad\quad \mid \mathrm{Inr}\ e1, \mathrm{Inr}\ e2 \to \varphi\ (\mathrm{Inr}\ e1, \mathrm{Inr}\ e2) \qquad \mid \mathrm{Inr}\ e_1, \mathrm{Inl}\ a_2 \to f_2\ a_2\ (\lambda\ be \to \varphi\ (\mathrm{Inr}\ e_1)\ be))\end{array}$$

3.2 A Problem of Context

In order to keep track of these unary specifications drawn from $\mathrm{W}^{\mathrm{Exc}}_1$ and $\mathrm{W}^{\mathrm{Exc}}_2$ in the relational proofs, we extend the relational judgment to

$$\vdash c_1\ \{w_1\} \sim c_2\ \{w_2\} \mid w_{\mathrm{rel}}.$$

Here, $w_1 : \mathrm{W}^{\mathrm{Exc}}_1 A_1$ is a unary specification for $c_1 : \mathrm{Exc}_1\, A_1$, symmetrically $w_2 : \mathrm{W}^{\mathrm{Exc}}_2 A_2$ is a unary specification for $c_2 : \mathrm{Exc}_2\, A_2$, and $w_{\mathrm{rel}} : \mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}(A_1, A_2)$ specifies the relation between the programs c_1 and c_2. Using this richer judgment, we would like a rule for sequencing computations as follows, where a bold variable $\boldsymbol{w}$ stands for the triple $(w_1, w_2, w_{\mathrm{rel}})$:

$$\frac{\vdash m_1\ \{w_1^m\} \sim m_2\ \{w_2^m\} \mid w_{\mathrm{rel}}^m \qquad \forall a_1, a_2 \vdash f_1\ a_1\ \{w_1^f\ a_1\} \sim f_2\ a_2\ \{w_2^m\ a_2\} \mid w_{\mathrm{rel}}^f\ a_1\ a_2}{\vdash \texttt{bind}^{\mathrm{Exc}_1}\ m_1\ f_1\ \{\texttt{bind}^{\mathrm{W}^{\mathrm{Exc}}_1}\ w_1^m\ w_1^f\} \sim \texttt{bind}^{\mathrm{Exc}_2}\ m_2\ f_2\ \{\texttt{bind}^{\mathrm{W}^{\mathrm{Exc}}_2}\ w_2^m\ w_2^f\} \mid \texttt{bind}^{\mathrm{W}^{\mathrm{Exc}}_{\mathrm{rel}}}\ \boldsymbol{w^m}\ \boldsymbol{w^f}}$$

What would the semantics of such a relational judgment be? A reasonable answer at first sight is to state formally the previous intuition in terms of unary and relational effect observations:

$$\vDash c_1\ \{w_1\} \sim c_2\ \{w_2\} \mid w_{\mathrm{rel}} \quad = \quad \theta^{\mathrm{Exc}}_1\, c_1 \le w_1 \ \wedge\ \theta^{\mathrm{Exc}}_2\, c_2 \le w_2 \ \wedge\ \theta^{\mathrm{Exc}}_{\mathrm{rel}}(c_1, c_2) \le w_{\mathrm{rel}}$$

However this naive attempt does not validate the rule for sequential composition above. The problem lies in the management of context. To prove the soundness of this rule, we have in particular to show that $\theta^{\mathrm{Exc}}_1\, (\texttt{bind}^{\mathrm{Exc}}_1\, m_1\, f_1) \le \texttt{bind}^{\mathrm{W}^{\mathrm{Exc}}_1}\ w_1^m\ w_1^f$ under the hypothesis $\theta^{\mathrm{Exc}}_1\, m_1 \le w_1^m \wedge \ldots$ and $\forall a_1, a_2, \theta^{\mathrm{W}^{\mathrm{Exc}}_1}\ (f_1\, a_1) \le w_1^f\ a_1 \wedge \ldots$, in particular the second hypothesis requires an element $a_2 : A_2$ that prevents[3] us from concluding by monotonicity of $\texttt{bind}^{\mathrm{W}^{\mathrm{Exc}}_1}$.

This problematic hypothesis only depends on the part of the context relevant for the left program and not on the full context, so we introduce structured contexts $\Gamma = (\Gamma_1, \Gamma_2)$ in our judgments, where Γ_1 and Γ_2 are simple contexts. The judgment $\Gamma \vdash c_1\ \{w_1\} \sim c_2\ \{w_2\} \mid w_{\mathrm{rel}}$ now presupposes that $\Gamma_i \vdash c_i : \mathrm{M}_i\, A_i$, $\Gamma_i \vdash w_i : \mathrm{W}_i\ (i = 1, 2)$ and that $\Gamma_1, \Gamma_2 \vdash w_{\mathrm{rel}} : \mathrm{W}_{\mathrm{rel}}(A_1, A_2)$. The semantics of this judgment is given by

$$\Gamma \vDash c_1\ \{w_1\} \sim c_2\ \{w_2\} \mid w_{\mathrm{rel}} \quad = \quad \left(\begin{array}{c} \forall \gamma_1 : \Gamma_1, \theta_1(c_1\, \gamma_1) \le w_1\, \gamma_1, \\ \forall \gamma_2 : \Gamma_2, \theta_2(c_2\, \gamma_2) \le w_2\, \gamma_2, \\ \forall (\gamma_1, \gamma_2) : \Gamma_1 \times \Gamma_2, \theta_{\mathrm{rel}}(c_1\, \gamma_1, c_2\, \gamma_2) \le w_{\mathrm{rel}}(\gamma_1, \gamma_2) \end{array}\right) \tag{18}$$

A conceptual understanding of this interpretation that will be useful in the following is to consider Γ as a (trivial) relation $\Gamma^{\boldsymbol{r}} = (\Gamma_1, \Gamma_2, \lambda(\gamma_1 : \Gamma_1)(\gamma_2 : \Gamma_2).\ \mathbb{1})$ instead of a pair and define the family of relations $\Theta^{\boldsymbol{r}}(\boldsymbol{\gamma}) = (\Theta_1(\gamma_1), \Theta_2(\gamma_2), \Theta_{\mathrm{rel}}\boldsymbol{\gamma})$ dependent over $\Gamma^{\boldsymbol{r}}$:

$$\Theta_1(\gamma_1 : \Gamma_1) = \theta_1(c_1\, \gamma_1) \le w_1\, \gamma_1, \qquad \Theta_2(\gamma_2 : \Gamma_2) = \theta_2(c_2\, \gamma_2) \le w_2\, \gamma_2,$$

$$\Theta_{\mathrm{rel}}(\boldsymbol{\gamma} : \Gamma)(w_1 : \Theta_1\, \gamma_1, w_2 : \Theta_2\, \gamma_2) = \theta_{\mathrm{rel}}(c_1\, \gamma_1, c_2\, \gamma_2) \le w_{\mathrm{rel}}\boldsymbol{\gamma}.$$

[3] Instead of insisting that $\vdash c_1\ \{w_1\} \sim c_2\ \{w_2\} \mid w_{\mathrm{rel}}$ proves the correctness of c_1 and c_2 with respect to w_1 and w_2 we could try to presuppose it, however this idea does not fare well since it would require a property akin of cancellability with respect to bind $\theta^{\mathrm{Exc}}_1\, (\texttt{bind}^{\mathrm{Exc}}_1\, m_1\, f_1) \le \texttt{bind}^{\mathrm{W}^{\mathrm{Exc}}_1}\ w_1^m\ w_1^f \Rightarrow \theta^{\mathrm{Exc}}_1\, m_1 \le w_1^m$ that has no reason to hold in our examples.

$$A^{\boldsymbol{r}}, B^{\boldsymbol{r}}, \Gamma^{\boldsymbol{r}} ::= \mathbb{0}^{\boldsymbol{r}} \mid \mathbb{1}^{\boldsymbol{r}} \mid \mathbb{B}^{\boldsymbol{r}} \mid \mathbb{N}^{\boldsymbol{r}} \mid A^{\boldsymbol{r}} + B^{\boldsymbol{r}} \mid (\boldsymbol{a} : A^{\boldsymbol{r}}) \times B^{\boldsymbol{r}}\ \boldsymbol{a} \mid (\boldsymbol{a} : A^{\boldsymbol{r}}) \to B^{\boldsymbol{r}}\ \boldsymbol{a}$$

$[\![-]\!]$ maps a relational type $A^{\boldsymbol{r}}$ to its underlying representation $[\![A^{\boldsymbol{r}}]\!] = (A_0, A_1, A_r)$

$$[\![\mathbb{0}^{\boldsymbol{r}}]\!] = (\mathbb{0}, \mathbb{0}, =) \qquad [\![\mathbb{1}^{\boldsymbol{r}}]\!] = (\mathbb{1}, \mathbb{1}, =) \qquad [\![\mathbb{B}^{\boldsymbol{r}}]\!] = (\mathbb{B}, \mathbb{B}, =) \qquad [\![\mathbb{N}^{\boldsymbol{r}}]\!] = (\mathbb{N}, \mathbb{N}, =)$$

$$[\![A^{\boldsymbol{r}} + B^{\boldsymbol{r}}]\!] = \left(\begin{array}{l} ab_1 : A_1 + B_1 \\ ab_2 : A_2 + B_2 \\ \mathtt{case}\ \ (ab_1, ab_2)\ [(\mathtt{Inl}\ a_1, \mathtt{Inl}\ a_2).A_{\text{rel}}\ a_1\ a_2 \mid (\mathtt{Inr}\ b_1, \mathtt{Inr}\ b_2).B_{\text{rel}}\ b_1\ b_2 \mid (\_, \_)\ .\ \mathbb{0}] \end{array} \right)$$

$$[\![(\boldsymbol{a} : A^{\boldsymbol{r}}) \times B^{\boldsymbol{r}}\ \boldsymbol{a}]\!] = \left(\begin{array}{l} (a_1, b_1) : (a_1 : A_1) \times B_1\ a_1, \\ (a_2, b_2) : (a_2 : A_2) \times B_2\ a_2, \\ (a_r : A_r\ a_1\ a_2) \times B_r\ a_1\ a_2\ a_r\ b_1\ b_2 \end{array} \right)$$

$$[\![(\boldsymbol{a} : A^{\boldsymbol{r}}) \to B^{\boldsymbol{r}}\ \boldsymbol{a}]\!] = \left(\begin{array}{l} f_1 : (a_1 : A_1) \to B_1\ a_1, \\ f_2 : (a_2 : A_2) \to B_2\ a_2, \\ (a_1 : A_1)(a_2 : A_2)(a_r : A_r\ a_1\ a_2) \to B_r\ a_1\ a_2\ a_r\ (f_1\ a_1)\ (f_2\ a_2) \end{array} \right)$$

