



École Normale Supérieure

Thèse de Doctorat

Spécialité Informatique

Principles of Program Verification for Arbitrary Monadic Effects

Kenji Maillard (ENS Ulm and Inria Paris)

Directeur :

Cătălin Hrițcu (Inria Paris)

Rapporteurs :

Shin-ya Katsumata (National Institute of Informatics, Tokyo, Japan)

Bas Spitters (Aarhus University, Denmark)

Examineurs :

Robert Atkey (University of Strathclyde, Glasgow, UK)

Gilles Barthe (MPI for Security and Privacy, Germany and IMDEA, Spain)

Pierre-Évariste Dagand (CNRS, LIP6, Paris)

Hugo Herbelin (Inria, PPS, IRIF, Paris)

Christine Paulin-Mohring (LRI, Université Paris Sud and Inria Saclay)

Nicolas Tabareau (Inria, IMT Atlantique, Nantes)

Monday 25th November, 2019

Abstract

Computational monads are a convenient algebraic gadget to uniformly represent side-effects in programming languages, such as mutable state, divergence, exceptions, or non-determinism. Various frameworks for verifying that programs and meet their specification have been proposed, but are all specific to a particular combination of side-effects. For instance, one can use Hoare logic to verify the functional correctness of programs with mutable state with respect to pre/post-conditions specifications.

This thesis devises a principled semantic framework for verifying programs with arbitrary monadic effects in a generic way with respect to such expressive specifications. The starting point are Dijkstra monads, which are monad-like structures that classify effectful computations satisfying a specification drawn from a monad. Dijkstra monads have already proven valuable in practice for verifying effectful code, and in particular, they allow the F^* program verifier to compute verification conditions.

We provide the first semantic investigation of the algebraic structure underlying Dijkstra monads and unveil a close relationship between Dijkstra monads and effect observations, i.e., mappings between a computational and a specification monad that respect their monadic structure. Effect observations are flexible enough to provide various interpretations of effects, for instance total vs partial correctness, or angelic vs demonic nondeterminism. Our semantic investigation relies on a general theory of specification monads and effect observations, using an enriched notion of relative monads and relative monad morphisms. We moreover show that a large variety of specification monads can be obtained by applying monad transformers to various base specification monads, including predicate transformers and Hoare-style pre- and postconditions. For defining correct monad transformers, we design a language inspired by the categorical analysis of the relationship between monad transformers and algebras for a monad.

We also adapt our framework to relational verification, i.e., proving relational properties between multiple runs of one or more programs, such as noninterference or program equivalence. For this we extend specification monads and effect observations to the relational setting and use them to derive the semantics and core rules of a relational program logic generically for any monadic effect. Finally, we identify and overcome conceptual challenges that prevented previous relational program logics from properly dealing with effects such as exceptions, and are the first to provide a proper semantic foundation and a relational program logic for exceptions.

À Jacques

Remerciements

Je remercie tous les membres du jury de s'être déplacés pour assister à cette soutenance. Merci en particulier à Shin-Ya et Bas pour leur relecture attentive de ce manuscrit et leurs multiples questions pertinentes.

Je tiens d'abord à remercier les multiples personnes qui m'ont introduit à la recherche, académique ou pas, notamment Masahito Hasegawa, Paul-André Melliès, Rosen Diankov et Nikhil Swamy qui m'ont encadré au cours de long stages à Kyoto, Paris, Tokyo et Seattle. Les points de vue variés sur la recherche qu'ils ont partagés m'ont fortement inspiré tout au long de ma thèse.

L'équipe Prosecco m'a accueilli malgré un parcours et un bagage scientifique un peu étranger à leur cœur de métier. Si j'ai pu me sentir perdu, plongé dans des domaines pour moi très exotiques, je remercie tous les membres de l'équipe pour le lieu stimulant qu'ils ont fait vivre : Nadim pour ses protocoles agrémentés de chatons kawaii, Marc pour ses questions sur $G(\text{rothendieck})\text{tk}$, Natalia, Ben et Marina pour les multiples occasions de débbugger un contexte SMT défaillant, Tomer pour son ambition de réécrire F^* en λ -prolog, Victor pour son stoïcisme devant des tâches qui en auraient rebuté plus d'un, Danel pour ses explications des catégories de compréhension avec ou sans modalités, Mathieu pour son soutien discret et efficace, Karthik et Bruno pour avoir monté Prosecco. Au fur et à mesure, Théo, Rob, Carmine, Jérémy, Blipp, Éric, Exe, Antoine et autres trublions sont venus mettre leur grain de sel, faisant parfois évoluer l'équipe au point que les termes d' $(\infty, 1)$ -catégories, d'univalence, de paramétrie et de recollement d'Artin sont devenus fréquents dans un certain groupe de lecture. Je n'oublie pas Jonathan et Guido, membres intermittents, passant régulièrement comme des spoutniks et dont les apports scientifiques comme gastronomiques ont considérablement enrichi mon expérience de doctorant.

Cette thèse n'a aussi été possible qu'avec le soutien continu des doctorants (sur plusieurs générations) et chercheurs de l'ex-pôle PPS que j'ai continué à hanter pendant longtemps, au point de se demander dans quel laboratoire j'effectuais ma thèse. Je me dois donc de remercier Daniel et ses éponges, Rémi et ses émois, les catégories terroristes Pierre, Chaït, Axel, Leonard, ainsi que Théo, Léo, Zeïnab, Nicolas, Victor, Tomaso, Simona qui ont organisé gâteaux, bières et autres séances de cinéma clandestines.

Merci à Gabriel pour m'avoir régulièrement poussé à me mettre au boulot et notamment lancé un grand mouvement d'étude sur le $\mu\tilde{\mu}$ (à poil ras) et la polarisation. Ce fut l'occasion de faire connaissance avec les membres de l'équipe Galinette dont l'expérience en $\mu\tilde{\mu}$ et en Coq m'a souvent été d'une grande aide. Merci donc à Guillaume, Xavier, Étienne, Théo, Simon et à l'ensemble de cette formidable équipe.

L'environnement scientifique riche de Paris m'a certainement fourni l'occasion de cultiver ma compréhension de domaines connexes, que ce soit des catégories de modèle ou du ROP¹, cependant je me dois de remercier les organisateurs d'événements extra-muros, en particulier des rencontres ChoCoLa à Lyon, des réunions GeoCal-LAC puis plus tard de celles de SCalP et LHC pour permettre un brassage plus large des idées et des personnalités. À l'international[e], les rencontres EUTYPES m'ont beaucoup apporté, et m'ont permis entre autres d'alourdir mon bilan carbone.

¹Return Oriented Programming

Je ne sais comment remercier Cătălin pour avoir accepté d'encadrer ma thèse dans un contexte ubuesque² sur un sujet aussi prometteur qu'ardu. Il n'y pas de doute que son suivi, ses conseils, sa motivation et son travail acharné ont permis au volume sous vos yeux de voir le jour.

Les nombreuses années passées à Paris ont été rythmées par l'intervention de nombreux joyeux lurons : merci Abel, Julie, Basile, Ambre, Annali, Guillaume, Ulysse, Anaël, Najib, Athanarik pour les expériences de cuisine un peu folle, les après-midi jeux et d'avoir été là.

Un remerciement très particulier à Marina et Jun qui ont subi à longueur d'années des expérimentations culinaires plus ou moins douteuses en fonction de mes sautes d'humeur fréquentes; à Anya qui m'a fait découvrir tant de choses que j'ignorais.

²Je prétendais étudier la cohomologie des faisceaux et ai soudainement décidé de partir programmer des robots déplaceur de boulons au Japon, comme quoi la cuisine mène à pleins de choses...

Contents

Contents	v
List of Figures	vi
1 Introduction	1
1.1 Reasoning About Monadic Programs	2
1.2 Understanding Dijkstra monads	4
1.3 Relational reasoning for arbitrary effects	5
1.4 Contributions	6
1.5 Foundations, Conventions & Notations	7
1.6 Outline	8
2 Enter the monad	11
2.1 Monads for the working programmer	12
2.2 Taming the monad zoo: a first glance at monad transformers	15
2.3 Specifications from monads	17
2.4 Effect observations	20
2.5 Conclusion & Related work	25
3 Abstracted away	29
3.1 Elements of the formal theory of monads	29
3.2 Relative monads	33
3.3 Framed bicategories	34
3.4 Framed functor, framed representability	38
3.5 Relative monad in a framed bicategory	41
3.6 Conclusion & Related work	49
4 Mass producing monad transformers	51
4.1 What is a monad transformer ?	51
4.2 Towards a language for defining monad transformers	52
4.3 A DSL for specification monad transformers	53
4.4 Embedding SM in Coq	64
4.5 Towards a categorical approach to relative monad transformers	67
4.6 Conclusion & Related work	72
5 Dijkstra monads	75
5.1 Definition & examples	75
5.2 Equivalence with effect observations	82
5.3 Dijkstra monads as relative monads, connection to graded monads	86
5.4 Conclusion & Related work	89
6 Relational reasoning	91

6.1	The logic of relational rules	91
6.2	Simplified Framework	95
6.3	Generic Framework	101
6.4	Product programs	109
6.5	Related work	111
6.6	Conclusion	113
Bibliography		115

List of Figures

1.1	Chapter dependencies	8
2.1	Basic specification monads	20
4.1	Syntax of SM	53
4.2	Typing rules for SM	54
4.3	Equational theory of SM	55
4.4	Elaboration of types from SM to \mathcal{L}	56
4.5	Denotation of SM terms	56
4.6	Typing rules for SM with linearity condition	61
4.7	PHOAS definition of the term syntax of SM	65
4.8	Binary parametricity predicate	66
4.9	Term syntax of SM using De Bruijn indices	66
4.10	Implementation of the state monad internally to SM	66
4.11	Reduction of abstract machine configurations	68
4.12	Stacks and Abstract machine reduction for SM	68
6.1	Pure relational rules	99
6.2	Generic monadic rules in the simple framework	99
6.3	Syntax of RDTT and translation to base type theory	103
6.4	Translation of the eliminator for sums in RDTT	103
6.5	Generic monadic rules in the full relational setting	108
6.6	Rules for exceptions	110

Chapter 1

Introduction

«En un mot, la cuisine, sans cesser d’être un art, deviendra scientifique et devra soumettre ses formules, empiriques trop souvent encore, à une méthode et à une précision qui ne laisseront rien au hasard.»

Auguste Escoffier, *Le Guide culinaire*, 1907

This manuscript is not dedicated to the art of cuisine, but to the science of computers and more precisely to *programs*, which are the recipes used by computers. From this point of view, a computer can be seen as a cook faithfully executing each step of a recipe in order to obtain a result. Since we are a picky customer, we do not accept just any kind of result though, and require the best quality, provided by formally verified programs.

The first task of program verification is to describe the expected behaviour of a program, via a formal description called a *specification*. The crux of program verification is to *prove* that the behavior of the program indeed satisfies the specification. For a simple example, consider the following program computing the Fibonacci sequence:

```
let rec fib (n : ℤ) : ℤ = if n ≤ 1 then n else fib (n - 1) + fib (n - 2)
```

What can we say about this program? From a mathematical point of view, we can *solve* the recursive equation $u_{n+2} = u_{n+1} + u_n$ with initial conditions $u_0 = 0, u_1 = 1$, obtaining the closed form $u_n = \frac{1}{\sqrt{5}}(\varphi^n - \varphi'^n)$ where $\varphi = \frac{1+\sqrt{5}}{2}$ and $\varphi' = -\frac{1}{\varphi}$. We could then specify that for any $n \geq 0$, *fib* *n* computes u_n , and to obtain a complete specification of *fib*, we should also explain what happens for negative integers $n < 0$, namely that it returns *n*. However, formally proving such a precise specification can be difficult. In this particular case, it entails replaying the standard mathematical proof providing the closed form u_n , an accessible but time consuming task. In certain scenarios, it might be enough for our purpose to prove a weaker, less precise specification, but much easier to show, for instance that *fib* *n* ≥ 0 . In general, there are many different specifications that we can assign to a program for the purpose of verification.

Now, suppose that some careless programmer were to write the following variation to compute the Fibonacci sequence:

```
let rec fib' (n : ℤ) : ℤ = if n = 0 || n = 1 then n else fib' (n - 1) + fib' (n - 2)
```

This implementation does not change much from the previous, the condition $n \leq 1$ was just replaced by $n = 0 \parallel n = 1$ and, for $n \geq 0$ it actually computes the same values. However, if you were to feed a negative integer, say -38 , to *fib'*, the following infinite reduction sequence unrolls

$$\begin{aligned} \text{fib}' -38 &\rightsquigarrow \text{fib}' -39 + \text{fib}' -40 \\ &\rightsquigarrow (\text{fib}' -40 + \text{fib}' -41) + (\text{fib}' -41 + \text{fib}' -42) \\ &\rightsquigarrow \dots \end{aligned}$$

and will continue executing for quite some time, since it will never hit the base case $n == 0 \parallel n == 1$. We call such a program that sometimes never returns a value a *divergent* or *partial* program, by opposition to a *total* program that always answers after computing for a finite – but arbitrary – number of steps. While this simple example is quite contrived, considering partial programs is a necessity if we want to implement expressive programs such as an evaluator for a Turing-complete programming language. And from a program verification perspective, it means that we need to be able to specify such partial programs and consequently specifications should have the ability to specify not only the value a program may return but also how partial it is.

This is the point where computations take their independence from the idealistic world of pure, total, mathematical functions. Concretely, *side-effects* can be used to distinguish the evaluation strategy employed to evaluate a program, so the latter can no longer be naively modelled as function returning a result. Nonetheless, to achieve anything, a useful program must at some point trigger effects to interact with the external world. Examples of such interactions are querying a user for input, storing persistent data to the file system, or exploring an unbounded search space, possibly nondeterministically. Since effects are ubiquitous in our daily programming activity, we would like to understand them deeply. We seek a solid and general theory explaining what effects are, how we can use them to write useful programs, and most importantly, how we can reason about the properties of such programs. As such, our work builds upon the general model of side-effects as *computational monads* (Moggi, 1989), which can naturally capture effects such as stateful computations, exceptions, non-termination, nondeterminism, or probabilities.

The aim of this thesis is to deepen our conceptual understanding of these monadic effects and to work out the general principles of program verification for programs with arbitrary monadic side-effects. To this end, we study a few areas of program verification and systematically associate to a program logic (i.e., a deductive system for proving assertions about programs) an algebraic semantic counterpart. These algebraic objects consist of various generalizations of monads and morphisms preserving the monadic structure. In the following sections we introduce these objects and how they help program verification: *specification monads* to describe the behaviour of programs, *effect observation* to connect computations with specifications, and *Dijkstra monads* to bind the three together, as well as their *relational* variants. A running idea throughout is that the common algebraic laws underlying the semantics of various program logics for specific effects provides insight into the nature of effects themselves.

1.1 Reasoning About Monadic Programs

Many approaches have been proposed for formally verifying effectful programs. In an imperative setting, Hoare (1969) introduced a *program logic* to reason about properties of programs. The judgments of this logic are *Hoare triples* of the form $\{pre\} c \{post\}$. Intuitively, if the precondition pre is satisfied, then running the program c leaves us in a situation where $post$ is satisfied, provided that c terminates. For imperative programs—i.e., statements changing the program’s state— pre and $post$ are predicates over the initial and the final state. These Hoare triples are derived using inference rules such as

$$\text{HOARE-SKIP} \frac{}{\{q\} \text{ skip } \{q\}} \quad \text{HOARE-SEQ} \frac{\{pre\} c_1 \{q\} \quad \{q\} c_2 \{post\}}{\{pre\} c_1; c_2 \{post\}} \quad (1.1)$$

Hoare’s approach can be directly adapted to the monadic setting by replacing imperative programs c with monadic computations $m : M A$. This approach was first proposed in Hoare Type Theory (Nanevski et al., 2008a,b), where a *Hoare monad* of the form $\text{HST } pre \ A \ post$ augments the state monad over A with a precondition $pre : S \rightarrow \mathbb{P}$ and postcondition $post : A \times S \rightarrow \mathbb{P}$. So while preconditions are, like in Hoare logic, predicates over initial states, postconditions are now predicates over both final states and results. Using this Hoare monad, we can reflect the

inference rules of Hoare logic inside the typing judgements

$$\begin{array}{c} \text{HTT-SKIP} \frac{}{\Gamma \vdash \text{skip} : \text{HST} (\lambda s. \text{post} ((), s)) \perp \text{post}} \\ \\ \text{HTT-SEQ} \frac{\Gamma \vdash c_1 : \text{HST} \text{pre} \perp q \quad \Gamma \vdash c_2 : \text{HST} (\lambda s. q ((), s)) A \text{post}}{\Gamma \vdash c_1; c_2 : \text{HST} \text{pre} A \text{post}} \end{array}$$

where we write `skip` for the monadic program returning `()` and $c_1; c_2$ for the sequential composition of monadic programs dropping the (irrelevant) result of c_1 . While this approach was successfully extended to a few other effects (Delbianco and Nanevski, 2013; Nanevski et al., 2008a, 2013), until our work, there was no general story on how to define a Hoare monad or even just the shape of pre- and postconditions for an arbitrary effect.