Fig. 3. Syntax of RDTT and translation to base type theory

Then the relational judgment $\Gamma \vDash c_1\ \{w_1\} \sim c_2\ \{w_2\} \mid w_{\text{rel}}$ can be interpreted as a dependent function $(\boldsymbol{\gamma} : \Gamma^{\boldsymbol{r}}) \to \Theta^{\boldsymbol{r}}\ \boldsymbol{\gamma}$ in an appropriate relational dependent type theory.

3.3 A Relational Dependent Type Theory

Adding unary specifications in the relational judgment enables a full treatment of exceptions, however the pure rules of section §2.4 do not deal with a structured context $\Gamma^{\boldsymbol{r}} = (\Gamma_1, \Gamma_2, \Gamma_{\text{rel}})$. In order to recover rules dealing with such a context, we apply the same recipe internally to a relational dependent type theory as described by Tonelli [2013]. In practice, this type theory could be described as a syntactic model in the sense of Boulier et al. [2017], that is a translation from a source type theory to a target type theory that we take to be our ambient type theory, where a type in the source theory is translated to a pair of types and a relation between them. We call the resulting source type theory RDTT and describe part of its construction in Figure 3. A systematic construction of RDTT at the semantic level is obtained by considering the category with families $\mathcal{S}\text{pan}(\mathcal{T}ype)$ consisting of families of types and functions indexed by the span $(1 \leftarrow \text{rel} \rightarrow 2)$, a special case of Kapulkin and Lumsdaine [2018]; Shulman [2014].

Moving from our ambient type theory to RDTT informs us on how to define rules coming from the type theory. For instance, generalizing the rule for if-then-else, we can use the motive $P(\boldsymbol{ab} : A^{\boldsymbol{r}} + B^{\boldsymbol{r}}) = \Theta^{\boldsymbol{r}}(\boldsymbol{ab}) : \text{Type}^{\boldsymbol{r}}$ on the dependent eliminator for sum type

$$\mathtt{elim\_sum} : (P : (A^{\boldsymbol{r}} + B^{\boldsymbol{r}}) \to \text{Type}^{\boldsymbol{r}}) \to (a : A^{\boldsymbol{r}} \to P\ a) \to (b : B^{\boldsymbol{r}} \to P\ b) \to (x : A^{\boldsymbol{r}} + B^{\boldsymbol{r}}) \to P\ x$$

to obtain a rule for case splitting. This eliminator translates to the large term in Figure 4 that induces the following relational rule using $\boldsymbol{w^l} = (w_1^l, w_2^l, w_{\text{rel}}^l)$, $\boldsymbol{w^r} = (w_1^r, w_2^r, w_{\text{rel}}^r)$ and the relational specifications of the conclusion – where we abbreviate pattern matching with a case construction

$$
\begin{aligned}
[\![\texttt{elim\_sum}]\!] : {} & (P_1 : A_1 + B_1 \to \mathrm{Type}) \to (P_2 : A_2 + B_2 \to \mathrm{Type}) \to \\
& (P_{\mathrm{rel}} : \forall (ab_1 : A_1 + B_1)(ab_2 : A_2 + B_2), (A^{\boldsymbol{r}} + B^{\boldsymbol{r}})_{\mathrm{rel}}\ ab_1\ ab_2 \to \mathrm{Type}) \to \\
& (\forall (a_1 : A_1), P_1\ (\texttt{Inl}\ a_1)) \to (\forall (a_2 : A_2), P_2\ (\texttt{Inl}\ a_2)) \to \\
& (\forall a_1\ a_2\ (a_{\mathrm{rel}} : A^{\boldsymbol{r}}\ a_1\ a_2), P_{\mathrm{rel}}\ (\texttt{Inl}\ a_1)\ (\texttt{Inl}\ a_2)\ a_{\mathrm{rel}}) \\
& (\forall (b_1 : B_1), P_1\ (\texttt{Inr}\ b_1)) \to (\forall (b_2 : B_2), P_2\ (\texttt{Inr}\ b_2)) \to \\
& (\forall b_1\ b_2\ (b_{\mathrm{rel}} : B^{\boldsymbol{r}}\ b_1\ b_2), P_{\mathrm{rel}}\ (\texttt{Inr}\ b_1)\ (\texttt{Inr}\ b_2)\ b_{\mathrm{rel}}) \to \\
& \forall ab_1\ ab_2\ (ab_{\mathrm{rel}} : (A^{\boldsymbol{r}} + B^{\boldsymbol{r}})_{\mathrm{rel}}\ ab_1\ ab_2), P_{\mathrm{rel}}\ ab_1\ ab_2\ ab_{\mathrm{rel}}
\end{aligned}
$$

Fig. 4. Relational translation of the eliminator for sum types

– as arguments to the eliminator

$$
\begin{array}{l}
\Gamma, \boldsymbol{a} : A^{\boldsymbol{r}} \vdash c_1[\texttt{Inl}\ a_1/ab_1]\ \{w_1^l\} \sim c_2[\texttt{Inl}\ a_2/ab_2]\ \{w_2^l\} \mid w_{\mathrm{rel}}^l[a_{\mathrm{rel}}/ab_{\mathrm{rel}}] \\
\Gamma, \boldsymbol{b} : B^{\boldsymbol{r}} \vdash c_1[\texttt{Inr}\ b_1/ab_1]\ \{w_1^r\} \sim c_2[\texttt{Inr}\ b_2/ab_2]\ \{w_2^r\} \mid w_{\mathrm{rel}}^r[b_{\mathrm{rel}}/ab_{\mathrm{rel}}]
\end{array}
$$

$$
\Gamma^{\boldsymbol{r}}, \boldsymbol{ab} : A^{\boldsymbol{r}} + B^{\boldsymbol{r}} \vdash \begin{array}{ll} c_1 & \{\texttt{case}\ \ ab_1\ [\texttt{Inl}\ a_1.w_1^l \mid \texttt{Inr}\ b_1.w_1^r]\} \\ \sim & \\ c_2 & \{\texttt{case}\ \ ab_2\ [\texttt{Inl}\ a_2.w_2^l \mid \texttt{Inr}\ b_2.w_2^r]\} \end{array} \Bigg| \texttt{case}\, ab_1, ab_2 \left[\begin{array}{l} \texttt{Inl}\ a_1, \texttt{Inl}\ a_2.w_{\mathrm{rel}}^l \\ \texttt{Inr}\ b_1, \texttt{Inr}\ b_2.w_{\mathrm{rel}}^r \end{array} \right]
$$

As in the simple setting, we can then refine this rule to obtain synchronous or asynchronous rules specifying a required shape for the programs c_1, c_2.

3.4 Relative Monads and Monad Morphisms

Before giving the general framework able to derive monadic rules dealing with exceptions, we return to the notions of relative monads and relative monad morphisms, since these will be the common underlying concept relating the simple and generic frameworks.

Definition 3 (relative monads [Altenkirch et al. 2015]). *Let $\mathcal{I}, \mathcal{C}$ be categories and $\mathcal{J} : \mathcal{I} \to \mathcal{C}$ a functor between these. A $\mathcal{J}$-relative monad is given by*

- *for each $A \in \mathcal{I}$, an object $\mathcal{T}\ A \in \mathcal{C}$*
- *for each $A \in \mathcal{I}$, a morphism $\texttt{ret}_A^{\mathcal{T}} \in \mathcal{C}(\mathcal{J}\ A; \mathcal{T}\ A)$*
- *for each $A, B \in \mathcal{I}$, a function $(-)^{\dagger_{\mathcal{T}}} : \mathcal{C}(\mathcal{J}\ A; \mathcal{T}\ B) \to \mathcal{C}(\mathcal{T}\ A; \mathcal{T}\ B)$*
- *satisfying the 3 following equations*

$$
f^{\dagger_{\mathcal{T}}} \circ \texttt{ret}_A^{\mathcal{T}} = f \qquad (\texttt{ret}_A^{\mathcal{T}})^{\dagger_{\mathcal{T}}} = \mathrm{id}_{\mathcal{T}\ A} \qquad g^{\dagger_{\mathcal{T}}} \circ f^{\dagger_{\mathcal{T}}} = (g^{\dagger_{\mathcal{T}}} \circ f)^{\dagger_{\mathcal{T}}}
$$

Noting *Type* for the category of types and functions of our ambient type theory, *Ord* for the category of preordered sets and monotonic functions, and $\mathrm{Disc} : \mathit{Type} \to \mathit{Ord}$ the functor equipping a type with its discrete preorder structure , a simple relational specification monad could be described as a relative monad $\mathrm{W}_{\mathrm{rel}} : \mathit{Type}^2 \to \mathit{Ord}$ over the functor $\mathrm{Disc} \circ \times : \mathit{Type}^2 \to \mathit{Ord}$ sending a pair of types (A_1, A_2) to their product $A_1 \times A_2$ equipped with a discrete preorder. The monotonicity condition imposed on $\texttt{bind}^{\mathrm{W}_{\mathrm{rel}}}$ amounts to require that all the structure is enriched in Ord[4] [Kelly 1982].