A popular alternative to proving properties of imperative programs is Dijkstra’s (1975) *weakest precondition calculus*. The main insight of this calculus is that from the syntax of a program c we can directly compute a weakest precondition $\text{wp}(c, \text{post})$ such that the formula $\text{pre} \Rightarrow \text{wp}(c, \text{post})$ is valid if and only if the triple $\{ \text{pre} \} c \{ \text{post} \}$ is derivable, which allows to partly automate the verification process by reducing it to a logical validity problem. Swamy et al. (2013) observed that it is possible to adopt Dijkstra’s technique to ML programs with state and exceptions elaborated to monadic style. They propose a notion of *Dijkstra monad* of the form $\text{DST } A \text{ wp}$, classifying stateful programs with exceptions returning values in A and where wp is a *predicate transformer* that specifies the behavior of the monadic computation. These predicate transformers are represented as functions that, given a postcondition on the final state, and either the result value of type A or an exception of type E , calculate a corresponding precondition on the initial state. The type of such predicate transformers can be written as follows (where \mathbb{P} is the type of propositions):

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

In subsequent work, Swamy et al. (2016) extended this to programs that combine multiple sub-effects. They compute more efficient weakest preconditions with respect to the actual effects of the code, instead of verifying everything using W^{ML} above. For example, pure computations are given specifications of type:

$$W^{\text{Pure}} A = \text{Cont}_{\mathbb{P}} A = (A \rightarrow \mathbb{P}) \rightarrow \mathbb{P},$$

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

$$W^{\text{St}} A = (A \times S \rightarrow \mathbb{P}) \rightarrow (S \rightarrow \mathbb{P}).$$

An important observation underlying this technique is that predicate transformers have a natural monadic structure ensuring that analogs of the inference rules 1.1 hold for each of these settings. For instance, it is not hard to see that the predicate transformer type W^{Pure} is simply the continuation monad with answer type \mathbb{P} , that W^{St} is the state monad transformer applied to W^{Pure} , and that W^{ML} is the state and exceptions monad transformers applied to W^{Pure} . While this observation was historically made for W^{Pure} and W^{St} , where the monad structure is more obvious, we realized in retrospective that the pre-/post-conditions used in Hoare logic also have such a monadic structure inducing exactly the rules (1.1):

$$\text{PP}^{\text{St}} A = (S \rightarrow \mathbb{P}) \times (A \times S \rightarrow \mathbb{P})$$

Generalizing over these examples, we introduce the notion of *specification monad*, capturing abstractly this class of monads expressing specifications. These monadic structures are a key ingredient of both Hoare monads and Dijkstra monads, providing a unified view of the specifications indexing these objects. Moreover, we investigate generic constructions of such specification monads, in particular based on monad transformers, which reveals a rich theory that can account for specifications for a variety of side-effects.

1.2 Understanding Dijkstra monads

Generalizing over the previous discussion, a Dijkstra monad $\mathcal{D} \ A \ w$ is a monad-like structure that classifies effectful computations returning values in A and specified by $w : W A$, where W is what we call a *specification monad*.¹ The pragmatic observation that Dijkstra monads and the associated verification methodology is effective for various effects (Swamy et al., 2016) led us to a quest to generalize Dijkstra monads to *arbitrary* monadic effects. The main questions to answer are: Given a monadic effect, how do we find a suitable specification monad for it? Is there a *single* specification monad that we can associate to each effect? If not, what are the *various* alternatives, and what are the constraints on this *association* for obtaining a proper Dijkstra monad?

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

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

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

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

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

The key idea at this point is to decouple the computational monad and the specification monad: instead of insisting on deriving both from the same monad transformer as in *DM4Free*, we consider them independently and only require that they are related by an *effect observation* (Katsumata, 2014), i.e., a mapping between two monads that respects their monadic structure.

$$\begin{array}{ccc} \mathbf{M} & \xrightarrow{\theta} & \mathbf{W} \\ \text{computational} & \text{effect observation} & \text{specification} \\ \text{monad} & & \text{monad} \end{array}$$

For instance, an effect observation from nondeterministic computations could map a finite set of possible outcomes to a predicate transformer in $(A \rightarrow \mathbb{P}) \rightarrow \mathbb{P}$. Given a finite set R of results in A and a postcondition $post : A \rightarrow \mathbb{P}$, there are only two reasonable ways to obtain a single proposition: either take the *conjunction* of $post\ v$ for every v in R (demonic nondeterminism), or the *disjunction* (angelic nondeterminism). For the case of IO, in our framework we can consider at least two effect observations relating the IO monad to two different specification monads, \mathbf{W}^{Fr} and \mathbf{W}^{Hist} , where \mathcal{E} is the alphabet of IO events:

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

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

How do these effect observations compare to Dijkstra monads? It turns out that they are two sides of the same coin: from an effect observation one can reconstruct a Dijkstra monad and conversely. In particular, thanks to the many degrees of freedom allowed by effect observations, we construct various novel Dijkstra monads in a uniform way.

1.3 Relational reasoning for arbitrary effects

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; Ștefan Ciobăcă 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 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 showing the correctness of 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, 2015; 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 trade-offs, for instance between automation and expressiveness (see section 6.5 for further discussion). In this manuscript, 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 we relate. 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, 2015, 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, general recursion, cost, and probabilities.

Leveraging the ideas developed in the unary (i.e., non-relational) setting outlined in [section 1.2](#), we distill the generic *relational reasoning principles* that work for many, if not all, monadic side-effects and that underlie relational program logics. An important insight is that the notion of specification monad can be extended to encompass relational specifications capturing a shared behaviour or a comparison of the behaviours of two programs, while keeping a compositional monad-like structure. For instance, considering two stateful programs $c_1 : \text{St}_{S_1} A_1$ and $c_2 : \text{St}_{S_2} A_2$, we can specify their behaviour by a pair of a precondition $pre : S_1 \times S_2 \rightarrow \mathbb{P}$ relating the initial states of the two programs and a postcondition $post : (A_1 \times S_1) \times (A_2 \times S_2) \rightarrow \mathbb{P}$ relating their results and final states. The specification monad structure on $\text{PP}_{\text{rel}}^{\text{St}}$ carries over to the type constructor

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

providing return and bind operations that make $\text{PP}_{\text{rel}}^{\text{St}}$ a *relational specification monad*. These relational specifications account for pairs of programs returning values in potentially distinct types. Likewise, our framework can relate programs using different computational monadic effects M_1, M_2 . *Relational effect observations* bridge the gap between these two computational monads and a relational specification monad W_{rel} :

$$\begin{array}{ccc} M_1, M_2 & \xrightarrow{\theta_{\text{rel}}} & W_{\text{rel}} \\ \text{left and right} & \text{relational} & \text{relational} \\ \text{computational monads} & \text{effect observation} & \text{specification monad} \end{array}$$

The diagram above provides a generic reconstruction of the semantics of relational program logics for arbitrary monadic effects. The game is then to reconstruct as canonically as possible the inference rules of relational program logics. In particular we observe that a clean separation can be achieved between *logical rules* independent of the computational effects, *generic monadic rules* ensuring compositionality of reasoning induced by the algebraic properties of relational effect observations, and *effect specific rules* that capture the specific details of the computational effects at hand. We show that logical and generic rules can be derived generically, independently of the effect, and we also provide a recipe for deriving the effect specific rules in our framework.

1.4 Contributions

- ▷ We provide a general theory of specification monads and effect observations that is useful for program verification. For specifications we identify various elementary specification monads such as Dijkstra-style predicate transformers as well as Hoare-style pre/postconditions, and extend the expressivity of these specification monads by applying

monad transformers. For effect observations we use relative monad morphisms to provide a flexible interpretation of effects, allowing for instance the choice between total and partial correctness, or between angelic and demonic nondeterminism.

- ▷ We develop a metalanguage for defining (specification) monad transformers whose design was inspired by the categorical analysis of the relationship between monad transformers and algebras for a monad. We implement the metalanguage in Coq, ultimately providing an effective method to derive correct-by-construction monad transformers out of a standard monad definition in the metalanguage.
- ▷ We provide the first formal definition of Dijkstra monads and unveil their close relationship to effect observations, yielding an effective method to build a variety of Dijkstra monads, and a practical methodology to verify effectful code for arbitrary monadic effects.
- ▷ We extend the notions of specification monads and effect observations to the relational setting, by introducing a general semantic framework for deriving relational program logics for arbitrary monadic effects.
- ▷ We identify and overcome conceptual challenges that prevented previous relational program logics from properly dealing with exceptions. For this, we propose a novel way of combining unary and relational specifications resulting in the first relational program logic for exceptions.
- ▷ We work out a theory of relative monads and use it to provide a unified conceptual foundation for specification monads and effect observations both in the unary and relational setting, as well as a presentation of Dijkstra monads as the lifting of relative monads.

This thesis is based on two recent papers: one that appeared at ICFP 2019 (Maillard et al., 2019a) and to appear at POPL 2020 (Maillard et al., 2019b). This is the culmination of a line of collaborative research in which I was involved during my PhD, which also resulted in other publications (Ahman et al., 2017, 2018; Bhargavan et al., 2017; Grimm et al., 2018).

1.5 Foundations, Conventions & Notations

We work as much as possible in a constructive metatheory that is loosely modelled on Coq, i.e., Martin-Löf Type Theory with dependent product $(x:A) \rightarrow B$, dependent sums $(x:A) \times B$, a predicative hierarchy of universes Type_i and an impredicative universe of proposition \mathbb{P} . Throughout the manuscript we assume extensionality for dependent products and sums, and propositions:

$$\begin{aligned} f = g : (x : A) \rightarrow B & \iff \forall (a : A), f\ a = g\ a : B[a/x] \\ u = v : (x : A) \times B & \iff \pi_1\ u = \pi_1\ v : A \wedge \pi_2\ u = \pi_2\ v : B[\pi_1\ u/x] \\ p = q : \mathbb{P} & \iff p \iff q \end{aligned}$$

We use the notation $\mathbb{1}$ to describe a terminal object, either a singleton or the category with one object and one identity arrow depending on the context. The unique morphism to $\mathbb{1}$ will be written $!_X$ where X is the domain of the morphism. When writing programs, we use either $*$ or $()$ to denote the unique inhabitant of $\mathbb{1}$.

We naively assume from times to times that equality on arbitrary types is proof-irrelevant, that is we assume *Uniqueness of Identity Type* (UIP), but we expect that most of the development could be achieved in a metatheory where UIP does not hold by restricting some of our constructions – e.g., indexed algebraic structures with equations such as Dijkstra monads – to types for which it holds, i.e., *hsets* in the terminology of Homotopy Type Theory (Univalent Foundations Program, 2013). The exception is [chapter 3](#) which uses quite a few classical results from the category theory literature whose constructive nature we ignore. Nevertheless, our implementation

in Coq derived from the ideas of that chapter comfort us in the opinion that there should be little obstruction, but a long and hard work to fully formalize it in a constructive metatheory.

Most of the programs illustrating this manuscript are written in a syntax freely inspired from F^* , with the exception of a few code listings in [chapter 4](#) describing the Coq implementation and consequently written directly in Coq. A substantial amount of the formalization done during the thesis preparation can be found at <https://gitlab.inria.fr/kmaillar/dijkstra-monads-for-all>. Sections and proofs that have been formalized end with a 🦉.

1.6 Outline

We close this introduction with a plan of the coming chapters and their logical dependencies presented in [Figure 1.1](#).

Chapter 2 first introduces computational monads from a programmer perspective, illustrating various effects that can be expressed as monads. This is followed by a few examples of monad transformers, which are the traditional way to build the zoo of monads modularly. The main contributions of this chapter are the introduction of specification monads and the investigation of effect observations, essential bridges between computational monads and specification monads.

Chapter 3 dives into the categorical world. It starts by recalling the formal theory of monads in a 2-category, introducing the main theoretical concepts enabling an abstract study of monads. The goal of this chapter is then to extend this formal theory to relative monads, a generalization that we achieve thanks to the notion of framed bicategory. A particular instantiation of relative monads in a framed bicategory provides an abstract definition of specification monads amenable to uniform generalizations to other settings such as relational verification.

Chapter 4 introduces a methodology for building correct monad transformers. While the theoretical foundations of this methodology is categorical, a more practical approach based on a syntactic meta-language for defining monad transformers is also introduced. We present the design choices guiding the implementation of this meta-language in the Coq proof assistant, ultimately providing an effective tool for generating verified monad transformers in Coq.

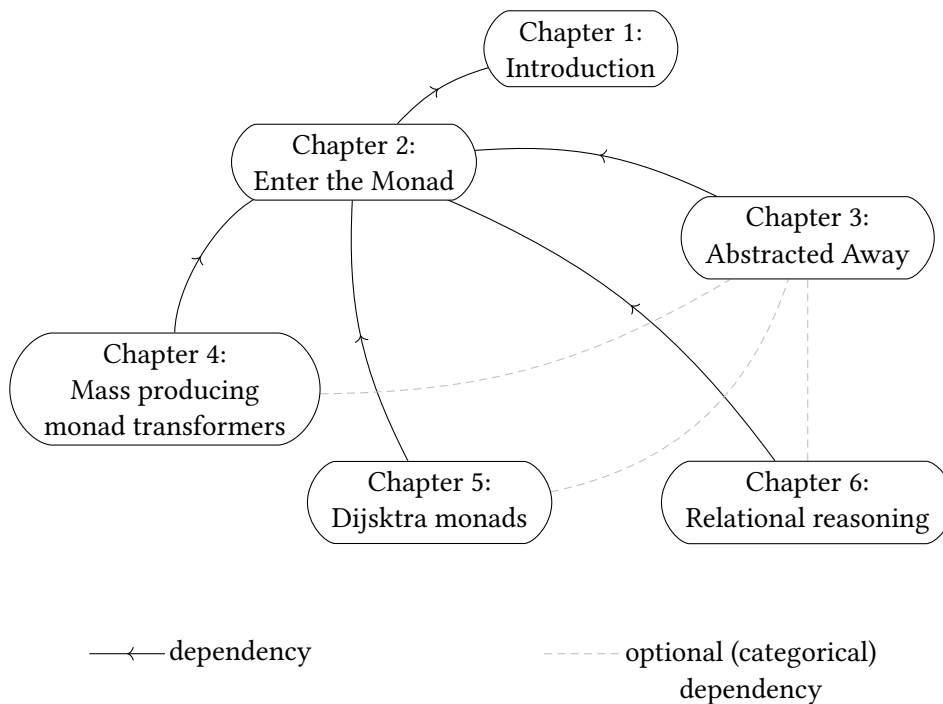


Figure 1.1: Chapter dependencies

Chapter 5 studies Dijkstra monads, a pragmatic approach to program verification, based on computation types indexed by specifications, which is used heavily in the F^* programming language. After defining Dijkstra monads, we provide some examples of their wide applicability. The main result of this chapter is the correspondence between Dijkstra monads and effect observations. A connection with graded monads (Fujii et al., 2016) is sketched using the unifying notion of relative monad.

Chapter 6 enters the realm of relational reasoning. We extend the notions of specification monad and effect observation to that setting, providing a general semantic foundation upon which we can define relational program logics for a variety of monadic effects. The case of exceptions is especially challenging and we explain how our framework can relate programs with exceptions by intertwining unary and relational reasoning.

Chapter 2

Enter the monad

«[...]
un soleil d'Austerlitz
un siphon d'eau de Seltz
un vin blanc citron
un Petit Poucet un grand pardon un calvaire de
pierre une échelle de corde
deux sœurs latines trois dimensions douze apôtres
mille et une nuits trente-deux positions six
parties du monde cinq points cardinaux dix
ans de bons et loyaux services sept péchés
capitaux deux doigts de la main dix gouttes
avant chaque repas trente jours de prison
dont quinze de cellule cinq minutes
d'entracte

et...

plusieurs ratons laveurs.»

Jacques Prévert, Inventaire, *Paroles*, 1946


This chapter provides a scenery of the basic notions that will be manipulated in the rest of the manuscript. The first two sections recall the well-known notions of *computational monads* and *monad transformers*. We explain how the former encapsulates side-effects in a uniform algebraic structure and how the latter provides a way to extend monads, achieving some amount of modularity.

We then introduce the novel notion of *specification monad*, a class of monads capturing specifications for effectful computations, casting specifications on the same footing as computations. Finally, our first tool for verification of programs with arbitrary monadic effects consist of a bridge between computations and specifications that we call an *effect observation* since it encodes a choice observation of a computational effect in a specification monad. Articulating computational monads and specification monads with effect observations turns out to provide a modular method to define verification system.

We provide examples for each introduced notion, and we will return to these examples throughout the thesis. Most of them have been defined inside Coq as part of an effort to provide a mechanized formalization of the content of this manuscript.

2.1 Monads for the working programmer

Side effects are an important part of programming. They arise in a multitude of shapes, be it imperative algorithms, nondeterministic operations, potentially diverging computations, or interactions with the external world. These various effects can be uniformly captured by the algebraic structure known as a *computational monad* (Benton et al., 2000; Moggi, 1989).

Definition 2.1.1 () *A monad is a type constructor $M : \text{Type} \rightarrow \text{Type}$, equipped with two operations*

$$\text{ret} : A \rightarrow M A \quad \text{and} \quad \text{bind} : M A \rightarrow (A \rightarrow M B) \rightarrow M B$$

defined for any types A, B , moreover satisfying the following equations

$$\text{bind}^M (\text{ret}^M a) f = f a \quad \text{bind}^M m \text{ret}^M = m$$

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

for any $a : A, f : A \rightarrow M B, m : M A, g : B \rightarrow M C$.