Simple relational effect observations from $\mathrm{M}_1, \mathrm{M}_2$ to $\mathrm{W}_{\mathrm{rel}}$ can also be interpreted as instances of relative monad morphisms. First, a pair of computational monads $\mathrm{M}_1, \mathrm{M}_2$ yields a monad $\mathrm{M}_1 \otimes \mathrm{M}_2 : \mathit{Type}^2 \to \mathit{Type}^2$ acting on pairs of types, that is $\mathrm{M}_1 \otimes \mathrm{M}_2\ (A_1, A_2) = (\mathrm{M}_1\ A_1, \mathrm{M}_2\ A_2)$ with monadic structures provided by each sides. Second, by proposition 2.3 of Altenkirch et al. [2015], the monad

[4] Type^2 can be enriched over *Ord* by change-of-enrichment through the monoidal functor Disc.

$\mathrm{M}_1 \otimes \mathrm{M}_2$ is a relative monad on the identity functor $\mathrm{Id}_{\mathcal{T}ype^2}$. A simple relational effect observation is a relative monad morphism from $\mathrm{M}_1 \otimes \mathrm{M}_2$ to $\mathrm{W}_{\mathrm{rel}}$ over the functor $\mathrm{Disc} \circ \dot{\times}$.

Definition 4 (Lax relative monad morphism). *Let $\mathcal{I}, C_1, C_2$ be categories enriched over $\mathcal{O}rd$ and $\mathcal{J}_1 : \mathcal{I} \to C_1, \mathcal{J}_2 : \mathcal{I} \to C_2, \mathcal{F} : C_1 \to C_2$ be $\mathcal{O}rd$-enriched functors such that $\varphi : \mathcal{F} \circ \mathcal{J}_1 \cong \mathcal{J}_2$. A* lax relative monad morphism *from a $\mathcal{J}_1$-relative monad $\mathcal{T}_1 : \mathcal{I} \to C_1$ to a $\mathcal{J}_2$-relative monad $\mathcal{T}_2 : \mathcal{I} \to C_2$ is*

- *a family of morphisms $\theta_A : \mathcal{F} \circ \mathcal{T}_1 A \to \mathcal{T}_2 A$ indexed by objects $A \in \mathcal{I}$,*
- *such that*

$$\theta \circ \mathcal{F}\,\mathtt{ret}^{\mathcal{T}_1} \le \mathtt{ret}^{\mathcal{T}_2} \circ \varphi \qquad\qquad \theta \circ \mathcal{F}(f^{\dagger_{\mathcal{T}_1}}) \le (\theta \circ \mathcal{F} f \circ \varphi^{-1})^{\dagger_{\mathcal{T}_2}} \circ \theta \tag{19}$$

We say that θ is a relative monad morphism *when the last two conditions are equalities.*

Crucially, this definition of relative monad morphism generalizes the notion defined by Altenkirch et al. [2015] by enabling different base functor, a relative monad analog to the monad opfunctors of Street [1972].[5] Up to the enrichment, we recover the definition of Altenkirch et al. [2015] by taking $\mathcal{J}_1 = \mathcal{J}_2$, $\mathcal{F} = \mathrm{Id}$ and $\varphi = \mathrm{id}$.

3.5 Relational Specification Monads, Relational Effect Observations

Motivated by the case of exceptions, we now define the general notion of a relational specification monad. This definition is obtained by instantiating the definitions of an (enriched) relative monad to our relational dependent type theory, ensuring that we obtain a theory uniform with the simple setting, and crucially that we can use a similar methodology to introduce relational rules. We note $\mathcal{S}\mathrm{pan}(\mathcal{O}rd)$ for the category of relations between ordered types, $\mathcal{J}_\times : \mathrm{Type} \times \mathrm{Type} \to \mathcal{S}\mathrm{pan}(\mathcal{O}rd)$ for the functor defined on objects by $\mathcal{J}_\times(A_1, A_2) = \mathrm{Disc}\, A_1 \xleftarrow{\pi_1} \mathrm{Disc}\, A_1 \times \mathrm{Disc}\, A_2 \xrightarrow{\pi_2} \mathrm{Disc}\, A_2$ and $\pi_{1,2} : \mathcal{S}\mathrm{pan}(\mathcal{O}rd) \to \mathrm{Type} \times \mathrm{Type}$ sending a relation $A_1 \leftarrow A_{\mathrm{rel}} \rightarrow A_2$ to its legs (A_1, A_2).

Definition 5. *A* relational specification monad *consist of a pair of unary specification monads $\mathrm{W}_1, \mathrm{W}_2 : \mathrm{Type} \to \mathcal{O}rd$ and a relative monad $\mathbf{W} : \mathrm{Type} \times \mathrm{Type} \to \mathcal{S}\mathrm{pan}(\mathcal{O}rd)$ over $\mathcal{J}_\times$ lifting $\mathrm{W}_1, \mathrm{W}_2$, that is such that $\pi_{1,2} \circ \mathbf{W} = \mathrm{W}_1 \times \mathrm{W}_2$, and whenever $\pi_{1,2}(f) = (f_1, f_2)$*

$$\pi_{1,2}(\mathtt{ret}^{\mathbf{W}}) = (\mathtt{ret}^{\mathrm{W}_1}, \mathtt{ret}^{\mathrm{W}_2}), \qquad\qquad \pi_{1,2}(\mathtt{bind}^{\mathbf{W}}\, f) = (\mathtt{bind}^{\mathrm{W}_1}\, f_1, \mathtt{bind}^{\mathrm{W}_2}\, f_2).$$

In components, a relational specification monad over unary specification monads $\mathrm{W}_1, \mathrm{W}_2$ consists of a relation $\mathrm{W}_{\mathrm{rel}}(A_1, A_2) : \mathrm{W}_1 A_1 \to \mathrm{W}_2 A_2 \to \mathrm{Type}$ equipped with a preorder $\le^{\mathrm{W}}$, and operations

$$\begin{aligned}
\mathtt{ret}^{\mathrm{W}_{\mathrm{rel}}} &: (a_1, a_2) : A_1 \times A_2 \to \mathrm{W}_{\mathrm{rel}}(A_1, A_2)\,(\mathtt{ret}^{\mathrm{W}_1}\, a_1)\,(\mathtt{ret}^{\mathrm{W}_2}\, a_2)\\
\mathtt{bind}^{\mathrm{W}_{\mathrm{rel}}} &: \quad w_1^m : \mathrm{W}_1 A_1 \to w_2^m : \mathrm{W}_2 A_2 \to w_{\mathrm{rel}}^m : \mathrm{W}_{\mathrm{rel}}(A_1, A_2)\, w_1^m\, w_2^m \to\\
&\qquad w_1^f : (A_1 \to \mathrm{W}_1 B_1) \to w_2^f : (A_2 \to \mathrm{W}_2 B_1) \to\\
&\qquad w_{\mathrm{rel}}^f : (((a_1, a_2) : A_1 \times A_2) \to \mathrm{W}_{\mathrm{rel}}(B_1, B_2)\,(w_1^f\, a_1)\,(w_2^f\, a_2)) \to\\
&\qquad \mathrm{W}_{\mathrm{rel}}(B_1, B_2)\,(\mathtt{bind}^{\mathrm{W}_1} w_1^m\, w_1^f)\,(\mathtt{bind}^{\mathrm{W}_2} w_2^m\, w_2^f)
\end{aligned}$$

satisfying equations analogous to the monadic laws.

If these operations look complex, in most of our examples the relation $\mathrm{W}_{\mathrm{rel}}(A_1, A_2)\, w_1\, w_2$ is independent of w_1 and w_2. This happens for our leading example of exceptions, but also for any relational specification monad constructed out of a simple relational specification monad. Indeed, we can associate to any simple relational specification monad $\mathrm{W}_{\mathrm{rel}}$ the relational specification monad $\mathrm{W}(A_1, A_2) = (\mathrm{W}_{\mathrm{rel}}(A_1, \mathbb{1}), \mathrm{W}_{\mathrm{rel}}(\mathbb{1}, A_2), \lambda w_1\, w_2.\, \mathrm{W}_{\mathrm{rel}}(A_1, A_2))$. The monadic operations just discard the superfluous arguments.

[5] However, in contrast to the situation of monads, these relative monad morphisms do not dualize well.

$$\textsc{Weaken}\ \frac{\Gamma^{\boldsymbol{r}} \vdash c_1\ \{w_1\} \sim c_2\ \{w_2\} \mid w_{\text{rel}} \qquad w_1 \leq^{\text{W}_1} w_1' \qquad w_2 \leq^{\text{W}_2} w_2' \qquad w_{\text{rel}} \leq^{\text{W}_{\text{rel}}} w_{\text{rel}}'}{\Gamma^{\boldsymbol{r}} \vdash c_1\ \{w_1'\} \sim c_2\ \{w_2'\} \mid w_{\text{rel}}'}$$

$$\textsc{Ret}\ \frac{\Gamma_1 \vdash a_1 : A_1 \qquad \Gamma_2 \vdash a_2 : A_2}{\Gamma^{\boldsymbol{r}} \vdash \texttt{ret}^{\text{M}_1} a_1\ \{\texttt{ret}^{\text{W}_1} a_1\} \sim \texttt{ret}^{\text{M}_2} a_2\ \{\texttt{ret}^{\text{W}_2} a_2\} \mid \texttt{ret}^{\text{W}_{\text{rel}}}(a_1, a_2)}$$

$$\textsc{Bind}\ \frac{\Gamma^{\boldsymbol{r}} \vdash m_1\ \{w_1^m\} \sim m_2\ \{w_2^m\} \mid w^m \qquad \Gamma^{\boldsymbol{r}}, \boldsymbol{a} : A^{\boldsymbol{r}} \vdash f_1\ a_1\ \{w_1^f\ a_1\} \sim f_2\ a_2\ \{w_2^m\ a_2\} \mid w^f\ \boldsymbol{a}}{\Gamma^{\boldsymbol{r}} \vdash \begin{array}{lll|l} \texttt{bind}^{\text{M}_1}\ m_1\ f_1 & \{\texttt{bind}^{\text{W}_1}\ w_1^m\ w_1^f\} & & \texttt{bind}^{\text{W}_{\text{rel}}}\ \boldsymbol{w^m}\ \boldsymbol{w^f} \\ \sim & & & \\ \texttt{bind}^{\text{M}_2}\ m_2\ f_2 & \{\texttt{bind}^{\text{W}_2}\ w_2^m\ w_2^f\} & & \end{array}}$$

Fig. 5. Generic monadic rules in the full relational setting

We now turn to the definition of a relational effect observation.