Intuitively, a computational monad provides a uniform interface MA for computations returning values of type A , for instance state passing functions with result type A for stateful computations. ret^M coerces a value $v : A$ to a trivial computation, for instance seeing v as a stateful computation leaving the state untouched. $\text{bind}^M m f$ sequentially composes the monadic computations $m : MA$ with $f : A \rightarrow MB$, for instance threading through the state.

The generic monad interface $(M, \text{ret}^M, \text{bind}^M)$ is, however, not enough to write programs that exploit the underlying effect. To this end, each computational monad also comes with *operations* for causing and manipulating effects. *Algebraic operations* form an important class of such operations introduced in (Plotkin and Power, 2002). An operation $\text{op} : A \times (M X)^B \rightarrow M X$ is said to be algebraic when the following equation holds

$$\text{bind} (\text{op} (a, f)) g = \text{op} (a, \lambda b. \text{bind} (f b) g)$$

Any such algebraic operation corresponds bijectively to a *generic effect* $\text{gen}_{\text{op}} : A \rightarrow M B$, and we will usually employ this latter presentation, often closer to the programming practice. Slightly abusing the terminology and following Piróg et al. (2018), we will also call “operation” more general functions manipulating the effects provided by a monad, for instance handlers (Plotkin and Pretnar, 2009). In the next subsections, we recall a few examples of computational monads and their operations to illustrate the range of computational effects that monad can account for.

2.1.1 Identity

The simplest monad is the identity monad $\text{Id } A = A$ with $\text{ret}^{\text{Id}} a = a$ and $\text{bind}^{\text{Id}} m f = f m$ satisfying trivially the monad laws. It does not support any operations but it will be useful when discussing *monad transformers* in section 2.2.

2.1.2 Partiality

A simple model for partial computations is given by adding a new element that expresses divergence, i.e. $\text{Div } A = A + \{\perp\}$. Returning a value v is the obvious injection, while sequencing m with f is given by applying f to m if m is a terminating value, or \perp if m was already diverging. A partial computation can diverge with the operation $\Omega : \text{Div } \mathbb{O}$, implemented as $\Omega = \text{inr } \perp$.

In a classical metatheory, $\text{Div } A$ is the free ω -cpo on A , so by standard domain theoretic results (Amadio and Curien (1998)), there is a fixpoint operator on $\text{Div } A$ ¹. However in a constructive metatheory, e.g., Coq, this simple model is too limited to implement a useful fixpoint

¹at least for ω -continuous functions, as provided by Kleene fixpoint

operation. Various more sophisticated approach can provide solutions to that problem. Delbianco and Nanevski's (2013) use complete lattices such as \mathbb{P} and Knaster-Tarski fixpoints.

Altenkirch et al. (2017) directly define the free ω -cpo on a type using quotient-inductive-inductive types to describe the standard construction of completing a type with a bottom element, limits of ω -chains and quotienting by the equivalence relation induced by the natural preorder on these.

A different approach, more in phase with the topics of this manuscript is to describe the syntax of programs with recursion. (McBride, 2015) describes a free monad (see subsection 2.1.7) with one operation `call` playing the role of a recursive call. Given a complete recursive definition, one can then handle these call operations in any monad supporting partiality. We illustrate how to define the skeleton of a function computing the Fibonacci sequence and how such a handling looks like if we were to have a primitive fixpoint operation below

type $GenRec\ A\ B\ X = | \text{Ret} : X \rightarrow GenRec\ A\ B\ X | \text{Call} : a:A \rightarrow (B\ a \rightarrow GenRec\ A\ B\ X) \rightarrow GenRec\ A\ B\ X$

let $fib\ (n:\mathbb{N}) : GenRec\ \mathbb{N}\ (\lambda _ . \mathbb{N})\ \mathbb{N} =$
 if $n \leq 1$ **then** 1 **else** $\text{Call}\ (n-1)\ (\lambda\ r_1 . \text{Call}\ (n-2)\ (\lambda\ r_2 . \text{Ret}\ (r_1 + r_2)))$

let rec $fixGenRec_0\ (f : (a:A) \rightarrow GenRec\ A\ B\ (B\ a))\ (m : GenRec\ A\ B\ X) : X =$
 match m **with**
 | $\text{Ret}\ x \rightarrow x$
 | $\text{Call}\ a\ k \rightarrow fixGenRec_0\ f\ (k\ (fixGenRec_0\ f\ (f\ a)))$

let $fixGenRec\ (f : (a:A) \rightarrow GenRec\ A\ B\ (B\ a))\ (a:A) : B\ a = fixGenRec_0\ f\ (f\ a)$

In a language without arbitrary fixpoints, for instance in Coq, we will instead use fixpoints provided by a suitable monad as above.

2.1.3 Exceptions

A computation that can potentially throw exceptions of type E can be represented by the monad $Exc\ A = A + E$. Returning a value v is the obvious left injection, while sequencing m with f is given by applying f to v if $m = \text{Inl}\ v$, or $\text{Inr}\ e$ if $m = \text{Inr}\ e$, i.e., when m raised an exception.

let $\text{ret}^{Exc}\ (v:A) : Exc\ A = \text{Inl}\ v$

let $\text{bind}^{Exc}\ (m : Exc\ A)\ (f : A \rightarrow Exc\ B) : Exc\ B =$
 match m **with**
 | $\text{Inl}\ v \rightarrow f\ v$
 | $\text{Inr}\ e \rightarrow \text{Inr}\ e$

The operation $\text{throw} : E \rightarrow Exc\ 0$ is defined by right injection of E into $Exc\ 0 = 0 + E$. Such an exception can be caught by the handler `catch`

let $\text{catch}\ (m : Exc\ A)\ (exc : E \rightarrow Exc\ A) : Exc\ A =$
 match m **with**
 | $\text{Inl}\ v \rightarrow \text{Inl}\ v$
 | $\text{Inr}\ e \rightarrow exc\ e$

When we take $E = \mathbb{1}$, exceptions coincide with the simple model of partiality, the monad $Div\ A = A + \mathbb{1}$.

2.1.4 State

A stateful computation can be modeled as a state-passing function, i.e., $St\ A = S \rightarrow A \times S$, where S is the type of the state. Returning a value v is the function $\lambda s. (v, s)$ that produces the value v and the unmodified state, whereas binding m to f is obtained by threading through the state,

i.e. $\lambda s. \text{let } (v, s') = m \text{ in } f v s'$. The state monad comes with operations $\text{get} : \text{St } S = \lambda s. (s, s)$ to retrieve the state, and $\text{put} : S \rightarrow \text{St } \mathbb{1} = \lambda s. \lambda s'. (*, s)$ to overwrite it.

This basic account of stateful computations can be refined by employing a structured state, for instance a store $S = \text{Loc} \rightarrow \text{Val}$ where Loc is a set of locations and Val is the type of (ground) value that can be written to the store. In that case, we can also refine the operations get and put , parametrizing them by accessed location in the store:

```
let getL(l : Loc) : St Val = λs. (s l, s)
let putL(l : Loc) (v : Val) : St 1 = λs. ((), λl'. if l' = l then v else s l')
```

We will see in [section 5.1](#), this idea is the basis of stateful verification in F^* , however with a much more complex memory model.

2.1.5 Nondeterminism

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

2.1.6 Interactive input-output (IO)

An interactive computation with input type I and output type O can be represented by the inductively defined monad

```
type IO A = | Ret : A → IO A | Input : (I → IO A) → IO A | Output : O → IO A → IO A
```

which describes three possible kinds of computations: either return a value (**Ret**), expect to receive an input and then continue (**Input**), or output and continue (**Output**). The monadic function ret^{IO} constructs a unique leaf tree using **Ret** and bind^{IO} does tree grafting. The operations perform input and output, and they are directly captured using the corresponding constructors.

```
let read : IO I = Input (λi. retIO i)          let write (o : O) : IO 1 = Output o (retIO ())
```

2.1.7 Free monads & monads presented by an equational theory

The monads for identity, exception, general recursion *GenRec* and interactive input-output are examples of *free monads*, that is monads inductively generated by a set of algebraic operations. Given any signature (S, P) consisting of a set S of operations and a function $P : S \rightarrow \text{Type}$ assigning to each operation its arity, we can construct the following monad consisting of terms on the signature (S, P) :

```
type Free S P X = | Ret : X → Free S P X | Op : (s:S) → (P s → Free S P X) → Free S P X
```

```
let retFree (x:X) : Free S P X = Ret x
```

```
let rec bindFree (m:Free S P X) (f: X → Free S P Y) : Free S P Y =
  match m with
  | Ret x → f x
  | Op s k → Op s (λ r. bindFree (k r) f)
```

with an associated generic effect $\text{let op } (s:S) : \text{Free S P } (P s) = \text{Op } s (\lambda r. \text{Ret } r)$.

More generally, we could consider an equational theory (S, P, E) , that is a signature (S, P) equipped with a set of equations E between terms on the signature (S, P) – formally a set E of pairs of terms. The monad associated to such a theory is the quotient of terms modulo the equivalence relation induced by the congruence closure of E . All the previous examples of monads are such *presented monads*. However, in absence of arbitrary effective quotients which may require

instances of axiom of choice (Blass, 1983) or quotient inductive types (QITs) (Altenkirch and Kaposi, 2016), we will refrain from using these in a constructive setting and prefer the previous per-effect presentation of the monads.

2.1.8 Probabilities

A probabilistic computation is a sub-probability distribution on possible outcomes, i.e., for a countable type A , $\text{Prob } A$ represents functions $f : A \rightarrow [0; 1]$ such that $\sum_{a \in A} f a \leq 1$. Restricting our attention to countable discrete probabilities, there is a monad structure on Prob known as the *Giry monad* (Giry, 1982). Returning a value v is the Dirac distribution at v , that is the distribution assigning weight 1 to v and 0 to any other value. Binding a distribution $m : \text{Prob } A$ to a function $f : A \rightarrow \text{Prob } B$ amounts to computing the distribution on B given by $\lambda y. \sum_{x \in \text{supp}(m)} f x y$. We can consider various basic distributions on countable spaces as operations, for instance $\text{flip} : [0; 1] \rightarrow \text{Prob } \mathbb{B}$ provides a Bernoulli distribution on booleans (with parameter given by the argument) and $\text{unif} : n : \mathbb{N} \rightarrow \text{Prob } (\text{fin } n)$ provides a uniform distribution on the finite type $\text{fin } n$ with n elements.

2.1.9 Continuations

Continuation passing style programming is captured by the continuation monad

let $\text{Cont } R A = (A \rightarrow R) \rightarrow R$

let $\text{retCont } (a:A) : \text{Cont } R A = \lambda k. k \ a$

let $\text{bindCont } (m:\text{Cont } R A) (f: A \rightarrow \text{Cont } R B) : \text{Cont } R B = \lambda k. m (\lambda a. f a k)$

Returning a value $v : A$ is just evaluating the continuation to this value, while sequencing two continuation-passing computations $m : \text{Cont}_R A$ and $f : A \rightarrow \text{Cont}_R B$ is a matter of building a suitable continuation for m with f . The continuation monad hosts an operation `call_cc`:

let $\text{call_cc } (f: (A \rightarrow \text{Cont } R R) \rightarrow \text{Cont } R R) : \text{Cont } R A =$
 $\lambda k. f (\lambda a. \text{retCont } (k a)) (\lambda r. r)$

The continuation monad is a canonical example of a monad without rank, meaning that it is not presentable by a (small) theory. Intuitively, this is due to the fact that we would need operations of arbitrary arity to present the continuation monad.

2.2 Taming the monad zoo: a first glance at monad transformers

The previous section presented a variety of computational monads covering most of the effect spectrum. However programs usually use more than a single effect at a time. An important question thus is how to combine these effects and the corresponding monads.

This question is actually harder than one could expect at first. Indeed, given two monads M_1, M_2 there might be one way to compose them, or multiple ways to do so, or even none. The various ways to compose M_1 and M_2 are encoded by *distributive laws* (Beck, 1969). Finding distributive laws for every pair of monads one wants to compose in a program is not only tedious but hardly practical. Two approaches try to bypass this problem and recover some amount of modularity.

One canonical way to compose monads can be obtained by restricting our attention to monads arising from *algebraic effects*, that is effects described only in terms of algebraic operations and equations between these operations (Hyland et al., 2006).

Otherwise, instead of insisting on composing two monads, we can consider *monad transformers* extending a base monad with new operations. Concretely, monad transformer \mathcal{T} maps a monad M to a monad $\mathcal{T}M$ and provides for any type A a coercion $\text{lift}_{\mathcal{T}} : M A \rightarrow \mathcal{T} M A$

materializing how $\mathcal{T}M$ extends M . Since we need to consider monads not arising from algebraic effects, in this manuscript we take in this second approach. The definition and construction of monad transformers is studied in depth in [chapter 4](#). In this section, we informally present examples of such transformers. As a particular case, applying a monad transformer \mathcal{T} with the identity monad Id provides a plain monad, often corresponding to one described in the previous section.

2.2.1 State

The state transformer StT on a fixed type of states S extends a monad M using state passing $\text{StT } M A = S \rightarrow M(A \times S)$ to provide operations $\text{get} : \mathbb{1} \rightarrow \text{StT } M S$ and $\text{put} : S \rightarrow \text{StT } M \mathbb{1}$. The lifting operation is defined by

let $\text{liftStT}(m : M A) : \text{StT } M A = \lambda s. \text{bind}^M m (\lambda a. \text{ret}^M(a, s))$

2.2.2 Exceptions

The exception transformer ExcT adds a set of exceptions E to the possible results of a monad M , that is $\text{ExcT } M A = M(A + E)$, providing an operation $\text{throw} : E \rightarrow \text{ExcT } M \mathbb{0}$. Lifting a computation from M to $\text{ExcT } M$ is defined as

let $\text{liftExcT}(m : M A) : \text{ExcT } M A = \text{bind}^M m (\lambda a. \text{ret}^M(\text{Inl } a))$

2.2.3 Reader, writer and other update transformers

If we want to extend a computation with a read-only environment S , the reader transformer $\text{RdT } M A = S \rightarrow M A$ fits our needs. Dually, if we only want to log informations, it's the writer monad $\text{WrT } M A = M(A \times \text{list } O)$ that we should use. As explained by [Ahman and Uustalu \(2013\)](#) for the case of plain monads, the two transformers are instances of a family of monad transformers called *update transformers* parametrized by a pair of a type S of states and a monoid $(\mathcal{O}, *, e)$ of updates acting on the states $\triangleright : \mathcal{O} \times S \rightarrow S$:

$$\text{UpdT } M A = S \rightarrow M(A \times \mathcal{O})$$

The monad structure on $\text{UpdT } M$ and the lift from M are given by

let $\text{retUpdT}(a : A) : \text{UpdT } M A = \lambda s. \text{ret}^M(a, e)$

let $\text{bindUpdT}(m : \text{UpdT } M A) (f : A \rightarrow \text{UpdT } M B) : \text{UpdT } M B =$
 $\lambda s_0. \text{bind}^M (m s_0) (\lambda (a, o_1). \text{bind}^M (f a (o_1 \triangleright s_0)) (\lambda (b, o_2). \text{ret}^M(b, o_2 * o_1)))$

let $\text{liftUpdT}(m : M A) : \text{UpdT } M A =$
 $\lambda s. \text{bind}^M m (\lambda a. \text{ret}^M(a, e))$

The reader transformer arises as the update monad associated to the pair $(S, \mathbb{1})$, where the trivial monoid $\mathbb{1}$ acts on S by identity. The writer transformer arises as the pair $(\mathbb{1}, \text{list } O)$ where the free monoid $\text{list } O$ acts trivially on the unit state.

2.2.4 Monotonic state and dependent update transformers

Moving to a dependently typed example, the monotonic state transformer MonStT is a refinement of the state transformer where the state updates are restricted along a fixed preorder on states $\preceq \subset S \times S$:

$$\text{MonStT } M A = (s_0 : S) \rightarrow M(A \times \{ s_1 : S \mid s_0 \preceq s_1 \})$$

As advocated by [Ahman et al. \(2018\)](#), only extending the computational monad with monotonic manipulations of stateful could enable monotonic reasoning, a cheap but efficient method to prove various stateful properties.

The monotonic state transformer can be seen as an instance of a dependent update transformer, a generalization of update transformers where the parametrizing monoid \mathcal{O} is replaced by a dependent family $\mathcal{P} : S \rightarrow \text{Type}$ indexed by the states and an adequate notion of action, forming together a *directed container* (S, \mathcal{P}) ([Ahman and Uustalu, 2013](#)). For a state $s : S$, the type $\mathcal{P} s$ describes the possible way to act in state s . The data of a directed container (S, \mathcal{P}) actually correspond to a category where S is the object set and $\mathcal{P} s$ are the morphism with domain s . The dependent update monad transformer maps a monad M to a monad on the carrier

$$\text{DUdTM } M A = (s : S) \rightarrow M(A \times \mathcal{P} s).$$

The case of monotonic state transformer is recovered by a directed container structure on the pair $(S, \lambda s_0. \{ s_1 : S \mid s_0 \preceq s_1 \})$.

2.3 Specifications from monads

As explained in [section 1.1](#), the realization that predicate transformers form monads ([Ahman et al., 2017](#); [Jacobs, 2014, 2015](#); [Swamy et al., 2013, 2016](#)) is the starting point to provide a uniform notion of specifications. This is true not only for weakest precondition transformers, but also for strongest postconditions, and pairs of pre- and postconditions as explained in details in the following subsections. We call collectively this class of monads *specification monads*. Intuitively, elements of a specification monad can be used to specify properties of some computation, e.g., W^{Pure} can specify pure or nondeterministic computations, and W^{St} can specify stateful computations.

What is a specification monad ? A conceptual definition will be given in [Def. 3.5.2](#), but for the time being we will be using the following elementary definition.

Definition 2.3.1. A specification monad is a monad W such that

- ▷ $W A$ is equipped with a preorder $\leq^{W A}$ for each type A , and
- ▷ bind^W is monotonic in both arguments:

$$\begin{aligned} \forall (w_1 \leq^{W A} w'_1). \forall (w_2 w'_2 : A \rightarrow W B). \\ (\forall x : A. w_2 x \leq^{W B} w'_2 x) \quad \Rightarrow \quad \text{bind}^W w_1 w_2 \leq^{W B} \text{bind}^W w'_1 w'_2 \end{aligned}$$

This order allows specifications to be compared as being either more or less precise. For example, for the specification monads W^{Pure} and W^{St} , the ordering is given by

$$\begin{aligned} w_1 \leq w_2 : W^{\text{Pure}} A & \Leftrightarrow \quad \forall (p : A \rightarrow \mathbb{P}). w_2 p \Rightarrow w_1 p \\ w_1 \leq w_2 : W^{\text{St}} A & \Leftrightarrow \quad \forall (p : A \times S \rightarrow \mathbb{P})(s : S). w_2 p s \Rightarrow w_1 p s \end{aligned}$$

For W^{Pure} and W^{St} to form ordered monads, it turns out that we need to restrict our attention to *monotonic* predicate transformers, i.e., those mapping (pointwise) stronger postconditions to stronger preconditions. This technical condition, quite natural from the point of view of verification, will be assumed implicitly for all the predicate transformers. We consider several basic specification monads, whose relationship is summarized by [Figure 2.1](#).

2.3.1 Predicate monad

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

2.3.2 Pre-/postcondition monad

One more expressive specification monad is the monad of pre- and postconditions

$$\mathcal{P}\text{rePost } A = \mathbb{P} \times (A \rightarrow \mathbb{P}),$$

bundling a precondition together with a postcondition. Here the behavior of returning a value $v : A$ is specified by requiring a trivial precondition and ensuring as above a singleton postcondition: $\text{ret}^{\mathcal{P}\text{rePost}} v = (\top, \lambda a. a = v) : \mathcal{P}\text{rePost } A$. And, given $p = (pre, post) : \mathcal{P}\text{rePost } A$ and a function $f = \lambda a. (pre' a, post' a) : A \rightarrow \mathcal{P}\text{rePost } B$, the sequential composition of two computations is naturally specified by defining

$$\text{bind}^{\mathcal{P}\text{rePost}} p f = ((pre \wedge \forall a. post a \implies pre' a) , \lambda b. \exists a. post a \wedge post' a b) : \mathcal{P}\text{rePost } B$$

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

We formally show that this specification monad is more expressive than the predicate monad above: Any predicate $p : \mathcal{P}\text{red } A$ can be coerced to $(\top, p) : \mathcal{P}\text{rePost } A$, and in the other direction, any pair $(pre, post) : \mathcal{P}\text{rePost } A$ can be approximated by the predicate $post$, giving rise to a Galois connection, as illustrated in [Figure 2.1](#). While the monad $\mathcal{P}\text{rePost}$ is intuitive for humans, generating efficient verification conditions is generally easier with predicate transformers ([Leino, 2005](#)).

2.3.3 Forward predicate transformer monad

The predicate monad $\mathcal{P}\text{red}$ can be extended in an alternative way. Instead of fixing a precondition as in $\mathcal{P}\text{rePost}$, a specification can be a function from preconditions to postconditions, for instance producing the strongest postcondition of computation for any precondition $pre : \mathbb{P}$ given as argument. Intuitively, such a forward predicate transformer on A should have type $\mathbb{P} \rightarrow (A \rightarrow \mathbb{P})$. However, to obtain a monad (i.e., satisfying the expected laws), we have to consider the smaller type

$$\mathcal{S}\text{Post } A = (pre : \mathbb{P}) \xrightarrow{\text{mon}} (A \rightarrow \mathbb{P}_{/pre})$$

of predicate transformers that are monotonic with respect to pre , where $\mathbb{P}_{/pre}$ is the subtype of propositions implying pre . Returning a value $v : A$ is specified by the predicate transformer

$\text{ret}^{\mathcal{SPost}} v = \lambda pre a. pre \wedge a = v$, and the sequential composition of two computations is specified as the predicate transformer $\text{bind}^{\mathcal{SPost}} m f = \lambda pre b. \exists a. f a (m pre a) b$, for $m : \mathcal{SPost} A$ and $f : A \rightarrow \mathcal{SPost} B$.

Any specification $post : \mathcal{Pred} A$ gives rise to a forward predicate transformer

$$\text{spostOfPred } post = \lambda(pre : \mathbb{P}) (a : A). pre \wedge post a \quad : \quad \mathcal{SPost} A$$

and conversely a forward predicate transformer $sp : \mathcal{SPost} A$ induces a canonical postcondition

$$\text{predOfSpost } sp = sp \top \quad : \quad \mathcal{Pred} A$$

If forward predicate transformer in \mathcal{SPost} could seem more expressive than \mathcal{Pred} , it turns out that the two functions spostOfPred and predOfSpost are inverse of each others.

2.3.4 Backward predicate transformer monad

As explained in [section 1.1](#), backward predicate transformers can be described using the continuation monad with propositions \mathbb{P} as the answer type, namely, $\text{Cont}_{\mathbb{P}} A = (A \rightarrow \mathbb{P}) \rightarrow \mathbb{P}$. Elements $w : \text{Cont}_{\mathbb{P}} A$ are predicate transformers mapping a postcondition $post : A \rightarrow \mathbb{P}$ to a precondition $w post : \mathbb{P}$, for instance the weakest precondition of the computation. Pointwise implication is a natural order on $\text{Cont}_{\mathbb{P}} A$:

$$w_1 \leq w_2 : \text{Cont}_{\mathbb{P}} A \quad \Leftrightarrow \quad \forall(p : A \rightarrow \mathbb{P}). w_2 p \Rightarrow w_1 p$$

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

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

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

This specification monad is more expressive than the pre-/postcondition one above. Formally, a pair $(pre, post) : \text{PrePost } A$ can be mapped to the monotonic predicate transformer

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

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

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

These two mappings define a Galois connection, as illustrated in [Figure 2.1](#). Further, this Galois connection exhibits $\text{PrePost } A$ as the submonad of $\text{W}^{\text{Pure}} A$ of *conjunctive* predicate transformers, i.e., predicate transformers w commuting with non-empty conjunctions/intersections.

2.3.5 A specification monad of relations between pre- and postconditions

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

$$SPost \cong Pred \xrightarrow{\quad} PrePost \xrightarrow{\quad} W^{Pure} \xrightarrow{\quad} RelPrePost$$

Each pair of parallel arrows forms a Galois connection.

Figure 2.1: Relationships between basic specification monads

2.3.6 Specification monads from transformers

Once we have a few basic specification monads as the one described above, a powerful way to construct specification monads is to apply monad transformers to existing specification monads. For instance, applying $ExcT M A = M(A + E)$ to W^{Pure} we get

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

W^{Exc} is a natural specification monad for programs that can throw exceptions, transporting a normal postcondition in $A \rightarrow \mathbb{P}$ and an exceptional postcondition in $E \rightarrow \mathbb{P}$ to a precondition in \mathbb{P} .

Besides accounting for exceptional termination, varying the monad transformer extend specifications to have access to ghost state or to provide information about footprints. An important point is that monad transformer provides an important modularity property: when specifying code, we can use as little facilities as needed and consequently produce less clutter in verification conditions. Further specification monads using this idea will be introduced along with the examples in [section 2.4](#).

Since specification monads also carry a preorder, we need the monad transformers to preserve this ordered structure. We will see in [chapter 4](#) that it is the case of all examples of monad transformers of [section 2.2](#).

2.3.7 Quantitative variations

Nothing prevents à priori to define specifications monads based on preorders different from propositions. For instance, the example of the backward predicate transformer monad for instance would have the structure of a specification monad independently of the choice of the ordered return type (\mathcal{R}, \preceq) replacing \mathbb{P} .

Taking \mathcal{R} to be the extended reals $[0; \infty]$, we recover a monad to specify pre-expectations of probabilistic programs ([Audebaud and Paulin-Mohring, 2006](#); [Kaminski et al., 2016](#)).

Another possibility is to take \mathcal{R} to be a set of available resources, for instance natural number to count the number of steps a program could take. This can be refined to positive rational or real numbers, obtaining a specification monad for cost analysis.

We did not pursue much further the analysis of such quantitative variants of specification monads, but expect that a sensible amount of the work developed here could extend to the quantitative setting.

2.4 Effect observations

Now that we have a presentation of specifications as elements of a monad, we need to relate computational monads to such specifications. Since an object relating computations to specifications provides a particular insight on the effects exhibited by the computation, they have been called *effect observations* ([Katsumata, 2014](#)). As explained in [section 1.2](#), a computational monad can have effect observations into multiple specification monads, or multiple effect observations into a single specification monad. Using the exceptions computational monad Exc as running example, we argue that *monad morphisms* provide a natural notion of effect observation in a unary monadic setting, and we provide instances of effect observations supporting this claim. Then, we

revisit the computational monads from [section 2.1](#), and present various natural effect observations for them since there is generally a large variety of options regarding both the specification monads and the effect observations when specifying and verifying monadic programs.

2.4.1 Effect observations are monad morphisms

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

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

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

and second, θ preserves the sequencing of computations:

$$\begin{aligned} \theta^{\text{Exc}}(\text{bind}^{\text{Exc}} (\text{inl } v) f) &= \theta^{\text{Exc}}(fv) = \text{bind}^{W^{\text{Exc}}} (\text{ret}^{W^{\text{Exc}}} v) (\theta^{\text{Exc}} \circ f) \\ &= \text{bind}^{W^{\text{Exc}}} \theta^{\text{Exc}}(\text{inl } v) (\theta^{\text{Exc}} \circ f) \\ \theta^{\text{Exc}}(\text{bind}^{\text{Exc}} (\text{inr } e) f) &= \theta^{\text{Exc}}(\text{inr } e) = \text{bind}^{W^{\text{Exc}}} \theta^{\text{Exc}}(\text{inr } e) (\theta^{\text{Exc}} \circ f) \end{aligned}$$

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

Definition 2.4.1 (Effect observation). *An effect observation θ is a monad morphism from a computational monad M to a specification monad W . More explicitly, it is a family of maps $\theta_A : M A \rightarrow W A$, natural in A and such that for any $v : A$, $m : M A$ and $f : A \rightarrow M B$ the following equations hold:*

$$\theta_A(\text{ret}^M v) = \text{ret}^W v \quad \theta_B(\text{bind}^M m f) = \text{bind}^W (\theta_A m) (\theta_B \circ f)$$

2.4.2 Specification monads are not canonical

When writing programs using the exception monad, we may want to write pure sub-programs that actually do not raise exceptions. In order to make sure that these sub-programs are pure, we could use the previous specification monad and restrict ourselves to postconditions that map exceptions to false (\perp): hence raising an exception would have an unsatisfiable precondition. However, as outlined in [section 1.2](#), a simpler solution is possible. Taking as specification monad W^{Pure} , we can define the following effect observation $\theta^\perp : \text{Exc} \rightarrow W^{\text{Pure}}$ by

$$\theta^\perp(\text{inl } v) = \lambda p. p \, v \quad \theta^\perp(\text{inr } e) = \lambda p. \perp$$

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

2.4.3 Effect observations are not canonical

Looking closely at the effect observation θ^\perp , it is clear that we made a rather arbitrary choice when mapping every exception $\text{inr } e$ to \perp . Mapping $\text{inr } e$ to true (\top) instead also gives us an effect observation, $\theta^\top : \text{Exc} \rightarrow W^{\text{Pure}}$. This effect observation assigns a trivial precondition to the throw operation, providing a *partial correctness* interpretation: given a program $m : \text{Exc } A$ and a postcondition $p : A \rightarrow \mathbb{P}$, if $\theta^\top(m)(p)$ is satisfiable and m evaluates to $\text{inl } v$ then $p \, v$ holds;

but m may also raise any exception instead. Thus, $\theta^\perp, \theta^\top : \text{Exc} \rightarrow W^{\text{Pure}}$ are two natural effect observations into the *same* specification monad. Even more generally, we can vary the choice for each exception; in fact, effect observations $\theta : \text{Exc} \rightarrow W^{\text{Pure}}$ are in one-to-one correspondence with maps $E \rightarrow \mathbb{P}$ (see [subsection 2.4.8](#) for a general account of this correspondence).

2.4.4 Effect observations from monad transformers

Even though there is, in general, no canonical effect observation for a computational monad, we can build an effect observation in the particular case of a monad of the shape $\mathcal{T}(\text{Id})$, i.e., a monad obtained by the application of a monad transformer to the identity monad. In that setting, fixing any we can build a canonical specification monad, namely $\mathcal{T}(W^{\text{Pure}})$, and a canonical effect observation into it. The effect observation is obtained simply by lifting the $\text{ret}^{W^{\text{Pure}}} : \text{Id} \rightarrow W^{\text{Pure}}$ function through the \mathcal{T} transformer. For instance, for the exception monad $\text{Exc} = \text{ExcT}(\text{Id})$ and the specification monad $W^{\text{Exc}} = \text{ExcT}(W^{\text{Pure}})$, the effect observation θ^{Exc} arises as simply $\theta^{\text{Exc}} = \text{ExcT}(\text{ret}^{W^{\text{Pure}}}) = \lambda m p. p m$. More generally, for any monad transformer \mathcal{T} (e.g. StT , ExcT , $\text{StT} \circ \text{ExcT}$, $\text{ExcT} \circ \text{StT}$) and any specification monad W (so not just W^{Pure} , but also e.g., any basic specification monad from [section 2.3](#)) we have a monad morphism

$$\theta^{\mathcal{T}} : \mathcal{T}(\text{Id}) \xrightarrow{\mathcal{T}(\text{ret}^W)} \mathcal{T}(W)$$

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

2.4.5 Effect observations for free monads

In order to give an effect observation θ^{Free} from a free monad induced by a signature (S, P) ([subsection 2.1.7](#)) to a specification monad W , it is enough to provide for each operations $s : S$ a specification $w_{\text{op}}(s) : W(P s)$.

```
let rec  $\theta^{\text{Free}}$  ( $w_{\text{op}} : (s:S) \rightarrow W(P s)$ ) ( $m : \text{Free } P S A$ ) :  $W A =$ 
  match  $m$  with
  | Ret  $a \rightarrow \text{ret } W a$ 
  | Op  $s k \rightarrow \text{bind } W(w_{\text{op}} s) (\lambda ps. \theta^{\text{Free}}(k ps))$ 
```

Conversely, any effect observation θ^{Free} induces a specification for each operations

$$w_{\text{op}} = \theta^{\text{Free}} \circ \text{gen}_{\text{op}} : (s:S) \rightarrow W(P s).$$

This correspondence is bijective and characteristic of free monads.

2.4.6 Observing nondeterminism

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

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

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

2.4.7 Observing Interactive Input-Output

Let us now consider programs in the IO monad (section 2.1). We want to define an effect observation $\theta : \text{IO} \rightarrow \text{W}$, for some specification monad W to be determined. A first thing to note is that since no equations constrain the `read` and `write` operations, IO is a free monad, we can specify their interpretations $\theta(\text{read}) : \text{W } I$ and $\forall(o : O). \theta(\text{write } o) : \text{W } \mathbb{1}$ separately from each other.

Simple effect observations for IO can already be provided using the specification monad W^{Pure} . The interpretation of the `write` operation in this simple case needs to provide a result in \mathbb{P} from an output element $o : O$ and a postcondition $p : \mathbb{1} \rightarrow \mathbb{P}$. Besides returning a constant proposition (like for $\theta^\perp, \theta^\top$ in subsection 2.4.2), a reasonable interpretation is to forget the `write` operation and return $p *$ (where $*$ is the unit value). For the definition of $\theta(\text{read}) : (I \rightarrow \mathbb{P}) \rightarrow \mathbb{P}$, we are given a postcondition $\text{post} : I \rightarrow \mathbb{P}$ on the possible inputs and we need to build a proposition. Two canonical solutions are to use either the universal quantification $\forall(i : I). \text{post } i$, requiring that the postcondition is valid for the continuation of the program for any possible input; or the existential quantification $\exists(i : I). \text{post } i$, meaning that there exists some input such that the program’s continuation satisfies the postcondition, analogously to the two modalities of evaluation logic (Moggi, 1995; Pitts, 1991).