Definition 6 (Lax relational effect observation). *A* lax relational effect observation *from* M_1, M_2 *to the relational specification monad* $\mathbf{W}$ *over* W_1, W_2 *is a lax relative monad morphism* $\boldsymbol{\theta}$ *from* $M_1 \otimes M_2$ *to* $\mathbf{W}$ *over* $\mathcal{J}_\times$*. A relational effect observation is* strict *when the inequalities (19) hold as equalities.*

Explicitly, such a lax relational effect observation $\boldsymbol{\theta}$ of three components $\theta_1, \theta_2, \theta_{\text{rel}}$ where $\theta_1 : \text{M}_1 \to \text{W}_1$, $\theta_2 : \text{M}_2 \to \text{W}_2$ are (plain) monad morphisms, and

$$\theta_{\text{rel}} \quad : \quad ((m_1, m_2) : \text{M}_1 A_1 \times \text{M}_2 A_2) \to \text{W}_{\text{rel}}(A_1, A_2)\,(\theta_1\, m_1)\,(\theta_2\, m_2)$$

verify the two inequations with respect to the monadic operations

$$\theta_{\text{rel}}(\texttt{ret}^{\text{M}_1}\, a_1, \texttt{ret}^{\text{M}_2}\, a_2) \leq \texttt{ret}^{\text{W}_{\text{rel}}}\,(a_1, a_2) : \text{W}_{\text{rel}}(A_1, A_2)\,(\theta_1\,(\texttt{ret}^{\text{M}_1}\, a_1))\,(\theta_2\,(\texttt{ret}^{\text{M}_2}\, a_2))$$

$$\theta_{\text{rel}}(\texttt{bind}^{\text{M}_1} m_1\, f_1, \texttt{bind}^{\text{M}_2} m_2\, f_2) \leq \texttt{bind}^{\text{W}_{\text{rel}}}\,(\theta_1\, m_1)\,(\theta_2\, m_2)\,(\theta_{\text{rel}}\, m_{\text{rel}})\ \theta_1 \circ f_1\ \ \theta_2 \circ f_2\ \ \theta_{\text{rel}} \circ (f_1 \times f_2)$$

Given a relational effect observation $\boldsymbol{\theta} : \text{M}_1 \otimes \text{M}_2 \to \mathbf{W}$, we can define in full generality the semantics of the relational judgment by the Equation 18. We introduce the generic monadic rules in Figure 5, and similarly to the simple setting obtain the following soundness theorem.

Theorem 4 (Soundness of monadic rules). *The relational rules in Figure 5 are sound with respect to any lax relational effect observation* $\boldsymbol{\theta}$*, that is*

$$\Gamma^{\boldsymbol{r}} \vdash c_1\ \{w_1\} \sim c_2\ \{w_2\} \mid w_{rel} \quad \Rightarrow \quad \forall \boldsymbol{\theta},\ \ \Gamma^{\boldsymbol{r}} \vDash_{\boldsymbol{\theta}} c_1\ \{w_1\} \sim c_2\ \{w_2\} \mid w_{rel}$$

3.6 Relational Specification Monad Transformers

Having a category of relational specification monads and relative monad morphisms between them, we define a *relational specification monad transformer* to be a pointed endofunctor on this category [Lüth and Ghani 2002]. Under mild assumptions, the usual state and exception transformer lifts to this setting, yielding in each case both a left-variant and a right-variant applying either to the left type A_1 or right one A_2 of a relational specification monad $\text{W}(A_1, A_2)$. Since the two variants are symmetric, we only detail the left ones.

Adding state. The usual state monad transformer maps a monad M to the monad $\text{StT}(\text{M})\, A = S \to \text{M}\,(A \times S)$. The left relational state monad transformer StT_{rel} maps a relational specification monad $\mathbf{W}\,(A_1, A_2) = (\text{W}_1\, A_1, \text{W}_2\, A_2, \lambda w_1\, w_2.\ \text{W}_{\text{rel}}(A_1, A_2)\, w_1\, w_2)$ to the relational specification monad with carrier

$$\text{StT}_{\text{rel}}(\mathbf{W})(A_1, A_2) \quad = \quad (\text{StT}(\text{W}_1)\, A_1,\ \ \text{W}_2 A_2,\ \ \lambda w_1\, w_2.\ (s_1 : S_1) \to \text{W}_{\text{rel}}(A_1 \times S_1, A_2)\,(w_1\, s_1)\, w_2)$$

The monadic operations on $\mathrm{StT}_{\mathrm{rel}}(\mathrm{W})_1$ are given by the usual state transformer. The added data resides in the `ret` and `bind` operations responsible for the relational part:

$$\texttt{let}\ \texttt{ret}^{\mathrm{StT(W)}}_{\mathrm{rel}}\ (a_1,a_2) : (s_1 : S_1) \to \mathrm{W}_{\mathrm{rel}}\ (A_1 \times S_1, A_2)\ (\texttt{ret}^{StT(W)_1}\ (a_1,s_1))\ (\texttt{ret}^{\mathrm{W}_2}\ a_2) = \lambda s_1.\ \texttt{ret}^{\mathrm{W}_{\mathrm{rel}}}\ ((a_1,s_1), a_2)$$

$$\begin{aligned}\texttt{let}\ \texttt{bind}^{\mathrm{StT(W)}}_{\mathrm{rel}}\ &(m_1 : \mathrm{StT(W)}_1\ A_1)\ (m_2 : \mathrm{W}_2\ A_2)\ (m_{\mathrm{rel}} : \mathrm{StT(W)}_{\mathrm{rel}}\ (A_1,A_2)\ m_1\ m_2)\\ &(f_1 : A_1 \to \mathrm{StT(W)}_1\ B_1)\ (f_2 : A_2 \to \mathrm{W}_2\ B_2)\\ &(f_{\mathrm{rel}} : (a_1,a_2){:}A_1 \times A_2 \to \mathrm{StT(W)}_{\mathrm{rel}}\ (B_1, B_2)\ (f_1\ a_1)\ (f_2\ a_2))\\ &: \mathrm{StT(W)}_{\mathrm{rel}}\ (B_1,B_2)\ (\texttt{bind}^{\mathrm{StT(W)}_1}\ m_1\ f_1)\ (\texttt{bind}^{\mathrm{W}_2}\ m_2\ f_2) =\\ \lambda s_1.\ \texttt{bind}^{\mathrm{W}_{\mathrm{rel}}}\ &(m_1\ s_1)\ m_2\ (m_{\mathrm{rel}}\ s_1)\ (\lambda\ (a_1,s_1').\ f_1\ a_1\ s_1')\ f_2\ (\lambda\ ((a_1,s_1'), a_2).\ f_{\mathrm{rel}}\ (a_1,a_2)\ s_1')\end{aligned}$$

Adding exceptions. In a similar flavor, the exception monad transformer ExcT mapping a monad M to $\mathrm{ExcT(M)}A = \mathrm{M}(A + E_1)$ gives raise to the carrier its relational specification monad counterpart $\mathrm{ExcT}_{\mathrm{rel}}(\mathbf{W})(A_1, A_2) = (\mathrm{ExcT}(\mathrm{W}_1)A_1, \mathrm{W}_2A_2, \mathrm{W}_{\mathrm{rel}}(A_1 + E_1, A_2))$.

However, in order to define the bind operation we need to restrict our attention to relational specification monad for which unary specification can be lifted to relational ones. This is provided by the structure of two maps:

$$\tau_1 : w_1 : \mathrm{W}_1(A_1, \mathbb{1}) \to \mathrm{W}_{\mathrm{rel}}(A_1, \mathbb{1})\ w_1\ (\texttt{ret}^{\mathrm{W}_2}\ ()),$$

$$\tau_2 : w_2 : \mathrm{W}_2(\mathbb{1}, A_2) \to \mathrm{W}_{\mathrm{rel}}(\mathbb{1}, A_2)\ (\texttt{ret}^{\mathrm{W}_1}\ ())\ w_2.$$

such that pairing each of them with identity provides a monad morphism, that is $(\mathrm{id}, \tau_1) : \mathrm{W}_1(A, \mathbb{1}) \to (w : \mathrm{W}_1(A, \mathbb{1})) \times \mathrm{W}_{\mathrm{rel}}(A, \mathbb{1})\ w\ (\texttt{ret}^{\mathrm{W}_2}\ ())$ respects the monadic equations. Any relational specification monad induced by a simple one has a canonical such structure, taking τ_1 and τ_2 to be identity. The state transformer and the exception transformer also preserve this structure.

Assuming that $\mathbf{W}$ is equipped with τ_1, τ_2, we define the return and bind operation on $\mathrm{ExcT}(\mathrm{W}_{\mathrm{rel}})$ as:

$$\texttt{let}\ \texttt{ret}^{\mathrm{ExcT(W)}_{\mathrm{rel}}}\ (a_1,a_2) : \mathrm{W}_{\mathrm{rel}}\ (A_1 + E_1, A_2)\ (\texttt{ret}^{\mathrm{ExcT(W)}_1}\ a_1)\ (\texttt{ret}^{\mathrm{W}_2}\ a_2) = \texttt{ret}^{\mathrm{W}_{\mathrm{rel}}}\ (\mathrm{Inl}\ a_1, a_2)$$

$$\begin{aligned}\texttt{let}\ \texttt{bind}^{\mathrm{ExcT(W)}_{\mathrm{rel}}}\ &(m_1 : \mathrm{ExcT(W)}_1\ A_1)\ (m_2 : \mathrm{W}_2\ A_2)\ (m_{\mathrm{rel}} : \mathrm{ExcT(W)}_{\mathrm{rel}}\ (A_1,A_2)\ m_1\ m_2)\\ &(f_1 : A_1 \to \mathrm{ExcT(W)}_1\ B_1)\ (f_2 : A_2 \to \mathrm{W}_2\ B_2)\\ &(f_{\mathrm{rel}} : (a_1,a_2){:}A_1 \times A_2 \to \mathrm{ExcT(W)}_{\mathrm{rel}}\ (B_1, B_2)\ (f_1\ a_1)\ (f_2\ a_2))\\ &: \mathrm{ExcT(W)}_{\mathrm{rel}}\ (B_1,B_2)\ (\texttt{bind}^{\mathrm{ExcT(W)}_1}\ m_1\ f_1)\ (\texttt{bind}^{\mathrm{W}_2}\ m_2\ f_2) =\\ \texttt{bind}^{\mathrm{W}_{\mathrm{rel}}}\ &m_1\ m_2\ m_{\mathrm{rel}}\ (\lambda\ ae_1.\ \texttt{match}\ ae_1\ \texttt{with} \mid \mathrm{Inl}\ a_1 \to f_1\ a_1 \mid \mathrm{Inr}\ e_1 \to \texttt{ret}^{\mathrm{W}_1}\ (\mathrm{Inr}\ e_1))\ f_2\\ &(\lambda\ ae_1\ a_2\ .\ \texttt{match}\ ae_1\ \texttt{with}\\ &\qquad \mid \mathrm{Inl}\ a_1 \to f_{\mathrm{rel}}\ a_1\ a_2\\ &\qquad \mid \mathrm{Inr}\ e_1 \to \texttt{bind}^{\mathrm{W}_{\mathrm{rel}}}\ (\tau_2\ (f_2\ a_2))\ (\lambda\ ((), b_2)\ .\ \texttt{ret}^{\mathrm{W}_{\mathrm{rel}}}\ (\mathrm{Inr}\ e_1, b_2)))\end{aligned}$$

Putting these monad transformer to practice, we can finally define the full relational specification monad for exceptions validating the rules in Figure 6 by first lifting the simple relational $\mathrm{W}^{\mathrm{Pure}}_{\mathrm{rel}}$ and applying the exception transformers on both left and right sides. Further, applications would involve specifications relating state and exceptions with rollback state. The structure provided by τ_1, τ_2 is a technical requirement, and we leave to further investigation the conceptual understanding of this structure in the setting of relational specification monads.

$$\frac{}{\Gamma^{r} \vdash \mathtt{throw}\, e_1\ \{\lambda\varphi_1.\ \varphi_1\,(\mathtt{Inr}\, e_1)\} \sim \mathtt{ret}^{\mathrm{Exc}}\, a_2\ \{\mathtt{ret}^{\mathrm{W}_2^{\mathrm{Exc}}}\, a_2\} \mid \lambda\varphi.\ \varphi\,(\mathtt{Inr}\, e_1, \mathtt{Inl}\, a_2)}$$

$$\frac{}{\Gamma^{r} \vdash \mathtt{ret}^{\mathrm{Exc}}\, a_1\ \{\mathtt{ret}^{\mathrm{W}_1^{\mathrm{Exc}}}\, a_1\} \sim \mathtt{throw}\, e_2\ \{\lambda\varphi_2.\ \varphi_2\,(\mathtt{Inr}\, e_2)\} \mid \lambda\varphi.\ \varphi\,(\mathtt{Inl}\, a_1, \mathtt{Inr}\, e_2)}$$

$$\frac{\Gamma^{r} \vdash c_1\ \{w_1\} \sim c_2\ \{w_2\} \mid w_{\mathrm{rel}} \qquad \Gamma^{r} \vdash c_1^{err}\ \{w_1^{err}\} \sim c_2^{err}\ \{w_2^{err}\} \mid w_{\mathrm{rel}}^{err}}{\Gamma^{r} \vdash \mathtt{catch}\, c_1\, c_1^{err}\ \{w^{\mathtt{catch}}\, w_1\, w_1^{err}\} \sim \mathtt{catch}\, c_2\, c_2^{err}\ \{w^{\mathtt{catch}}\, w_2\, w_2^{err}\} \mid w_{\mathrm{rel}}^{\mathtt{catch}}\, w_{\mathrm{rel}}\, \boldsymbol{w^{err}}}$$

$$\begin{aligned}
&\mathtt{let}\ w^{\mathtt{catch}}\ (w : \mathrm{W}^{\mathrm{Exc}} A)\ (werr : E \to \mathrm{W}^{\mathrm{Exc}} A) : \mathrm{W}\ A = \\
&\quad \lambda\varphi.\ w\,(\lambda\ ae.\ \mathtt{match}\ ae\ \mathtt{with} \mid \mathtt{Inl}\ a \to \mathtt{ret}^{\mathrm{W}^{\mathrm{Exc}}}\ a\ \varphi \mid \mathtt{Inr}\ e \to werr\ e\ \varphi)
\end{aligned}$$

$$\begin{aligned}
&\mathtt{let}\ w_{\mathrm{rel}}^{\mathtt{catch}}\ (w{:}\mathrm{W}_{\mathrm{rel}}^{\mathrm{Exc}}\ (A_1{,}A_2))\ (werr_1 : E_1 \to \mathrm{W}^{\mathrm{Exc}}{}_1 A_1)\ (werr_2 : E_2 \to \mathrm{W}^{\mathrm{Exc}}{}_2 A_2) \\
&\qquad\qquad (werr_{\mathrm{rel}} : E_1 \times E_2 \to \mathrm{W}_{\mathrm{rel}}^{\mathrm{Exc}}(A_1{,}A_2)) : \mathrm{W}_{\mathrm{rel}}^{\mathrm{Exc}}(A_1{,}A_2) = \\
&\quad \lambda\varphi.\ w\,(\lambda\ (ae_1, ae_2).\ \mathtt{match}\ ae_1, ae_2\ \mathtt{with} \\
&\qquad \mid \mathtt{Inl}\ a_1, \mathtt{Inl}\ a_2 \to \mathtt{ret}^{\mathrm{W}_{\mathrm{rel}}^{\mathrm{Exc}}}\ (a_1{,}a_2)\ \varphi \\
&\qquad \mid \mathtt{Inr}\ e_1, \mathtt{Inl}\ a_2 \to werr_1\ e_1\ (\lambda\ ae_1 \to \varphi\ (ae_1, \mathtt{Inl}\ a_2)) \\
&\qquad \mid \mathtt{Inl}\ a_1, \mathtt{Inr}\ e_2 \to werr_2\ e_2\ (\lambda\ ae_2 \to \varphi\ (\mathtt{Inl}\ a_1, ae_2)) \\
&\qquad \mid \mathtt{Inr}\ e_1, \mathtt{Inr}\ e_2 \to werr_{\mathrm{rel}}\ (e_1{,}e_2)\ \varphi)
\end{aligned}$$

Fig. 6. Rules for exceptions

4 EMBEDDING RELATIONAL PROGRAM LOGICS

4.1 Relational Hoare Logic

As explained in the introduction, Benton [2004]'s seminal relational Hoare logic (RHL) is at the origin of many works on relational program logics (see also §6). We present here a syntactic embedding of RHL, showing that our simple framework can host usual program logics.

Concretely, we define a translation from While-language to monadic programs using the Imp monad, and show that the translation of all Benton [2004]'s rules (with the exception of two partial equivalence specific ones) are admissible in our framework using the effect observation $\theta_{\mathrm{rel}}^{\mathrm{Part}}$.

The translation from direct-style imperative programs to monadic ones follows closely Moggi's [1989] interpretation of call-by-value in his monadic metalanguage. The Imp monad of §2.6 directly interprets read and write, and while loops are translated using the following definable combinator

$$\begin{aligned}
&\mathtt{let}\ \mathtt{while}\ (guard{:}\ \mathrm{Imp}\ \mathbb{B})\ (body : \mathrm{Imp}\ \mathbb{1}) : \mathrm{Imp}\ \mathbb{1} = \\
&\quad \mathtt{do_while}\ (\mathtt{bind}^{\mathrm{Imp}}\ guard\ (\lambda\ b\ .\ \mathtt{if}\ b\ \mathtt{then}\ \mathtt{bind}^{\mathrm{Imp}}\ body\ (\lambda\ ()\ .\ \mathtt{ret}^{\mathrm{Imp}}\ \mathtt{tt})\ \mathtt{else}\ \mathtt{ret}^{\mathrm{Imp}}\ \mathtt{ff}))
\end{aligned}$$

The proofs of admissibility for the various rules exhibit a recurrent pattern. We first use weakening to adapt the specification obtained through the translation to an appropriate shape for the rules of our logic. Then we use the pure and generic monadic rules to decompose the programs on both sides. Finally, effect-specific rules together with admissibility of the premises finish the proof.