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

$$\begin{aligned} \theta^{\text{HistST}}(\text{write } o) &= \lambda(p : \mathbb{1} \times \text{list } \mathcal{E} \rightarrow \mathbb{P}) (\log : \text{list } \mathcal{E}). p(*, (\text{Out } o) :: \log) &: \text{W}^{\text{HistST}}(\mathbb{1}) \\ \theta^{\text{HistST}}(\text{read}) &= \lambda(p : I \times \text{list } \mathcal{E} \rightarrow \mathbb{P}) (\log : \text{list } \mathcal{E}). \forall i. p(i, (\text{In } i) :: \log) &: \text{W}^{\text{HistST}}(I) \end{aligned}$$

This specification monad is however somewhat inconvenient in that postconditions are written over the *global* history of events, instead of over the events of the expression in question. Further, one can write specifications that “shrink” the global history of events, such as $\lambda p \log. p \langle *, [] \rangle$, which *no* expression satisfies. For these reasons, we introduce an *update monad* (Ahman and Uustalu, 2013) variant of W^{HistST} , written W^{Hist} , which provides a more concise way to describe the events. In particular, in W^{Hist} the postcondition specifies only the events produced by the expression, while the precondition is still free to specify any previously-produced events, allowing us to define:

$$\begin{aligned} \theta^{\text{Hist}}(\text{write } o) &= \lambda(p : \mathbb{1} \times \text{list } \mathcal{E} \rightarrow \mathbb{P}) (\log : \text{list } \mathcal{E}). p \langle *, [\text{Out } o] \rangle &: \text{W}^{\text{Hist}}(\mathbb{1}) \\ \theta^{\text{Hist}}(\text{read}) &= \lambda(p : I \times \text{list } \mathcal{E} \rightarrow \mathbb{P}) (\log : \text{list } \mathcal{E}). \forall i. p \langle i, [\text{In } i] \rangle &: \text{W}^{\text{Hist}}(I) \end{aligned}$$

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

$$\begin{aligned} \text{bind}^{\text{W}^{\text{HistST}}} w f &= \lambda p \log. w (\lambda (x, \log') . f x p \log') \log \\ \text{bind}^{\text{W}^{\text{Hist}}} w f &= \lambda p \log. w (\lambda (x, \log') . f x (\lambda (y, \log'') . p (y, \log' ++ \log'')) (\log ++ \log')) \log \end{aligned}$$

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

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

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

$$\begin{aligned}\theta^{\text{Fr}}(\text{write } o) &= \lambda(p : \mathbb{1} \times \text{list } \mathcal{E} \rightarrow \mathbb{P}). p(*, [\text{Out } o]) &: W^{\text{Fr}}(\mathbb{1}) \\ \theta^{\text{Fr}}(\text{read}) &= \lambda(p : I \times \text{list } \mathcal{E} \rightarrow \mathbb{P}). \forall i. p(i, [\text{In } i]) &: W^{\text{Fr}}(I)\end{aligned}$$

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

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

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

2.4.8 Effect Observations from Monad Algebras

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

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

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

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

2.5 Conclusion & Related work

Following a well established tradition in functional programming languages (Benton et al., 2000; Moggi, 1989; Wadler, 1990), we presented a variety of monads encapsulating computational effects such as state, nondeterminism and interactive input output, and explained how an important subset of these arise from monad transformers.

The notion of specification monad is inspired by a line of categorical work on weakest precondition. Jacobs (2014, 2015, 2017) studies adjunctions between state transformers and predicate transformers, obtaining a class of specification monads from the state monad transformer and an abstract notion of logical structures. He gives abstract conditions for the existence of such specification monads. Hasuo (2015) builds on the state-predicate adjunction of Jacobs to provide algebra-based effect observations (in the style of subsection 2.4.8) for various computation and specification monads. Working inside type theory, our work focus on concrete recipes for building specification monads useful for practical verification.

Effect observations as monad morphisms were introduced by Katsumata (2014) in his study of graded monads to give semantics to type-and-effect systems. For each of these computational monads, we proposed effect observations to multiple specification monads providing multiple options in order to verify programs using these effects. The actual choice of the effect observation to use depends on various trade-offs between expressivity of the specification, difficulty of the properties to verify (e.g., partial or total termination), modularity with respect to context. We argue that the possibility to adapt to various context and at minor implementation cost thanks to the decorrelation between the computational monad, the specification and the effect observation is a key asset of this framework, that should be developed further in a practical implementation.

We now review further related work that was not presented yet.

Alternative representation of effects Levy (2004) refines the approach to effects advocated by Moggi (1989) replacing a computational monad M with an adjunction $F \dashv U$. This allow a finer treatment of the order of evaluation, admitting a treatment of side-effects in both call-by-value and call-by-name settings together with a well-behaved equational theory. To our knowledge, program logics for CBPV are yet to be defined and studied.

Local state & monads on resource indexed families An important variation on stateful computations not presented in this chapter is the possibility of allocating and deallocating chunks of memory. The local state monad introduced in (Plotkin and Power, 2002) provides such capabilities at the cost of more complex state-indexed types. Instead of considering monads on plain types, we could also consider monads on families of types indexed by some notion of resource³. This leads to monads tracking not only stateful computations but also allocations and deallocations (Maillard and Melliès, 2015; Melliès, 2014; Power, 2006; Staton, 2010), manipulating addresses in a heap (Kammar et al., 2017) or even a set of qbits (Staton, 2015).

Combining theories Instead of accumulating monad transformers on top of a basic monad, an important body of work focus on the direct combination of effects, in particular for those presented by an equational theory (Hyland et al., 2006). The combination of these algebraic effects with continuations is studied in (Hyland et al., 2007), and provides in particular a negative results about the combination of interactive input-output and continuations (in the category of sets) that apply as well to Coq.

Predicate transformer semantics Katsumata (2013) gives a semantic account of Lindley and Stark (2005)’s $\top\top$ -lifting, a generic way of lifting relations on values to relations on monadic

³It would corresponds to changing the underlying category to be some category of presheaves, which might be achievable inside type theory using the work of (Boulier et al., 2017; Jaber et al., 2012, 2016)

computations, parameterized by a basic notion of relatedness at a fixed type. Monad morphisms $M A \rightarrow ((A \rightarrow \mathbb{P}) \rightarrow \mathbb{P})$, i.e. effect observations from M to the backward predicate transformer specification monad W^{Pure} , are also unary relational liftings $(A \rightarrow \mathbb{P}) \rightarrow (M A \rightarrow \mathbb{P})$, and could be generated by $\top\top$ -lifting. Further, binary relational liftings could be used to generate monadic relations that yield Dijkstra monads by the construction in [chapter 5](#). In both cases, what is specifiable about the underlying computation would be controlled by the chosen basic notion of relatedness.

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

First-order approach to verification with generic side-effects [Rauch et al. \(2016\)](#) provide a generic verification framework for *first-order* monadic programs. Their work is quite different from ours, even beyond the restriction to first-order programs, since their specifications are “innocent” effectful programs, which can observe the computational context (e.g., state), but not change it. This introduces a tight coupling between computations and specifications, while we provide much greater flexibility through effect observations.

The FreeSpec framework ([Letan et al., 2018](#)) uses algebraic effects and handlers to define in Coq a set of components interacting through interfaces. The specification are given pairs of pre-/postconditions and attached to each components.

Logical approach to effects Generic reasoning about computational monads dates back to [Moggi’s \(1989\)](#) seminal work, who proposes an embedding of his computational metalanguage into higher-order logic. Pitts & Moggi’s evaluation logic ([Moggi, 1995](#); [Pitts, 1991](#)) later introduces modalities to reason about the result(s) of computations, but not about the computational context. [Plotkin and Pretnar \(2008\)](#) propose a generic logic for algebraic effects that encompasses Moggi’s computational λ -calculus, evaluation logic, and Hennesy-Milner logic, but does not extend to Hoare-style reasoning for state. [Simpson and Voorneveld \(2018\)](#) and [Matache and Staton \(2019\)](#) explore logics for algebraic effects by specifying the effectful behaviour of algebraic operations using a collection of effect-specific modalities instead of equations. Their modalities are closely related to how we derive effect observations $\theta : M \rightarrow W^{\text{Pure}}$ and thus program specifications from M -algebras on \mathbb{P} in [subsection 2.4.8](#), as intuitively the conditions they impose on their modalities ensure that these can be collectively treated as an M -algebra on \mathbb{P} . In recent work concurrent to ours, [Voorneveld \(2019\)](#) studies a logic based on quantitative modalities by considering truth objects richer than \mathbb{P} , including $S \rightarrow \mathbb{P}$ for stateful and $[0, 1]$ for probabilistic computation.

The notion of specification monad we use is quite simple, counting the bare minimum to start talking about specification. However it is lacking for actually defining a logic. This choice was voluntary in order not to restrict the applicability of the framework, in particular for quantitative reasoning as would be needed when reasoning about costs or probabilities. In practice, most of the examples we presented support a rich logic and we would like to reflect this in the definition of more restricted classes of specification monads from which we could define a logic.

Reasoning directly 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 effectful programs, without resorting to a monadic encoding.

Chapter 3

Abstracted away

『そしてそこには出口がない。出口を見つけられる可能性すらない。君は時の迷宮の中に迷いこんでしまっている。なによりもいちばん大きな問題は、そこから出ていきたいという気持ちを君がまったく抱けないでいることだ。そうだね?』

村上 春樹, 海辺のカフカ, 2006

Computational monads are the key algebraic structure to obtain compositionality of sequential programs even in an effectful setting. A conceptual understanding of the tools enabling verification of such program should make use of this monadic structure, as for instance specification monads. However, plain monads do not fully account for these objects that we use to study monadic program verification. Beside being monads, specification monads comes with a preorder structure and various axioms ensuring the well-behavedness of these preorders with respect to the monadic operations. We would like to obtain these conditions as an instance of a more general notion of monad. We hope by pursuing this goal that a general approach will lead to simpler proofs, not cluttered by the details of the objects we are manipulating.

In this chapter, we introduce a few abstract categorical constructions generalizing that of plain monads and used extensively in the following chapters. Our starting point is the formal theory of monads, following [Street \(1972\)](#), that provides a general formulation of monads and associated concepts in an arbitrary 2-category. In particular the theory applies to enriched settings and, keeping in mind specification monads, we are foremost interested in the *Pos*-enriched case.

The monad-like structure arising in the context of monadic program verification however are often not endofunctors: we present the theory of relative monads ([Altenkirch et al., 2015](#)) that was developed for that purpose. Motivated by enriched variants of relative monads, for instance on preorders, we sketch the foundations of a formal theory of relative monads. Framed bicategories ([Shulman, 2008](#)) is a natural setting to pursue such a generalization. We present framed bicategories, introduce relative monads in those, and define notions of algebras. We close the chapter by showing that to some extent the formal theory of relative monads we present here naturally extends that of monads.

3.1 Elements of the formal theory of monads

The notion of monad admits a general definition in an arbitrary 2-category or even a bicategory due to [Bénabou \(1967\)](#). We begin this chapter recalling briefly the notion of 2-/bi-category, before presenting a few elements of the formal theory of monads as developed in ([Kelly and Street, 1974](#); [Lack and Street, 2002](#); [Street, 1972](#)). A far more complete reference on the topics touched here is ([Lack, 2009](#)).

3.1.1 A brief introduction to 2-categories

Definition 3.1.1. A bicategory \mathcal{B} consists of

- ▷ a set of 0-cells $|\mathcal{B}|$,
- ▷ for each pair of 0-cells $x, y \in |\mathcal{B}|$, a category $\mathcal{B}(x, y)$ whose objects are called 1-cells and morphisms are called 2-cells,
- ▷ with identity 1-cell id_x for each 0-cell x ,
- ▷ and a bifunctorial composition $\circ_{x,y,z} : \mathcal{B}(y, z) \times \mathcal{B}(x, y) \rightarrow \mathcal{B}(x, z)$ for 0-cells $x, y, z \in |\mathcal{B}|$,
- ▷ such that the following unitality and associativity square commute up to natural isomorphisms λ, ρ, α called respectively left unitor, right unitor and associator

$$\begin{array}{ccccc} \mathcal{B}(x, y) \times \mathcal{B}(y, y) & \xrightarrow{\circ} & \mathcal{B}(x, y) & \xleftarrow{\circ} & \mathcal{B}(y, y) \times \mathcal{B}(x, y) \\ & \nwarrow \rho & \parallel & \nearrow \lambda & \\ & \mathcal{B}(x, y) \times \text{id}_x & \mathcal{B}(x, y) & \text{id}_y \times \mathcal{B}(x, y) & \end{array}$$

$$\begin{array}{ccc} \mathcal{B}(y, z) \times \mathcal{B}(x, y) \times \mathcal{B}(w, x) & \xrightarrow{\circ \times \mathcal{B}(w, x)} & \mathcal{B}(x, z) \times \mathcal{B}(w, x) \\ \mathcal{B}(y, z) \times \circ \downarrow & & \downarrow \alpha \\ \mathcal{B}(y, z) \times \mathcal{B}(w, y) & \xrightarrow{\circ} & \mathcal{B}(w, z) \end{array}$$

- ▷ and such that the following two coherence diagrams commute where we abbreviated $\mathcal{B}(x, y)$ by $\mathcal{B}_{x,y}$ and noted \cdot for functor composition as well as action of functors on natural transformations.

$$\begin{array}{c} \circ \cdot (\circ \times \mathcal{B}_{x,y}) \cdot (\mathcal{B}_{y,z} \times \text{id}_y \times \mathcal{B}_{x,y}) \xrightarrow{\alpha \cdot (\mathcal{B}_{y,z} \times \text{id}_y \times \mathcal{B}_{x,y})} \circ \cdot (\mathcal{B}_{y,z} \times \circ) \cdot (\mathcal{B}_{y,z} \times \text{id}_y \times \mathcal{B}_{x,y}) \\ \searrow \circ \cdot (\rho \times \mathcal{B}_{x,y}) \quad \swarrow \circ \cdot (\mathcal{B}_{y,z} \times \lambda) \\ \quad \quad \quad \circ_{x,y,z} \end{array}$$

$$\begin{array}{c} \circ \cdot (\circ \times \mathcal{B}_{v,w}) \cdot (\circ \times \mathcal{B}_{x,w} \times \mathcal{B}_{v,w}) \\ \swarrow \circ \cdot (\alpha \times \mathcal{B}_{v,w}) \quad \searrow \alpha \cdot (\circ \times \mathcal{B}_{x,w} \times \mathcal{B}_{v,w}) \\ \circ \cdot (\circ \times \mathcal{B}_{v,w}) \cdot (\mathcal{B}_{y,z} \times \circ \times \mathcal{B}_{v,w}) \quad \quad \quad \circ \cdot (\circ \times \circ) \\ \swarrow \alpha \cdot (\mathcal{B}_{y,z} \times \circ \times \mathcal{B}_{v,w}) \quad \searrow \alpha \cdot (\mathcal{B}_{y,z} \times \mathcal{B}_{x,y} \times \circ) \\ \circ \cdot (\mathcal{B}_{y,z} \times \circ) \cdot (\mathcal{B}_{y,z} \times \circ \times \mathcal{B}_{v,w}) \xrightarrow{\circ \cdot (\mathcal{B}_{y,z} \times \alpha)} \circ \cdot (\mathcal{B}_{y,z} \times \circ) \cdot (\mathcal{B}_{y,z} \times \mathcal{B}_{x,y} \times \circ) \end{array}$$

The first coherence diagram means that simplifying identities on the left or on the right using the adequate unitor gives the same result and the second coherence diagram enforces associativity of the associator.

A bicategory \mathcal{B} where the associator and unitors are identities is called a *strict* 2-category. A folklore result from [Curien et al. \(2014\)](#); [Power \(1989\)](#) shows that any bicategory can be strictified to a strict 2-category in the sense that a bicategory \mathcal{B} can be embedded in a strict 2-category such that the embedding is an equivalence of bicategories. Another way to state this coherence theorem is that all diagrams built out of associators and unitors commute, and so we will omit them in all diagrams since they can be inserted in an essentially unique way.

Examples of 2-categories

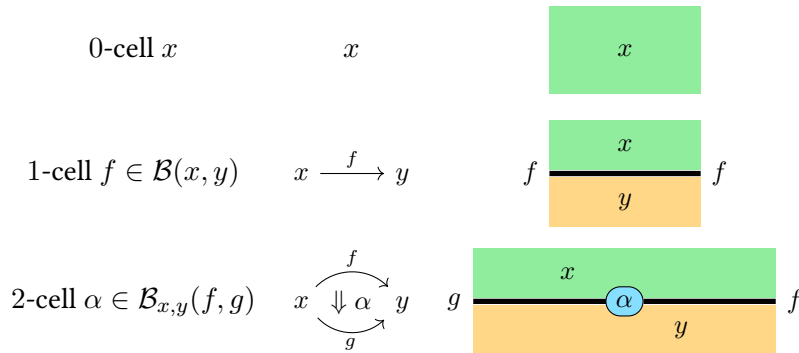
- ▷ \mathcal{Cat} is the 2-category of small categories, functors and natural transformations.
- ▷ For an enriching category \mathcal{V} , $\mathcal{V}\mathcal{Cat}$ is the 2-category of \mathcal{V} -enriched categories, \mathcal{V} -enriched functors and \mathcal{V} -enriched natural transformations.
- ▷ There is another natural 2-category whose 0-cells are small categories, the bicategory \mathcal{Distr} whose 1-cells are distributors between categories and 2-cells are natural transformations between distributors. The relationship between \mathcal{Cat} and \mathcal{Distr} can be seen as a categorification of the relationship between the (1-)categories \mathcal{Set} of set and functions and \mathcal{Rel} of sets and relations.

An adequate notion of morphism between 2-categories is that of 2-functor¹.

Definition 3.1.2. A 2-functor F from a 2-category \mathcal{B} to a 2-category \mathcal{K} consists of:

- ▷ a function $|F| : |\mathcal{B}| \rightarrow |\mathcal{K}|$ mapping 0-cells of \mathcal{B} to 0-cells of \mathcal{K}
- ▷ a functor $F_{x,y} : \mathcal{B}(x, y) \rightarrow \mathcal{K}(F x, F y)$ for each pair of 0-cells $x, y \in |\mathcal{B}|$
- ▷ with invertible 2-cells $i_x : \text{id}_{F x} \xrightarrow{\sim} F \text{id}_x$ for each 0-cell $x \in |\mathcal{B}|$ and $m_{f,g} : F g \circ F f \xrightarrow{\sim} F(g \circ f)$ for each pair of composable 1-cells f, g in \mathcal{B} , natural in f, g
- ▷ satisfying three coherence diagrams similar to those for a monoidal functor ensuring that unitors and associators are respected.

Working with 2-categories: string diagrams Since working inside a 2-category involves manipulating objects at three distinct levels, the usual diagrammatic notations can quickly become hard to read and obscure the actual proof. *String diagrams*, formally introduced in Joyal and Street (1991), provide a graphical calculus that can greatly simplify definitions and proofs inside a 2-category. The key idea is that proofs in a 2-category primarily manipulate 2-cells so they should be the most visible. This is achieved by taking the Poincaré dual of the standard diagrams: 0-cells become surfaces and 2-cells become points, whereas 1-cells are still represented as lines.



We leave out the name of 0-cells in further diagrams since these can be inferred from the 1-cells. Vertical juxtaposition of string diagrams correspond to composition of 1-cells – and its functorial action on 2-cells – whereas horizontal juxtaposition is composition of 2-cells. Note that we take here the slightly non-standard convention of reading string diagrams from top to bottom and from right to left in order to have compatible notations with the graphical calculus for framed bicategories in [section 3.3](#).

¹We only use in this manuscript the notion of *strong* 2-functor and not the more general notion of *lax* 2-functor defined in (Bénabou, 1967)

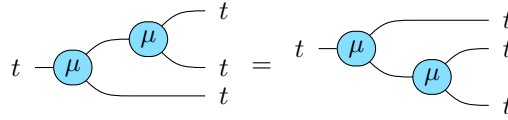
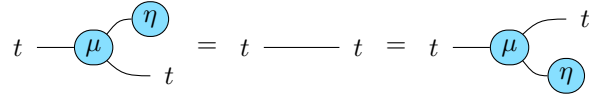
3.1.2 Monads in a 2-category

Definition 3.1.3. A monad in a 2-category \mathcal{B} consists of

- ▷ a 1-cell $t : X \rightarrow X$ on a 0-cell X of \mathcal{B}
- ▷ with 2-cells $\eta : \text{id}_X \rightarrow t$ and $\mu : t \circ t \rightarrow t$ called respectively unit and multiplication



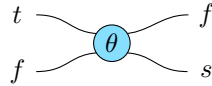
- ▷ such that the following diagrams commute



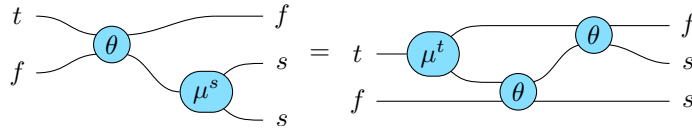
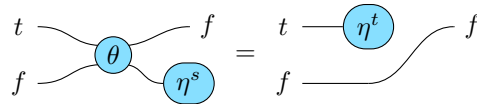
A monad in a bicategory \mathcal{B} is noted (X, t) leaving the unit and multiplication implicit. There is a natural notion of morphism between monads in a bicategory \mathcal{B} .

Definition 3.1.4. A monad morphism between monads (X, t) and (Y, s) consists of

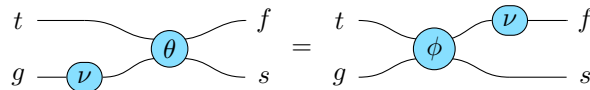
- ▷ A 1-cell $f : X \rightarrow Y$
- ▷ and a 2-cell $\theta : s \circ f \rightarrow f \circ t$



- ▷ such that the following identities hold



Definition 3.1.5. A monad morphism transformation between monad morphisms (f, θ) and (g, ϕ) from (X, t) to (Y, s) consists of a 2-cell $\nu : f \rightarrow g$ such that



We can put these definitions together to form a 2-category $\mathcal{Mnd}(\mathcal{B})$ whose 0-cells are monads in \mathcal{B} , 1-cells are monad morphisms and 2-cells are monad morphism transformations. We define a (2-)functor U from $\mathcal{Mnd}(\mathcal{B})$ to \mathcal{B} forgetting everything related to monads. In more details, U sends a monad (X, t) to X , a monad morphism (f, θ) to f and a monad morphism transformation to its underlying 2-cell.

Object of algebras

Definition 3.1.6. An algebra for a monad (T, η, μ) on a category \mathcal{C} is given by

- ▷ An object $c \in \mathcal{C}$ called the carrier of the algebra
- ▷ and a morphism $\alpha \in \mathcal{C}(T c, c)$ called the structure map of the algebra
- ▷ such that the two following identities hold

$$\alpha \circ \eta_c = \text{id}_c \qquad \alpha \circ \mu_c = \alpha \circ T \alpha$$

A T -algebra morphism from (c, α) to (c', α') consists of a morphism $f \in \mathcal{C}(c, c')$ such that $f \circ \alpha = \alpha' \circ T f$. T -algebras and T -algebra morphisms form a category called the Eilenberg-Moore category of T and note \mathcal{C}^T .

The formal theory of monads also extends the notion of algebra to an arbitrary 2-category. A monad (X, t) in a 2-category \mathcal{B} induces by post-composition a monad $\mathcal{B}(A, t)$ on the category $\mathcal{B}(A, X)$ for any 0-cell $A \in \mathcal{B}$. The mapping sending a 0-cell $A \in \mathcal{B}$ to the Eilenberg-Moore category $\mathcal{B}(A, X)^{\mathcal{B}(A, t)}$ extends to a 2-functor $\text{Alg}_t : \mathcal{B}^{\text{op}(1)} \rightarrow \text{Cat}$. An Eilenberg-Moore object for t is, when it exists, a 0-cell $X^t \in \mathcal{B}$ representing the functor Alg_t , that is such that $\text{Alg}_t(A) \cong \mathcal{B}(A, X^t)$ naturally in $A \in \mathcal{B}^{\text{op}(1)}$.

When Eilenberg-Moore objects exist for two monads (X, t) and (Y, s) , monad morphisms $(f, \theta) : (X, t) \rightarrow (Y, s)$ are in bijection with pairs of 1-cells (f, \tilde{f}) where $f : X \rightarrow Y$ and $\tilde{f} : X^t \rightarrow Y^s$.

3.2 Relative monads

The classical theory of monads is not enough to capture all the structure we need to model formal verification of programs. In particular specification monads (section 2.3) already go beyond the classical theory since they need to be equipped with orders. We could hope that it would be enough to move from a *Set*-enriched setting to a *Pos*-enriched setting, in the sense of enriched category theory (Kelly, 1982), however there is no reason *a priori* for a specification monad on *Set* to lift to *Pos*. Thus the formal monad theory in *PosCat* falls short of describing our peculiar use-case.

In order to provide a formal categorical account for specification monads, and for other monad-like objects developed for relational reasoning (chapter 6), we commit ourselves to a generalization of monads known as *relative monads*. A relative monad relax the notion of monad by endowing a monad-like structure to functors that need not to be endofunctors (Altenkirch et al., 2015). For this notion to make sense, we need to specify relative monads with respect to a base functor $\mathcal{J} : \mathcal{I} \rightarrow \mathcal{C}$, and the classical notion of monad is recovered when taking $\mathcal{J} = \text{Id}$. The price to pay for this generalization is a more technical theory, in particular to connect relative monads to a notion of monoid in an abstract enough setting.

Definition 3.2.1 (Relative monad in Cat). Let $\mathcal{J} : \mathcal{I} \rightarrow \mathcal{C}$ be a functor between categories \mathcal{I}, \mathcal{C} . A \mathcal{J} -relative monad is given by

- ▷ a function on objects $M : |\mathcal{I}| \rightarrow |\mathcal{C}|$,
- ▷ a family of morphisms $\text{ret}_x \in \mathcal{C}(\mathcal{J} x, M x)$ for any $x \in \mathcal{I}$,
- ▷ a family of functions $\text{bind}_{x,y} : \mathcal{C}(\mathcal{J} x, M y) \rightarrow \mathcal{C}(M x, M y)$

such that the following equations hold

$$\begin{aligned} \text{bind}_{x,x}(\text{ret}_x) &= \text{id}_x & \text{bind}_{x,y}(f) \circ \text{ret}_x &= f \\ \text{bind}_{x,z}(\text{bind}_{y,z}(g) \circ f) &= \text{bind}_{y,z}(g) \circ \text{bind}_{x,y}(f) \end{aligned}$$

The definition of a relative monad generalizes directly the presentation familiar to programmers of a plain monad on a category \mathcal{C} as a Kleisli triple $(T, \eta, (-)^\dagger)$ where we write $(-)^\dagger$ for the Kleisli extension operation $\mathcal{C}(X, TY) \rightarrow \mathcal{C}(TX, TY)$.

As hinted before, our examples of specification monads can be understood as relative monads from Set to Pos , relative to the functor Disc sending a set to itself seen as a discrete poset. However, since bind is required to be monotonic in both arguments, we will also need to consider a Pos -enriched setting. Note that the bind operation is defined as a function between hom-sets and need not to be representable as a “multiplication” natural transformation: there is in general no way to compose M twice. This means that in order to enrich this definition in a category \mathcal{V} , for instance $\mathcal{V} = \text{Pos}$, we need to consider not only \mathcal{V} -categories, \mathcal{V} -functors and \mathcal{V} -natural transformations, but also the structure of \mathcal{V} -hom objects, namely \mathcal{V} -profunctors. The further generalization to framed bicategories (Shulman, 2008) in the next sections will provide a synthetic and convenient context to consider these objects together.

Relative monads as presented in (Altenkirch et al., 2015) also come with their notions of morphism and algebras that we recall here. Until the end of this section, we fix categories \mathcal{I}, \mathcal{C} , a base functor $\mathcal{J} : \mathcal{I} \rightarrow \mathcal{C}$ and \mathcal{J} -relative monads M, M' .

Definition 3.2.2. A \mathcal{J} -relative monad morphism from M to M' is a natural transformation $\theta : M \rightarrow M'$ such that

$$\theta_x \circ \text{ret}_x^M = \text{ret}_x^{M'} \quad \theta_y \circ (\text{bind}_{x,y}^M f) = \text{bind}_{x,y}^{M'} (\theta_y \circ f) \circ \theta_x$$

for any objects $x, y \in \mathcal{I}$ and $f \in \mathcal{C}(\mathcal{J}x, My)$.

Definition 3.2.3. An Eilenberg-Moore algebra, or simply M -algebra, is an object $a \in \mathcal{C}$ together with a natural transformation

$$\alpha_x : \mathcal{C}(\mathcal{J}x, a) \rightarrow \mathcal{C}(Mx, a)$$

satisfying the two identities

$$\alpha_x(f) \circ \text{ret}_x = f \quad \alpha_y(\text{bind} f \circ g) = \alpha_x(f) \circ g$$

for any $x, y \in \mathcal{I}$, $f : \mathcal{J}x \rightarrow a$, $g : \mathcal{J}y \rightarrow Mx$.

M -algebras together with the appropriate notion of morphism form a category $\mathcal{EM}(M)$. The Kleisli category $\mathcal{Kl}(M)$ is the category with object set $|\mathcal{I}|$ and with morphisms $\mathcal{Kl}(M)(x, y) = \mathcal{C}(\mathcal{J}x, My)$. Any morphism of relative monad $\theta : M \rightarrow M'$ induces two factorizations

$$\mathcal{Kl}(\theta) : \mathcal{Kl}(M) \rightarrow \mathcal{Kl}(M') \quad \text{and} \quad \mathcal{EM}(\theta) : \mathcal{EM}(M') \rightarrow \mathcal{EM}(M).$$

3.3 Framed bicategories

We would like to extend the notion of relative monads to categories other than Cat , in particular to the ordered setting, however 2-categories does not seem to be the right setting for a formal theory of relative monads. Indeed, an object of an arbitrary 2-category does not necessarily have the hom-structure that we would need in order to define a bind operation. The notion of *framed bicategory* introduced by Shulman (2008) provides this data. In this section, we briefly present this notion and recall the instances that we will use further.

Definition 3.3.1. A framed bicategory \mathcal{F} is a double category with a distinguished class of 2-cells verifying a universal property, that is

- ▷ a set of objects or 0-cells $|\mathcal{F}|$;
- ▷ for each pair of objects $X, Y \in |\mathcal{F}|$, a set of vertical arrows or vertical 1-cells $\mathcal{F}_v(X, Y)$, and a set of pro-arrows or horizontal 1-cells $\mathcal{F}_h(X, Y)$. We write $f : X \rightarrow Y$ for a vertical arrow $f \in \mathcal{F}_v(X, Y)$ and $h : X \rightrightarrows Y$ for a proarrow $M \in \mathcal{F}_h(X, Y)$.
- ▷ for each frame as on the left, a set of 2-cells ${}_f\mathcal{F}_g(M, N)$, where $\alpha \in {}_f\mathcal{F}_g(M, N)$ is noted as on the right

$$\begin{array}{ccc} X & \xrightarrow{M} & Y \\ f \downarrow & & \downarrow g \\ X' & \xrightarrow{N} & Y' \end{array} \qquad \begin{array}{ccc} X & \xrightarrow{M} & Y \\ f \downarrow & \Downarrow \alpha & \downarrow g \\ X' & \xrightarrow{N} & Y' \end{array}$$

- ▷ vertical and horizontal units and compositions \circ and \odot noted as follows,

$$\begin{array}{ccc} X & \xrightarrow{M} & Y \\ \parallel & & \parallel \\ X & \xrightarrow{M} & Y \end{array} \qquad \begin{array}{ccc} X & \xrightarrow{X} & X \\ f \downarrow & & \downarrow f \\ Y & \xrightarrow{Y} & Y \end{array}$$

$$\begin{array}{ccc} X_1 & \xrightarrow{M_1} & Y_1 \\ f \downarrow & \Downarrow \alpha & \downarrow f' \\ X_2 & \xrightarrow{M_2} & Y_2 \\ g \downarrow & \Downarrow \beta & \downarrow g' \\ X_3 & \xrightarrow{M_3} & Y_3 \end{array} = gf \quad \begin{array}{ccc} X_1 & \xrightarrow{M_1} & Y_1 \\ \downarrow & \Downarrow \beta\alpha & \downarrow \\ X_3 & \xrightarrow{M_3} & Y_3 \end{array} \quad \begin{array}{ccc} X & \xrightarrow{M_1} & Y \xrightarrow{M_2} Z \\ u \downarrow & \Downarrow \alpha & \downarrow v \quad \Downarrow \beta \quad \downarrow w \\ X' & \xrightarrow{N_1} & Y' \xrightarrow{N_2} Z' \end{array} = \begin{array}{ccc} X & \xrightarrow{M_1 \odot M_2} & Z \\ u \downarrow & \Downarrow \alpha \odot \beta & \downarrow w \\ X' & \xrightarrow{N_1 \odot N_2} & Z' \end{array}$$

Vertical composition is associative and unital, whereas the horizontal composition is usually associative and unital only up to coherent natural isomorphisms, the associator and unitors². We do not explicitly write those, appealing to the fact that they can be strictified in the same fashion as for (weak) 2-categories. The two compositions are related by a distributivity law that in fine ensures that all diagrams have a unique well-defined reading.

- ▷ for any vertical cells f, g and horizontal cell M as on the left, a cartesian filler for the niche formed by f, M, g , that is a 2-cell $\chi \in {}_f\mathcal{F}_g(f^*Mg^*, M)$. Being cartesian means here that χ satisfies the following universal property: any other filler $\alpha \in {}_{fh}\mathcal{F}_{gk}(N, M)$ of the niche, factors through χ , yielding a 2-cell $f^*\alpha g^* \in {}_h\mathcal{F}_k(N, f^*Mg^*)$ unique up to unique globular isomorphism (that is a 2-cell whose vertical borders are identities).

$$\begin{array}{ccc} X_1 & & Y_1 \\ f \downarrow & & \downarrow g \\ X_2 & \xrightarrow{M} & Y_2 \end{array} \rightsquigarrow \begin{array}{ccc} X_1 & \xrightarrow{f^*Mg^*} & Y_1 \\ f \downarrow & \Downarrow \chi & \downarrow g \\ X_2 & \xrightarrow{M} & Y_2 \end{array} \qquad \begin{array}{ccc} X_1 & \xrightarrow{N} & Y_1 \\ fh \downarrow & \Downarrow \alpha & \downarrow gk \\ X_3 & \xrightarrow{M} & Y_3 \end{array} = \begin{array}{ccc} X_1 & \xrightarrow{N} & Y_1 \\ h \downarrow & \Downarrow f^*\alpha g^* & \downarrow k \\ X_2 & \xrightarrow{f^*Mg^*} & Y_2 \\ f \downarrow & \Downarrow \chi & \downarrow g \\ X_3 & \xrightarrow{M} & Y_3 \end{array}$$

²We are actually describing a *pseudo* double category.