An easy corollary of our proof is that Benton [2004]'s relational rules are valid for our partial correctness interpretation. However, our interpretation treats non-termination in a slightly different way from his semantics. Indeed, our partial correctness semantics relates two programs whenever one of them diverges, whereas his requires both program to have the same divergence behaviour. A main difference is that our semantics is more compositional and allows to compute a *precise* specification by applying $\theta_{\mathrm{rel}}^{\mathrm{Part}}$ to the parts and combining the results, while for Benton's semantics this will

not produce precise specifications. This makes additional rules sound with respect to our semantics, allowing for instance to derive that $\vdash \texttt{skip} \sim \texttt{loop} \left\{ \lambda\varphi(s_1^i, s_2^i).\ \forall a_1, a_2, s_1^f, s_2^f.\varphi((a_1, s_1^f), (a_2, s_2^f)) \right\}$ (equivalent to $\vdash \{ \top \}\ \texttt{skip} \sim \texttt{loop}\ \{ \top \}$ in pre-/postcondition form), although it is of course a choice whether one wants a semantics that validates such rules or not. Another difference is that Benton's semantics assumes a classical logic, in which one can "decide" termination, while our semantics easily works in a constructive logic. We leave as future work to investigate if Benton's semantics can be successfully expressed using a simple *lax* effect observation.

4.2 Relational Hoare Type Theory

Nanevski et al. [2013] introduce Relational Hoare Type Theory (RHTT) for the specific goal of proving noninterference properties of programs. RHTT builds upon powerful but specific semantic objects embedded in the type theory of Coq to support specifications relating two runs of a single program. We explain here how we can reconstruct their model with a relational specification monad and an effect observation. This connection between the two frameworks could help extending RHTT to other effects, for instance exceptions.

A model of state and partiality. The effects supported by RHTT are manipulation of a structured heap – a refined version of the simple state monad of §2.1 – and partiality. In order to model these effects, a close variant of the following monad is used

$$\mathrm{M}\,A = (p : \mathit{heap} \to \mathbb{P}) \times (f : (r :\leq p) \to A \to \mathit{heap} \to \mathbb{P}) \times \texttt{coherent}(f)$$

where $\leq p = \{r : \mathit{heap} \to \mathbb{P} \mid \forall h, r\,h \Rightarrow p\,h\}$ and the predicate `coherent` specifies that f is defined by its value on singleton predicates consisting of only one heap. Using predicates enables the definition of fixpoint operators, in the same fashion as we did in our interpretation of while loops for the Imp effect in §2.6.

The relational specification used by Nanevski et al. [2013] is a variation on the simple relational monad of stateful pre- and postconditions from §2.2 where the precondition only takes one input heap corresponding to the fact only one program is considered at a time.

$$\mathrm{PP}_{\mathrm{rel}}(A_1, A_2) = (\mathit{heap} \to \mathbb{P}) \times (\mathit{heap} \times \mathit{heap} \to A_1 \times A_2 \to \mathit{heap} \times \mathit{heap} \to \mathbb{P})$$

Taking the same computational monad M on both sides, that is $\mathrm{M}_1 = \mathrm{M}_2 = \mathrm{M}$, we define the following simple relational effect observation $\theta : M, M \to \mathrm{PP}_{\mathrm{rel}}$

$$\theta(c_1, c_2) = (\lambda h_0.\ \pi_1\,c_1\,h_0 \wedge \pi_1\,c_2\,h_0,\\ \lambda(h_1, h_2)(a_1, a_2)(h_1', h_2').\ \pi_1\,c_1\,h_1 \wedge \pi_2 c_1 h_1 a_1 h_1' \wedge \pi_1\,c_2\,h_2 \wedge \pi_2 c_2 h_2 a_2 h_2')$$

5 PRODUCT PROGRAMS

The product programs methodology is an approach to prove relational properties that can serve as an alternative to relational program logics [Barthe et al. 2011, 2016]. In this section we show how to understand this methodology from the point of view of our framework.

Product programs reduce the problem of verifying relational properties on two programs c_1 and c_2 to the problem of verifying properties on a single *product program* c capturing at the same time the behaviors of c_1 and c_2. To prove a relational property w on programs c_1 and c_2, the methodology tells us to proceed as follows. First, we construct a product program c of c_1 and c_2. Then, by standard methods, we prove that the program c satisfies the property w seen as a non-relational property. Finally, from a general argument of soundness, we can conclude that φ must hold on c_1 and c_2. In what follows, we show how these three steps would be understood in our framework if we wanted to prove $\vDash_\theta c_1 \sim c_2\ \{ w \}$.

First of all, we need a notion of product program. In the setting of monadic programs, we capture a product program of $c_1 : \mathrm{M}_1 A_1$ and $c_2 : \mathrm{M}_2 A_2$ as a program $c : \mathrm{P}(A_1, A_2)$, where P is a relative

monad over $(A_1, A_2) \mapsto A_1 \times A_2$ (see §3.4). We can think of $c : \mathrm{P}(A_1, A_2)$ as a single computation that is computing both a value of type A_1 and a value of type A_2 at the same time. We expect P to support the effects from both M_1 and M_2, mixing them in a controlled way. As a concrete example, we can define products of stateful programs – $\mathrm{M}_1 A_1 = \mathrm{St}_{S_1} A_1$ and $\mathrm{M}_2 A_2 = \mathrm{St}_{S_2} A_2$ – inhabiting the relative monad $\mathrm{P}^{\mathrm{St}}(A_1, A_2) = \mathrm{St}_{S_1 \times S_2}(A_1 \times A_2)$. To complete the definition of product programs, we also need to explain when a concrete product program $c : \mathrm{P}(A_1, A_2)$ is capturing the behavior of $c_1 : \mathrm{M}_1 A_1$ and $c_2 : \mathrm{M}_2 A_2$. We propose to capture this in a relation $c_1 \times c_2 \rightsquigarrow c$ that exhibits the connection between pairs of computations and their potential product programs. This relation should be closed under the monadic construction of the effects, that is

$$\frac{a_1 : A_1 \qquad a_2 : A_2}{\mathtt{ret}^{\mathrm{M}_1}\ a_1 \times \mathtt{ret}^{\mathrm{M}_2}\ a_2 \rightsquigarrow \mathtt{ret}^{\mathrm{P}}\ (a_1, a_2)} \qquad \frac{m_1 \times m_2 \rightsquigarrow m_{\mathrm{rel}} \qquad \forall a_1\ a_2, f_1\ a_1 \times f_2\ a_2 \rightsquigarrow f_{\mathrm{rel}}\ (a_1, a_2)}{\mathtt{bind}^{\mathrm{M}_1}\ m_1\ f_1 \times \mathtt{bind}^{\mathrm{M}_2}\ m_2\ f_2 \rightsquigarrow \mathtt{bind}^{\mathrm{P}}\ m_{\mathrm{rel}}\ f_{\mathrm{rel}}}$$

but also spells out how particular effects that P supports correspond to the effects from M_1 and M_2.

Second, to fully reproduce the product program methodology, we need to explain how specifications relate to product programs. We can use simple relational specification monads (§2.2) for specifying the properties on products programs. The lifting of unary specification monads described there extends to unary effect observations, providing an important source of examples of effect observations for product programs. For example, going back to the example of state, we can specify product programs in $\mathrm{P}(A_1, A_2) = \mathrm{St}_{S_1 \times S_2}(A_1 \times A_2)$ with specifications provided by the simple relational specification monad $\mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}$, and the effect observation $\zeta : \mathrm{P} \to \mathrm{W}^{\mathrm{St}}_{\mathrm{rel}}$ obtained by lifting the unary effect observation $\theta^{\mathrm{St}} : \mathrm{St} \to \mathrm{W}^{\mathrm{St}}$ of the introduction, resulting in

$$\zeta\ (f : S_1 \times S_2 \to (A_1 \times A_2) \times (S_1 \times S_2)) = \lambda\varphi\ (s_1, s_2)\,.\ \varphi\ \sigma(f\ (s_1, s_2))$$

where $\sigma : (A_1 \times A_2) \times (S_1 \times S_2) \to (A_1 \times S_1) \times (A_2 \times S_2)$ simply swaps the arguments. Then, the concrete proof verifying the property w in this step consists of proving $\zeta(c) \leq w$ as usual.

Finally, the third step simply relies on proving and then applying a soundness theorem for product programs. In the case of stateful computations, this theorem has the following form:

Theorem 5 (Soundness of product programs for state). *If $c_1 \times c_2 \rightsquigarrow c$ and $\zeta(c) \leq w$, then $\vDash_{\theta^{\mathrm{St}}_{rel}} c_1 \sim c_2\ \{\ w\ \}$.*

In this case, the soundness theorem is proved by analyzing the relation $c_1 \times c_2 \rightsquigarrow c$ and showing in each case that our choice of $\theta^{\mathrm{St}}_{\mathrm{rel}}$ and ζ agree.

The interpretation of product programs as computations in a relative monad accommodate well the product program methodology. In particular we expect that algebraic presentations of these relative monads used for product programs could shed light on the choice of primitive rules in relational program logics, in a Curry-Howard fashion. We leave this as a stimulating future work.

6 RELATED WORK

Many different relational verification tools have been proposed, making different tradeoffs, especially between automation and expressiveness. This section surveys this prior work, starting with the techniques that are closest related to ours.

Relational program logics. Relational program logics are very expressive and provide a formal foundation for various tools, which have found practical applications in many domains. Benton [2004] introduced Relational Hoare Logic (RHL) as a way to prove the correctness of various static analysis and optimizing transformations for imperative programs. Yang [2007] extended this to the relational verification of pointer-manipulating programs. Barthe et al.'s [2009] introduced pRHL as an extension of RHL to discrete probabilities and showed that pRHL can provide a solid foundation for cryptographic proofs, which inspired further research in this area [Barthe et al. 2014; Basin et al. 2017; Petcher and Morrisett 2015; Unruh 2019] and lead to the creation of semi-automated tools such

as EasyCrypt [Barthe et al. 2013a]. Barthe et al. [2013b] also applied variants of pRHL to differential privacy, which led to the discovery of a strong connection [Barthe et al. 2017] between coupling proofs in probability theory and relational program logic proofs, which are in turn connected to product programs even without probabilities [Barthe et al. 2016].

Carbin et al. [2012] introduced a program logic for proving acceptability properties of approximate program transformations. Nanevski et al. [2013] proposed Relational Hoare Type Theory (RHTT), a verification system for proving rich information flow and access control policies about pointer-manipulating programs in dependent type theory. Banerjee et al. [2016] addressed similar problems using a relational program logic with framing and hypotheses. Sousa and Dillig [2016] devised Cartesian Hoare Logic for verifying k-safety hyperproperties and implement it in the Descartes tool. Finally, Aguirre et al. [2017] introduced Relational Higher-Order Logic (RHOL) as a way of proving relational properties of *pure* programs in a simply typed λ-calculus with inductive types and recursive definitions. RHOL was later separately extended to two different monadic effects: cost [Radicek et al. 2018] and continuous probabilities with conditioning [Sato et al. 2019].

Each of these logics is specific to a particular combination of side-effects that is fixed by the programming language and verification framework. We instead introduce a general framework for defining program logics for *arbitrary* monadic effects. We show that logics such as RHL and HTT can be reconstructed within our framework, and we expect this to be the case for many of the logics above. It would also be interesting to investigate whether RHOL can also be extended to arbitrary monads, but even properly representing arbitrary monads, which is completely straightforward in dependent type theory, is not obvious in less powerful systems such as HOL. In this respect, Lochbihler [2018] recently built a library for effect polymorphic definitions and proofs in Isabelle/HOL, based on value-monomorphic monads and relators.

Type systems and static analysis tools. Various type systems and static analysis tools have been proposed for statically checking relational properties in a sound, automatic, but over-approximate way. The type systems for information flow control generally trade off precision for good automation [Sabelfeld and Myers 2003]. Various specialized type systems and static analysis tools have also been proposed for checking differential privacy [Barthe et al. 2015b; Gaboardi et al. 2013; Gavazzo 2018; Winograd-Cort et al. 2017; Zhang and Kifer 2017; Zhang et al. 2019] or doing relational cost analysis [Çiçek et al. 2017; Qu et al. 2019].

Product program constructions. Product program constructions and self-composition are techniques aimed at reducing the verification of k-safety hyperproperties [Clarkson and Schneider 2010] to the verification of traditional (unary) safety proprieties of a product program that emulates the behavior of multiple input programs. Multiple such constructions have been proposed [Barthe et al. 2016] targeted for instance at secure IFC [Barthe et al. 2011; Naumann 2006; Terauchi and Aiken 2005; Yasuoka and Terauchi 2014], program equivalence for compiler validation [Zaks and Pnueli 2008], equivalence checking and computing semantic differences [Lahiri et al. 2012], program approximation [He et al. 2018]. Sousa and Dillig's [2016] Descartes tool for k-safety properties also creates k copies of the program, but uses lockstep reasoning to improve performance by more tightly coupling the key invariants across the program copies. Antonopoulos et al. [2017] develop a tool that obtains better scalability by using a new decomposition of programs instead of using self-composition for k-safety problems. Eilers et al. [2018] propose a modular product program construction that permits hyperproperties in procedure specifications. Recently, Farzan and Vandikas [2019] propose an automated verification technique for hypersafety properties by constructing a proof for a small representative set of runs of the product program.

Logical relations and bisimulations. Many semantic techniques have been proposed for reasoning about relational properties such as observational equivalence, including techniques based

on binary logical relations [Ahmed et al. 2009; Benton et al. 2009, 2013, 2014; Dreyer et al. 2010, 2011, 2012; Mitchell 1986], bisimulations [Dal Lago et al. 2017; Koutavas and Wand 2006; Sangiorgi et al. 2011; Sumii 2009], and combinations thereof [Hur et al. 2012, 2014]. While these powerful techniques are often not directly automated, they can still be used for verification [Timany and Birkedal 2019] and for providing semantic correctness proofs for relational program logics [Dreyer et al. 2010, 2011] and other verification tools [Benton et al. 2016; Gavazzo 2018].

Other program equivalence techniques. Beyond the ones already mentioned above, many other techniques targeted at program equivalence have been proposed; we briefly review several recent works: Benton et al. [2009] do manual proofs of correctness of compiler optimizations using partial equivalence relations. Kundu et al. [2009] do automatic translation validation of compiler optimizations by checking equivalence of partially specified programs that can represent multiple concrete programs. Godlin and Strichman [2010] propose proof rules for proving the equivalence of recursive procedures. Lucanu and Rusu [2015] and Ştefan Ciobâcă et al. [2016] generalize this to a set of co-inductive equivalence proof rules that are language-independent. Wang et al. [2018] verify equivalence between a pair of programs that operate over databases with different schemas using bisimulation invariants over relational algebras with updates. Finally, automatically checking the equivalence of processes in a process calculus is an important building block for security protocol analysis [Blanchet et al. 2008; Chadha et al. 2016].

Reasoning about effectful semantics. Relating monadic expressions is natural and very wide-spread in proof assistants like Coq, Isabelle [Lochbihler 2018], or F* [Grimm et al. 2018], with various degrees of automation. Boulier et al. [2017]; Casinghino et al. [2014]; Pédrot and Tabareau [2018] extend dependent type theory with a few selected primitive effects: partiality, exceptions, reader. The resulting theory allows to some extent to reason directly on pairs of effectful programs, without resorting to a monadic encoding. In another line of work, Barthe et al. [2019] proposed to encode the semantics of imperative programs and their relational properties in an extension of first-order logic that can be automated by Vampire.

7 CONCLUSION AND FUTURE WORK

This paper introduced a principled framework for building relational program logics. We extended the work of Maillard et al. [2019] to the relational setting, and solved the additional challenges of correlating two independent computations with a relational specification, by we leveraging relative monads and introducing a novel notion of relative monad morphism. Now it's time to put this framework to the test and discover whether it can be in part automated, whether it can be scaled to realistic relational verification tasks, and whether it can deal with more complex effects.

In particular, it would be interesting to see whether our generic framework of §3 can support other control effects, such as breaking out of loops and continuations. These are, however, challenging to accommodate, even in our unary setting [Maillard et al. 2019]. Another interesting direction is providing a more precise treatment of nontermination. While the simple framework of §2 can already handle nontermination by choosing globally between total and partial correctness with an effect observation, the generic framework of §3 could allow to explicitly observe whether each computation terminates or not inside the relational specifications. This could allow one to choose at the specification level between partial or total correctness and between Benton [2004]'s or our semantics for RHL, or to define termination-sensitive noninterference, or that one computation terminates whenever the other does. Another interesting research direction, opened by the correspondence with product programs, would be to develop techniques to select which proof rules should be considered as primitive, using proof-theoretical tools like focusing [Zeilberger

2009], and also to investigate at the categorical level notions of presentations of relative monads, in connection with the theory of monads with arities [Berger et al. 2012].

ACKNOWLEDGMENTS

We thank Alejandro Aguirre, Danel Ahman, Robert Atkey, Gilles Barthe, Shin-ya Katsumata, Satoshi Kura, Guido Martínez, Ramkumar Ramachandra, Nikhil Swamy, Éric Tanter, and the anonymous reviewers for their helpful feedback. This work was, in part, supported by the European Research Council under ERC Starting Grant SECOMP (715753) and by Nomadic Labs via a grant on the "Evolution, Semantics, and Engineering of the F⋆ Verification System."

REFERENCES

C. Abate, R. Blanco, D. Garg, C. Hriţcu, M. Patrignani, and J. Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. *CSF*, 2019. To Appear.

A. Aguirre, G. Barthe, M. Gaboardi, D. Garg, and P.-Y. Strub. A relational logic for higher-order programs. *ICFP*, 2017.

D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. *POPL*. 2017.

A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. *POPL*. 2009.

T. Altenkirch, J. Chapman, and T. Uustalu. Monads need not be endofunctors. *LMCS*, 11(1), 2015.

T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei. Decomposition instead of self-composition for proving the absence of timing channels. *PLDI*. 2017.

P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in coq. In T. Uustalu, editor, *Mathematics of Program Construction*. 2006.

A. Banerjee, D. A. Naumann, and M. Nikouei. Relational logic with framing and hypotheses. *FSTTCS*. 2016.

G. Barthe, B. Grégoire, and S. Zanella-Béguelin. Formal certification of code-based cryptographic proofs. *POPL*, 2009.

G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *MSCS*, 21(6):1207–1252, 2011.

G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. EasyCrypt: A tutorial. In A. Aldini, J. Lopez, and F. Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. 2013a.

G. Barthe, B. Köpf, F. Olmedo, and S. Zanella-Béguelin. Probabilistic relational reasoning for differential privacy. *TOPLAS*, 35(3):9:1–9:49, 2013b.

G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. *POPL*. 2014.

G. Barthe, T. Espitau, B. Grégoire, J. Hsu, L. Stefanesco, and P. Strub. Relational reasoning via probabilistic coupling. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, 2015a.

G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, A. Roth, and P. Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. *POPL*. 2015b.