The framed category Distr The canonical example of a framed bicategory is given by Distr whose

- ▷ objects are (small) categories $\mathcal{C}, \mathcal{D} \in \text{Distr}$,
- ▷ vertical arrows $\mathcal{J} \in \text{Distr}_v(\mathcal{C}, \mathcal{D})$ are functors $\mathcal{J} : \mathcal{C} \rightarrow \mathcal{D}$,
- ▷ horizontal arrows $\mathcal{H} \in \text{Distr}_h(\mathcal{C}, \mathcal{D})$ are *distributors* $\mathcal{H} : \mathcal{C} \rightarrow \mathcal{D}$ (a.k.a. profunctors) (Ben-abou, 2000), that is bifunctors $\mathcal{H} : \mathcal{D}^{\text{op}} \times \mathcal{C} \rightarrow \text{Set}$, and
- ▷ 2-cells $\alpha \in {}_{\mathcal{J}}\text{Distr}_{\mathcal{K}}(\mathcal{G}, \mathcal{H})$ are natural transformations $\alpha_{c,d} : \mathcal{G}(d, c) \rightarrow \mathcal{H}(\mathcal{K}d, \mathcal{J}c)$

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{\mathcal{G}} & \mathcal{D} \\ \mathcal{J} \downarrow & \Downarrow \alpha & \downarrow \mathcal{K} \\ \mathcal{C}' & \xrightarrow{\mathcal{H}} & \mathcal{D}' \end{array}$$

Vertical identities and composition are identity functors and functor composition as in Cat . Horizontal identities $\mathcal{C}(-, -) : \mathcal{C} \rightarrow \mathcal{C}$ are hom-sets bifunctors and horizontal composition $\mathcal{H} \odot \mathcal{G} : \mathcal{C} \rightarrow \mathcal{E}$ for distributors $\mathcal{H} : \mathcal{C} \rightarrow \mathcal{D}$, $\mathcal{G} : \mathcal{D} \rightarrow \mathcal{E}$ is given by the coend formula

$$\mathcal{H} \odot \mathcal{G}(e, c) = \int^{d \in \mathcal{D}} \mathcal{G}(e, d) \times \mathcal{H}(d, c)$$

If $F, G : \mathcal{C} \rightarrow \mathcal{D}$ are functors, we note that vertical 2-cells $\alpha \in {}_F\text{Distr}_G(\mathcal{C}(-, -), \mathcal{D}(-, -))$ are in natural bijection with natural transformations $G \rightarrow F$, witnessed by the following calculation

$$\begin{aligned} [\mathcal{C}^{\text{op}} \times \mathcal{C}, \text{Set}](\mathcal{C}(-, -), \mathcal{D}(G-, F-)) &\cong \int_{(c, c') \in \mathcal{C}^{\text{op}} \times \mathcal{C}} \text{Set}(\mathcal{C}(c, c'), \mathcal{D}(Gc, Fc')) \\ &\cong \int_{c' \in \mathcal{C}} \int_{c \in \mathcal{C}^{\text{op}}} \text{Set}(\mathcal{C}(c, c'), \mathcal{D}(Gc, Fc')) \\ &\cong \int_{c' \in \mathcal{C}} [\mathcal{C}^{\text{op}}, \text{Set}](\mathcal{C}(-, c'), \mathcal{D}(G-, Fc')) \\ &\cong \int_{c' \in \mathcal{C}} \mathcal{D}(Gc', Fc') \\ &\cong [\mathcal{C}, \mathcal{D}](G, F) \end{aligned}$$

where the second isomorphism holds by Fubini, the fourth by Yoneda lemma and the others by formulation of the set of natural transformations as a end.

Because of this correspondence, we will more generally note $\alpha : g \rightarrow f$ for a 2-cell $\alpha \in {}_f\mathcal{F}_g(\mathcal{C}, \mathcal{D})$ in an arbitrary framed bicategory \mathcal{F} with identities as horizontal domains and codomains.

The framed category $\mathcal{V}\text{-Distr}$ If \mathcal{V} is a complete and cocomplete symmetric monoidal closed category, we can generalize the definition of Distr to the \mathcal{V} -enriched setting (Kelly, 1982), obtaining a framed bicategory $\mathcal{V}\text{-Distr}$ consisting of \mathcal{V} -categories, \mathcal{V} -functors, \mathcal{V} -distributors and \mathcal{V} -natural transformations between distributors. We will frequently use this example with \mathcal{V} a cartesian closed-category.

Underlying 2-categories A framed bicategory \mathcal{F} naturally induces two different 2-categories:

- ▷ the 2-category \mathcal{F}_v consisting of 0-cells, *vertical* arrows and vertical 2-cells, that is 2-cells such that the horizontal domains and codomains are identities:

$$\begin{array}{ccc} X & \xrightarrow{\text{id}} & X \\ f \downarrow & \Downarrow \alpha & \downarrow g \\ Y & \xrightarrow{\text{id}} & Y \end{array}$$

- ▷ the 2-category \mathcal{F}_h consisting of 0-cells, *horizontal* arrows and globular 2-cells, that is 2-cells such that the vertical arrows are identities:

$$\begin{array}{ccc} X & \xrightarrow{M} & Y \\ \parallel & \Downarrow \alpha & \parallel \\ X & \xrightarrow{N} & Y \end{array}$$

Depending on the context, we will sometimes consider the underlying 1-category of the (strict) 2-category \mathcal{F}_v keeping the same notation.

String diagrams notations To ease calculations in framed bicategories, we will use a variant of the string diagram notation developed in (Myers, 2016). As it is usually the case with string diagrams, we represent a cell on the left by its Poincaré dual on the right.

$$\begin{array}{ccc} X_1 & \xrightarrow{M} & Y_1 \\ f \downarrow & \Downarrow \alpha & \downarrow g \\ X_2 & \xrightarrow{N} & Y_2 \end{array} \qquad \begin{array}{c} M \\ \parallel \\ f \leftarrow \alpha \leftarrow g \\ \parallel \\ N \end{array}$$

A 0-cell is corresponds to an area, a vertical arrow to an horizontal simple line, an horizontal arrow to a double line and a 2-cell to a point, represented by a labelled node. We read these diagrams from top to bottom and right to left. This convention is taken to be coherent with the direction of natural transformations in Distr : a natural transformation $\alpha : \mathcal{K} \rightrightarrows \mathcal{J}$ between functors $\mathcal{K}, \mathcal{J} : \mathcal{C} \rightarrow \mathcal{D}$, that is $\alpha \in {}_{\mathcal{J}}\text{Distr}_{\mathcal{K}}(\mathcal{C}, \mathcal{D})$, will be drawn as as follows.

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{\zeta} & \mathcal{D} \\ \mathcal{J} \downarrow & \Downarrow \alpha & \downarrow \mathcal{K} \\ \mathcal{D} & \xrightarrow{\quad} & \mathcal{D} \end{array} \qquad \mathcal{J} \leftarrow \alpha \leftarrow \mathcal{K}$$

String diagrams also account for cartesian fillers by bending the strings. Given a niche as on the left, a cartesian filler for this niche is depicted on the right.

$$\begin{array}{ccc} f \leftarrow & ? & \leftarrow g \\ \parallel & & \parallel \\ M & & M \end{array} \qquad \begin{array}{c} f \leftarrow \swarrow \quad \searrow \leftarrow g \\ \parallel \\ M \end{array}$$

$$\begin{array}{c} N \\ \parallel \\ h \leftarrow \boxed{f^* \alpha g^*} \leftarrow k \\ \downarrow \quad \downarrow \quad \downarrow \\ f^* M g^* \end{array} = \begin{array}{c} N \\ \parallel \\ h \leftarrow \alpha \leftarrow k \\ \swarrow f \quad \searrow g \\ \parallel \\ M \end{array}$$

Applying the property to the adequate niche, we have for any vertical arrow $f : X \rightarrow Y$, horizontal arrows $f^*Y : X \rightrightarrows Y$ and $Yf^* : Y \rightrightarrows X$ called respectively the companion and conjoint of f . The relationship between f , f^*Y and Yf^* is described by the following string diagrams:

$$\begin{array}{ccc}
\begin{array}{c} f^*Y \\ \downarrow \\ f \end{array} & \begin{array}{c} f \\ \downarrow \\ f^*Y \end{array} & \begin{array}{c} Yf^* \\ \downarrow \\ f \end{array} \quad \begin{array}{c} f \\ \downarrow \\ Yf^* \end{array} \\
\\
\begin{array}{c} f \\ \downarrow \\ f \end{array} & = & f \longleftarrow f \\
\\
\begin{array}{c} f^*Y \\ \downarrow \\ f^*Y \end{array} & = & \begin{array}{c} f^*Y \\ \downarrow \\ f^*Y \end{array} \\
\\
\begin{array}{c} Yf^* \\ \downarrow \\ Yf^* \end{array} & = & \begin{array}{c} Yf^* \\ \downarrow \\ Yf^* \end{array}
\end{array}$$

Since vertical arrows can be bent both to a companion and conjoint, we mark the direction of the arrow to keep track of which we are talking about: a single vertical line is a companion when it is directed from top to bottom, and a conjoint when directed from bottom to top. We will sometimes abbreviate both f^*Y and Yf^* by just f^* on diagrams to simplify notations, leaving to the reader the task to infer whether we are talking about the conjoint or companion of f from the non-ambiguous direction of arrows.

3.4 Framed functor, framed representability

In order to define objects by universal properties in a framed bicategory, we develop the basic notions of framed representability. We start by recalling the notions of (strong) framed functor and framed natural transformation defined in (Shulman, 2008).

Definition 3.4.1. *Let \mathcal{F}, \mathcal{G} be framed bicategories. A (strong) framed functor $\mathcal{K} : \mathcal{F} \rightarrow \mathcal{G}$ is a vertically strict, horizontally strong double functor between the underlying double categories of \mathcal{F} and \mathcal{G} . In components, it consists of:*

- ▷ a function $\mathcal{K} : |\mathcal{F}| \rightarrow |\mathcal{G}|$ from 0-cells in \mathcal{F} to 0-cells in \mathcal{G} ,
- ▷ a functorial action on vertical 1-cells $\mathcal{K}_v : \mathcal{F}_v(X, Y) \rightarrow \mathcal{G}_v(\mathcal{K}X, \mathcal{K}Y)$,
- ▷ a pseudo-functorial action on horizontal 1-cells $\mathcal{K}_h : \mathcal{F}_h(X, Y) \rightarrow \mathcal{G}_h(\mathcal{K}X, \mathcal{K}Y)$ with globular 2-cells $\mathcal{K}_h M \odot \mathcal{K}_h N \cong \mathcal{K}_h (M \odot N)$ and $\mathcal{K}_h X \cong \mathcal{K}X$ satisfying the coherence axioms for a strong 2-functor,
- ▷ a functorial assignment of 2-cells

$$\begin{array}{ccc}
X_1 \xrightarrow{M} Y_1 & & \mathcal{K}X_1 \xrightarrow{\mathcal{K}_h M} \mathcal{K}Y_1 \\
f \downarrow \quad \Downarrow \alpha \quad \downarrow g & \mapsto & \mathcal{K}_v f \downarrow \quad \Downarrow \mathcal{K}\alpha \quad \downarrow \mathcal{K}_v g \\
X_2 \xrightarrow{N} Y_2 & & \mathcal{K}X_2 \xrightarrow{\mathcal{K}_h N} \mathcal{K}Y_2
\end{array}$$

Fixing a framed bicategory \mathcal{F} , any object $C \in |\mathcal{F}|$ defines a “framed presheaf” \mathfrak{L}_C , that is a framed functor from \mathcal{F}^{op} , the framed category obtained from \mathcal{F} by formally reversing the direction of vertical 1-cells and 2-cells, to the framed bicategory Distr of categories, functors, distributors and natural transformations. The framed functor \mathfrak{L}_C maps:

- ▷ a 0-cell $X \in |\mathcal{F}|$ to the category $\mathfrak{L}_C(X) = \mathcal{F}_v(X, C)$ of vertical morphisms from X to C and vertical 2-cells;

- ▷ a vertical 1-cell $f \in \mathcal{F}_v^{\text{op}}(X, Y) = \mathcal{F}_v(Y, X)$ to the functor $\mathcal{F}_v(X, C) \rightarrow \mathcal{F}_v(Y, C)$ obtained by precomposition by f ;
- ▷ an horizontal 1-cell $M \in \mathcal{F}_h(X, Y)$ to the distributor whose component at $h_X \in \mathcal{F}_v(X, C), h_Y \in \mathcal{F}_v(Y, C)$ consists of the set of 2-cells $\mathfrak{J}_C(M)_{h_X, h_Y} = \{ \alpha \mid \alpha \in {}_{h_X}\mathcal{F}_{h_Y}(M, C) \}$ as represented below on the left, and the functorial action is given by composition of vertical 2-cells $\alpha_X : g_X \rightrightarrows h_X, \alpha_Y : h_Y \rightrightarrows g_Y$ on the sides as described on the right

$$\begin{array}{ccc}
X & \xrightarrow{M} & Y \\
h_X \downarrow & \Downarrow \alpha & \downarrow h_Y \\
C & \xrightarrow{C} & C
\end{array}
\qquad
\begin{array}{ccccccc}
X & \xrightarrow{X} & X & \xrightarrow{M} & Y & \xrightarrow{Y} & Y \\
g_X \downarrow & \Downarrow \alpha_X & h_X \downarrow & \Downarrow \alpha & \downarrow h_Y & \Downarrow \alpha_Y & \downarrow g_Y \\
C & \xrightarrow{C} & C & \xrightarrow{C} & C & \xrightarrow{C} & C
\end{array}$$

- ▷ a 2-cell $\gamma \in {}_f\mathcal{F}^{\text{op}}_g(M, N) = {}_f\mathcal{F}^{\text{op}}_g(N, M)$ to the natural transformation between distributors $\mathfrak{J}_C(\gamma) : \mathfrak{J}_C(M) \rightrightarrows \mathfrak{J}_C(N)$ given at component $h_X \in \mathcal{F}_v(X, C), h_Y \in \mathcal{F}_v(Y, C)$ by the function

$$\mathfrak{J}_C(\gamma)_{h_X, h_Y} = \begin{array}{ccc} X & \xrightarrow{M} & Y \\ h_X \downarrow & \Downarrow \alpha & \downarrow h_Y \\ C & \xrightarrow{C} & C \end{array} \longmapsto \begin{array}{ccc} X' & \xrightarrow{N} & Y' \\ f \downarrow & \Downarrow \gamma & \downarrow g \\ X & \xrightarrow{M} & Y \\ h_X \downarrow & \Downarrow \alpha & \downarrow h_Y \\ C & \xrightarrow{C} & C \end{array}$$

Given a vertical 1-cell $f : C \rightarrow C'$ in \mathcal{F} , we can define functors $\mathfrak{J}_f(X) : \mathfrak{J}_C(X) \rightarrow \mathfrak{J}_{C'}(X)$ for any object $X \in |\mathcal{F}|$ by postcomposition with f . We can extend this family of functors to define a *framed natural transformation* $\mathfrak{J}_f : \mathfrak{J}_C \rightrightarrows \mathfrak{J}_{C'}$

Definition 3.4.2. A framed natural transformation $\nu : \mathcal{K} \rightrightarrows \mathcal{L}$ between framed functors $\mathcal{K}, \mathcal{L} : \mathcal{F} \rightarrow \mathcal{G}$ consists of a family $\nu_X : \mathcal{K} X \rightarrow \mathcal{L} X$ of vertical 1-cells of \mathcal{G} indexed by 0-cells $X \in |\mathcal{F}|$ natural with respect to vertical 1-cells in \mathcal{F} and a compatible family $\nu_M \in {}_{\nu_X}\mathcal{G}_{\nu_Y}(\mathcal{K}_h M, \mathcal{L}_h M)$ indexed by horizontal 1-cells $M \in \mathcal{F}_h(X, Y)$ natural with respect to 2-cells in \mathcal{F} satisfying additionally the two equations

$$\begin{array}{ccc}
\mathcal{K} X & \xrightarrow{\mathcal{K}_h M} & \mathcal{K} Z \\
\nu_X \downarrow & \Downarrow \nu_{M \odot N} & \downarrow \nu_Y \\
\mathcal{L} X & \xrightarrow{\mathcal{L}_h M} & \mathcal{L} Z
\end{array} = \begin{array}{ccccccc}
\mathcal{K} X & \xrightarrow{\mathcal{K}_h M} & \mathcal{K} Y & \xrightarrow{\mathcal{K}_h N} & \mathcal{K} Z \\
\nu_X \downarrow & \Downarrow \nu_M & \downarrow \nu_Y & \Downarrow \nu_M & \downarrow \nu_Z \\
\mathcal{L} X & \xrightarrow{\mathcal{L}_h M} & \mathcal{L} Y & \xrightarrow{\mathcal{L}_h N} & \mathcal{L} Z
\end{array}$$

$$\begin{array}{ccc}
\mathcal{K} X & \xrightarrow{\mathcal{K}_h X} & \mathcal{K} X \\
\nu_X \downarrow & \Downarrow \nu_X & \downarrow \nu_X \\
\mathcal{L} X & \xrightarrow{\mathcal{L}_h X} & \mathcal{L} X
\end{array} = \begin{array}{ccc}
\mathcal{K} X & \xrightarrow{\mathcal{K}_h X} & \mathcal{K} X \\
\nu_X \downarrow & & \downarrow \nu_X \\
\mathcal{L} X & \xrightarrow{\mathcal{L}_h X} & \mathcal{L} X
\end{array}$$

where we silently use the isomorphisms witnessing pseudo-functoriality for horizontal composition and identities.