G. Barthe, J. M. Crespo, and C. Kunz. Product programs and relational program logics. *JLAMP*, 85(5):847–859, 2016.

G. Barthe, B. Grégoire, J. Hsu, and P. Strub. Coupling proofs are probabilistic product programs. *POPL*. 2017.

G. Barthe, R. Eilers, P. Georgiou, B. Gleiss, L. Kovács, and M. Maffei. Verifying relational properties using trace logic. Draft, 2019.

D. A. Basin, A. Lochbihler, and S. R. Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *IACR Cryptology ePrint Archive*, 2017:753, 2017.

N. Benton. Simple relational correctness proofs for static analyses and program transformations. *POPL*. 2004.

N. Benton, J. Hughes, and E. Moggi. Monads and effects. *APPSEM*. 2000.

N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. Relational semantics for effect-based program transformations: higher-order store. *POPL*. 2009.

N. Benton, M. Hofmann, and V. Nigam. Proof-relevant logical relations for name generation. *TLCA*. 2013.

N. Benton, M. Hofmann, and V. Nigam. Abstract effects and proof-relevant logical relations. *POPL*. 2014.

N. Benton, A. Kennedy, M. Hofmann, and V. Nigam. Counting successes: Effects and transformations for non-deterministic programs. In S. Lindley, C. McBride, P. W. Trinder, and D. Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. 2016.

C. Berger, P.-A. Melliès, and M. Weber. Monads with arities and their associated theories. *Journal of Pure and Applied Algebra*, 216(8-9):2029–2048, 2012. New introduction; Section 1 shortened and redispatched with Section 2; Subsections on symmetric operads (3.14) and symmetric simplicial sets (4.17) added; Bibliography completed.

B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. *J. Log. Algebr. Program.*, 75(1):3–51, 2008.
S. Boulier, P. Pédrot, and N. Tabareau. The next 700 syntactical models of type theory. *CPP*, 2017.
N. Bowler, S. Goncharov, P. B. Levy, and L. Schröder. Exploring the boundaries of monad tensorability on set. *Logical Methods in Computer Science*, 9(3), 2013.
M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. *PLDI*. 2012.
C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, 2014.
R. Chadha, V. Cheval, Ştefan Ciobâcă, and S. Kremer. Automated verification of equivalence properties of cryptographic protocols. *ACM Trans. Comput. Log.*, 17(4):23:1–23:32, 2016.
E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. Relational cost analysis. *POPL*, 2017.
M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010.
Ştefan Ciobâcă, D. Lucanu, V. Rusu, and G. Rosu. A language-independent proof system for full program equivalence. *Formal Asp. Comput.*, 28(3):469–497, 2016.
U. Dal Lago, F. Gavazzo, and P. B. Levy. Effectful applicative bisimilarity: Monads, relators, and Howe's method. *LICS*. 2017.
G. A. Delbianco and A. Nanevski. Hoare-style reasoning with (algebraic) continuations. *ICFP*. 2013.
D. Dreyer, G. Neis, A. Rossberg, and L. Birkedal. A relational modal logic for higher-order stateful ADTs. *POPL*. 2010.
D. Dreyer, A. Ahmed, and L. Birkedal. Logical step-indexed logical relations. *Logical Methods in Computer Science*, 7(2), 2011.
D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.*, 22(4-5):477–528, 2012.
M. Eilers, P. Müller, and S. Hitz. Modular product programs. In A. Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 2018.
F. Faissole and B. Spitters. Synthetic topology in homotopy type theory for probabilistic programming. PPS 2017 - Workshop on probabilistic programming semantics, 2017. Poster.
A. Farzan and A. Vandikas. Automated hypersafety verification. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. 2019.
C. Führmann. Varieties of effects. *FOSSACS*, 2002.
M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. *POPL*. 2013.
F. Gavazzo. Quantitative behavioural reasoning for higher-order effectful programs: Applicative distances. *LICS*. 2018.
T. Girka, D. Mentré, and Y. Régis-Gianas. A mechanically checked generation of correlating programs directed by structured syntactic differences. In *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings*, 2015.
T. Girka, D. Mentré, and Y. Régis-Gianas. Verifiable semantic difference languages. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017*, 2017.
M. Giry. A categorical approach to probability theory. *Categorical Aspects of Topology and Analysis*. 1982.
B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. In Z. Manna and D. A. Peled, editors, *Time for Verification, Essays in Memory of Amir Pnueli*. 2010.
N. Grimm, K. Maillard, C. Fournet, C. Hriţcu, M. Maffei, J. Protzenko, T. Ramananandro, A. Rastogi, N. Swamy, and S. Zanella-Béguelin. A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. *CPP*, 2018.
S. He, S. K. Lahiri, and Z. Rakamaric. Verifying relative safety, accuracy, and termination for program approximations. *J. Autom. Reasoning*, 60(1):23–42, 2018.
C. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and kripke logical relations. *POPL*. 2012.
C. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. A logical step forward in parametric bisimulations. Technical Report MPI-SWS-2014-003, 2014.
B. Jacobs. Dijkstra and Hoare monads in monadic computation. *Theor. Comput. Sci.*, 604:30–45, 2015.
C. Kapulkin and P. L. Lumsdaine. Homotopical inverse diagrams in categories with attributes, 2018.
S. Katsumata. Parametric effect monads and semantics of effect systems. *POPL*. 2014.
G. Kelly. *Basic Concepts of Enriched Category Theory*. Lecture note series / London mathematical society. Cambridge University Press, 1982.
V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. *POPL*. 2006.
S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. *PLDI*. 2009.
S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. SYMDIFF: A language-agnostic semantic diff tool for imperative programs. *CAV*. 2012.

A. Lochbihler. Effect polymorphism in higher-order logic (proof pearl). *JAR*, 2018.
D. Lucanu and V. Rusu. Program equivalence by circular reasoning. *Formal Asp. Comput.*, 27(4):701–726, 2015.
C. Lüth and N. Ghani. Composing monads using coproducts. *ICFP*. 2002.
K. Maillard, D. Ahman, R. Atkey, G. Martínez, C. Hriţcu, E. Rivas, and É. Tanter. Dijkstra monads for all. *PACMPL*, 3(ICFP): 104:1–104:29, 2019.
J. C. Mitchell. Representation independence and data abstraction. In *POPL*. 1986.
E. Moggi. Computational lambda-calculus and monads. *LICS*. 1989.
A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. *ICFP*. 2008a.
A. Nanevski, J. G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *JFP*, 18(5-6):865–911, 2008b.
A. Nanevski, A. Banerjee, and D. Garg. Dependent type theory for verification of information flow and access control policies. *ACM TOPLAS*, 35(2):6, 2013.
D. A. Naumann. From coupling relations to mated invariants for checking information flow. *ESORICS*. 2006.
P. Pédrot and N. Tabareau. Failure is not an option - an exceptional type theory. *ESOP*, 2018.
A. Petcher and G. Morrisett. The foundational cryptography framework. *POST*. 2015.
G. D. Plotkin and J. Power. Notions of computation determine monads. *FOSSACS*, 2002.
G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. *ESOP*. 2009.
W. Qu, M. Gaboardi, and D. Garg. Relational cost analysis for functional-imperative programs. To appear at ICFP, 2019.
I. Radicek, G. Barthe, M. Gaboardi, D. Garg, and F. Zuleger. Monadic refinements for relational cost analysis. *PACMPL*, 2 (POPL):36:1–36:32, 2018.
A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 33(1):5:1–5:69, 2011.
T. Sato, A. Aguirre, G. Barthe, M. Gaboardi, D. Garg, and J. Hsu. Formal verification of higher-order probabilistic programs: reasoning about approximation, convergence, bayesian inference, and optimization. *PACMPL*, 3(POPL):38:1–38:30, 2019.
M. Shulman. Univalence for inverse diagrams and homotopy canonicity. *Mathematical Structures in Computer Science*, 25: 1203–1277, 2014.
M. Sousa and I. Dillig. Cartesian Hoare logic for verifying k-safety properties. *PLDI*. 2016.
R. Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2, 1972.
E. Sumii. A complete characterization of observational equivalence in polymorphic *lambda*-calculus with general references. *CSL*. 2009.
N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. *PLDI*, 2013.
N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.
T. Terauchi and A. Aiken. Secure information flow as a safety problem. *SAS*. 2005.
A. Timany and L. Birkedal. Mechanized relational verification of concurrent programs with continuations. To appear at ICFP, 2019.
A. Timany, L. Stefanesco, M. Krogh-Jespersen, and L. 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.
S. Tonelli. Investigations into a model of type theory based on the concept of basic pair. Master's thesis, Stockholm University, 2013. supervisors Erik Palmgren and Giovanni Sambin.
D. Unruh. Quantum relational Hoare logic. *PACMPL*, 3(POPL):33:1–33:31, 2019.
Y. Wang, I. Dillig, S. K. Lahiri, and W. R. Cook. Verifying equivalence of database-driven applications. *PACMPL*, 2(POPL): 56:1–56:29, 2018.
D. Winograd-Cort, A. Haeberlen, A. Roth, and B. C. Pierce. A framework for adaptive differential privacy. *PACMPL*, 1(ICFP): 10:1–10:29, 2017.
H. Yang. Relational separation logic. *Theor. Comput. Sci.*, 375(1-3):308–334, 2007.
H. Yasuoka and T. Terauchi. Quantitative information flow as safety and liveness hyperproperties. *Theor. Comput. Sci.*, 538: 167–182, 2014.
A. Zaks and A. Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. *FM*. 2008.
N. Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009.
D. Zhang and D. Kifer. LightDP: towards automating differential privacy proofs. *POPL*. 2017.
H. Zhang, E. Roth, A. Haeberlen, B. C. Pierce, and A. Roth. Fuzzi: A three-level logic for differential privacy. *CoRR*, abs/1905.12594, 2019.