It is shown in (Shulman, 2008) (proposition 6.17) that framed bicategories, framed functors and framed natural transformations form a strict 2-category. We use part of that fact to state the following lemma:

Lemma 3.4.1. For any framed bicategory \mathcal{F} , the assignment $C \mapsto \mathfrak{J}_C$ extends to a functor $\mathfrak{J} : \mathcal{F}_v \rightarrow [\mathcal{F}^{\text{op}}, \text{Distr}]$ where \mathcal{F}_v is the 1-category defined by vertical arrows in \mathcal{F} and $[\mathcal{F}^{\text{op}}, \text{Distr}]$ is the category of framed functors from \mathcal{F}^{op} to Distr and framed natural transformations.

Proof. We already proved that $\mathfrak{J}_C : \mathcal{F}^{\text{op}} \rightarrow \text{Distr}$ is a framed functor. Given a 1-cell $f : C \rightarrow C'$, the framed natural transformation $\mathfrak{J}_f : \mathfrak{J}_C \rightarrow \mathfrak{J}_{C'}$ is given on a 0-cell X by the functor defined by postcomposition by f , and on a horizontal 1-cell M by the natural transformation between distributors $\mathfrak{J}_C(M) \rightarrow \mathfrak{J}_{C'}(M)$ induced by postcomposition with the identity 2-cell on f . \square

Lemma 3.4.2 (framed (weak) Yoneda lemma). *Let \mathcal{F} be a framed category, C an object of \mathcal{F} and $\mathcal{H} : \mathcal{F}^{\text{op}} \rightarrow \text{Distr}$ a framed functor. There is a natural bijection*

$$[\mathcal{F}^{\text{op}}, \text{Distr}](\mathfrak{J}_C, \mathcal{H}) \cong |\mathcal{H}(C)|$$

Proof. The bijection $\varphi : [\mathcal{F}^{\text{op}}, \text{Distr}](\mathfrak{J}_C, \mathcal{H}) \xrightarrow{\sim} |\mathcal{H}(C)|$ is defined by $\varphi(\nu) = \nu_C(\text{id}_C)$. For an object $h \in \mathcal{H}(C)$, its inverse $\varphi^{-1}(h)$ is the framed natural transformation given at components $X \in |\mathcal{F}|$ and $M : X \rightarrowtail Y$ by

$$\begin{aligned} \varphi^{-1}(h)_X &= f \in \mathfrak{J}_C(X) = \mathcal{F}_v(X, C) \mapsto \mathcal{H}(f)(h) \in \mathcal{H}(X) \\ \varphi^{-1}(h)_M &= \alpha \in \mathfrak{J}_C(M) \mapsto \mathcal{H}(\alpha)(\text{id}_h) \in \mathcal{H}(M) \end{aligned}$$

where $\text{id}_h \in \text{Id}_{\mathcal{H}(C)}(h, h) \cong \mathcal{H}(\text{Id}_C)(h, h)$ is the element of the distributor $\mathcal{H}(\text{Id}_C)$ representing the identity on h . To prove that they are inverse to each other, we first compute $\varphi \circ \varphi^{-1}(h) = \varphi_C^{-1}(\text{id}_C) = \mathcal{H}(\text{id}_C)(h) = h$. For the other equality, we observe that, by naturality, any natural transformation $\nu \in [\mathcal{F}^{\text{op}}, \text{Distr}](\mathfrak{J}_C, \mathcal{H})$ verifies the identities

$$\begin{aligned} \nu_X(f) &= \nu_X \circ \mathfrak{J}_C(f)(\text{id}_C) = \mathcal{H}(f) \circ \nu_C(\text{id}_C) \\ \nu_M(\alpha) &= \nu_M \circ \mathfrak{J}_C(\alpha)(\text{Id}_C) = \mathcal{H}(\alpha) \circ \nu_{\text{Id}_C}(\text{id}_{\text{Id}_C}) \end{aligned}$$

where $X \in \mathcal{F}, f \in \mathfrak{J}_C(X) = \mathcal{F}_v(X, C), M : X \rightarrowtail Y, \alpha \in \mathfrak{J}_C(M) = \bigcup_{f,g} f\mathcal{F}_g(M, C)$. In particular, we have

$$\begin{aligned} \varphi^{-1}(\varphi(\nu))_X(f) &= \mathcal{H}(f)(\varphi(\nu)) = \mathcal{H}(f) \circ \nu_C(\text{id}_C) = \nu_X(f) \\ \varphi^{-1}(\varphi(\nu))_M(\alpha) &= \mathcal{H}(\alpha)(\text{id}_{\varphi(\nu)}) = \mathcal{H}(\alpha) \circ \nu_{\text{Id}_C}(\text{id}_{\text{Id}_C}) = \nu_M(\alpha) \end{aligned}$$

that is $\varphi^{-1} \circ \varphi(\nu) = \nu$. \square

Corollary 3.4.1. *The functor $\mathfrak{J} : \mathcal{F}_v \rightarrow [\mathcal{F}^{\text{op}}, \text{Distr}]$ is full and faithful.*

Proof. For any objects $C, D \in |\mathcal{F}_v|$,

$$[\mathcal{F}^{\text{op}}, \text{Distr}](\mathfrak{J}_C, \mathfrak{J}_D) \cong |\mathfrak{J}_D(C)| = \mathcal{F}_v(C, D)$$

\square

We say that a framed functor $\mathcal{H} : \mathcal{F}^{\text{op}} \rightarrow \text{Distr}$ is represented by an object $C \in |\mathcal{F}|$ if there is a framed natural isomorphism $\mathcal{H} \cong \mathfrak{J}_C$, and \mathcal{H} is *representable* if there exists an object C representing it.

Dually, any object $C \in |\mathcal{F}|$ defines a framed functor ${}^c\mathfrak{J}_C : \mathcal{F} \rightarrow \text{Distr}$ whose action on 0-cell is given by ${}^c\mathfrak{J}_C(X) = \mathcal{F}_v(C, X)$. Dualizing all the previous discussion, ${}^c\mathfrak{J}$ defines a full and faithful functor $\mathcal{F}_v \rightarrow [\mathcal{F}, \text{Distr}]^{\text{op}}$. A framed functor $\mathcal{G} : \mathcal{F} \rightarrow \text{Distr}$ is said to be *co-representable* if it is isomorphic to ${}^c\mathfrak{J}_C$ for some $C \in |\mathcal{F}|$.

3.5 Relative monad in a framed bicategory

Having in hand the powerful notion of framed bicategory, we now set out to define what is a relative monad inside a framed bicategory. The idea is that thanks to the horizontal morphisms playing the role of a hom-structure on the objects of a framed bicategory \mathcal{F} , we can almost reenact the definition in \mathcal{Cat} in \mathcal{F} , replacing the base functor by a vertical arrow.

Definition 3.5.1 (Relative monad). *Let \mathcal{F} be a framed bicategory, $j : I \rightarrow C$ be a vertical 1-cell of \mathcal{F} . A j -relative monad is a triple $(m, \text{ret}, \text{bind})$ composed of*

- ▷ a vertical 1-cell $m : I \rightarrow C$,
- ▷ a 2-cell $\text{ret} : j \rightarrow m$, or diagrammatically

$$\begin{array}{ccc} I & \xlongequal{\quad} & I \\ m \downarrow & \Downarrow \text{ret} & \downarrow j \\ C & \xlongequal{\quad} & C \end{array} \quad m - \text{ret} - j$$

- ▷ a 2-cells $\text{bind} : C(j, m) \rightarrow C(m, m)$, where we note $C(h, k) = k^* C h^*$,

$$\begin{array}{ccc} I & \xrightarrow{C(j, m)} & I \\ \parallel \Downarrow \text{bind} \parallel & & \\ I & \xrightarrow{C(m, m)} & I \end{array} \quad \begin{array}{ccc} m^* C & C j^* & \\ \downarrow & \uparrow & \\ & \text{bind} & \\ \downarrow & \uparrow & \\ m^* C & C m^* & \end{array}$$

inducing a mapping $(-)^{\dagger}$ from 2-cells to 2-cells ${}_{mb}\mathcal{F}_{ja}(M, C) \rightarrow {}_{mb}\mathcal{F}_{ma}(M, C)$ for vertical arrows $a : A \rightarrow C, b : B \rightarrow C$ and horizontal arrow $M : B \rightarrow A$

$$\begin{array}{c} R \\ \parallel \\ b \leftarrow \leftarrow \leftarrow \leftarrow \leftarrow a \\ \leftarrow \leftarrow \leftarrow \leftarrow \leftarrow m \\ \leftarrow \leftarrow \leftarrow \leftarrow \leftarrow m \\ \leftarrow \leftarrow \leftarrow \leftarrow \leftarrow \alpha^{\dagger} \end{array} = \begin{array}{c} R \\ \parallel \\ b \leftarrow \leftarrow \leftarrow \leftarrow \leftarrow a \\ \leftarrow \leftarrow \leftarrow \leftarrow \leftarrow m \\ \leftarrow \leftarrow \leftarrow \leftarrow \leftarrow j \\ \leftarrow \leftarrow \leftarrow \leftarrow \leftarrow \alpha \\ \leftarrow \leftarrow \leftarrow \leftarrow \leftarrow \text{bind} \\ \leftarrow \leftarrow \leftarrow \leftarrow \leftarrow m \end{array}$$

- ▷ such that the following equations hold

$$\text{ret}^{\dagger} = \text{id}_m \quad \alpha^{\dagger} \circ \text{ret} = \alpha \quad (\beta^{\dagger} \circ \alpha)^{\dagger} = \beta^{\dagger} \circ \alpha^{\dagger}$$

for any objects $X, Y, Z \in \mathcal{F}$, vertical 1-cells $f : X \rightarrow i, g : Y \rightarrow i, h : Z \rightarrow i$ and 2-cells $\alpha : j \circ f \rightarrow m \circ g, \beta : j \circ g \rightarrow m \circ h$. In string diagrams notations, the relative monad equations give respectively

$$\begin{array}{ccc} \begin{array}{c} \text{ret} \\ \downarrow \uparrow \\ \text{bind} \\ \downarrow \uparrow \\ m \end{array} & = & m - m \\ \begin{array}{ccc} m^* C & C j^* & \\ \downarrow & \uparrow & \\ & \text{bind} & \\ \downarrow & \uparrow & \\ m^* C & C m^* & \end{array} & = & \begin{array}{ccc} m^* C & C j^* & \\ \downarrow & \uparrow & \\ & \text{ret} & j \\ \downarrow & \uparrow & \\ m^* C & & \end{array} \end{array}$$

$$\begin{array}{ccc} \begin{array}{ccc} m^* C & C j^* & m^* C & C j^* \\ \downarrow & \uparrow & \downarrow & \uparrow \\ & \text{bind} & & \text{bind} \\ \downarrow & \uparrow & \downarrow & \uparrow \\ m^* C & & C m^* & \end{array} & = & \begin{array}{ccc} m^* C & C j^* & m^* C & C j^* \\ \downarrow & \uparrow & \downarrow & \uparrow \\ & \text{bind} & & \text{bind} \\ \downarrow & \uparrow & \downarrow & \uparrow \\ m^* C & & C m^* & \end{array} \end{array}$$

Inside Distr A relative monad in Distr is the same thing as a relative monad over a functor (Def. 3.2.1) between small categories. Given such a j -relative monad m in Distr , j being a vertical arrow therein, it is quite immediate that m has the structure of a relative monad over the functor j and satisfies the required equations.

Conversely, in order to show that a relative monad $(M, \text{ret}, \text{bind})$ over a functor $\mathcal{J} : I \rightarrow C$ is a relative monad in Distr , it is enough to show that M extends to a functor $I \rightarrow C$ and that ret and bind are natural in the appropriate sense. The relevant proof can be found in (Altenkirch et al., 2015).

Specification monads as relative monads in Pos-Distr Specification monads are our original motivation for moving to relative monads in the enriched setting of Pos-Distr . Indeed, the two ingredients needed to define specification monads are:

1. a carrier for specification monads mapping sets to preorders, and
2. preordered hom-sets, and natural transformations between preorder enriched posets that are monotonic so that the bind operation is monotonic in *both* arguments.

The main protagonists for a formal definition of specification monads are the Pos -categories Set and Pos ³ where the former is seen as an enriched category through the monoidal⁴ functor $\text{Disc} : \text{Set} \rightarrow \text{Pos}$ sending a set to itself equipped with the discrete preorder, and the latter by cartesian closedness. We note $\underline{\text{Disc}} : \text{Set} \rightarrow \text{Pos}$ the lifting of Disc to Pos -functor.

Definition 3.5.2. A specification monad is a $\underline{\text{Disc}}$ -relative monad in Pos-Distr .

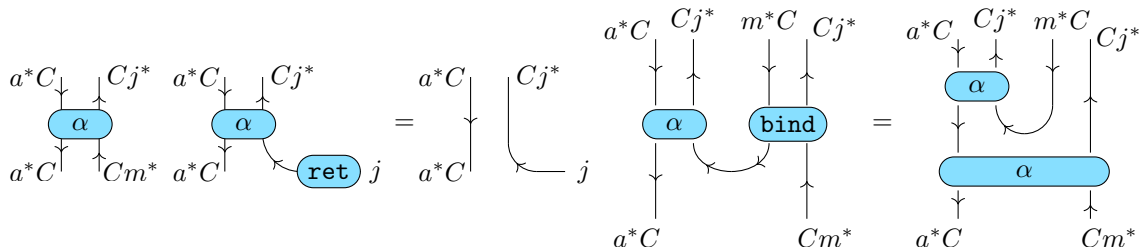
This can be seen as a more abstract presentation of preorder-enriched monads (Katsumata and Sato, 2013; Rauch et al., 2016).

Eilenberg-Moore Algebras We fix a base vertical arrow $j : I \rightarrow C$ and a j -relative monad m in \mathcal{F} . We want to extend the notion of algebra for a relative monad. Of course, in a general framed bicategory we may not have a “unit object,” as we do in Cat , so we need to define algebras with respect to an arbitrary vertical arrow instead of an “object of C ”, in a similar way as using generalized elements in ordinary category theory.

Definition 3.5.3. An m -algebra is an object $A \in \mathcal{F}$, a vertical arrow $a : A \rightarrow C$ together with a natural transformation $\alpha : C(j, a) \rightarrow C(m, a)$ satisfying the two identities

$$\alpha_x(f) \circ \text{ret}_x = f \qquad \alpha_y(\alpha_x(f) \circ g) = \alpha_x(f) \circ \text{bind}_g$$

for any $x, y \in \mathcal{I}$, $f : \mathcal{J}x \rightarrow a$, $g : \mathcal{J}y \rightarrow Mx$.



Definition 3.5.4. An m -algebra morphism between m -algebras (A, a, α) and (B, b, β) is a pair of a vertical arrow $f : A \rightarrow B$ and a 2-cell $\varphi : a \rightarrow bf$ satisfying the equation

³In order to see these as objects of Pos-Distr , we need to distinguish two levels of smallness, as provided by different universes. Since our constructions are independent of the universe level – they are universe polymorphic – we keep the same notations for all levels.

⁴for the cartesian product monoidal structure on Set and Pos .

$$f \circ \alpha = \beta$$

There is a framed functor $\mathcal{A}lg_m : \mathcal{F}_v^{\text{op}} \rightarrow \text{Distr}$ sending an object A to the category of m -algebras and m -algebra morphisms between them. The action of $\mathcal{A}lg_m$ on a vertical 1-cell $f : A \rightarrow B$ gives a functor $\mathcal{A}lg_m(f) : \mathcal{A}lg_m(B) \rightarrow \mathcal{A}lg_m(A)$ sending an m -algebra $(B, b : B \rightarrow C, \beta)$ from B to an m -algebra $(A, bf : A \rightarrow C, \beta_f)$ where β_f is defined as follow using that $(bf)^*C \cong f^*b^*C$:

$$f^* \circ \beta = \beta_f$$

On a horizontal 1-cell $M : A \rightarrow B$, $\mathcal{A}lg_m(M)$ is the distributor whose component at $(a : A \rightarrow C, \alpha), (b : B \rightarrow C, \beta)$ is given by the set of 2-cells $\nu \in {}_a\mathcal{F}_b(M, C)$ such that

$$M \circ \nu = \alpha$$

Finally, the action of $\mathcal{A}lg_m$ on a 2-cell is given by vertical precomposition.

Definition 3.5.5. An Eilenberg-Moore object for a j -relative monad m in \mathcal{F} is a representing object for the framed functor $\mathcal{A}lg_m$ (in the sense of [section 3.4](#)). When it exists, it is noted $\mathcal{E}M_m$.

The relative monad m always has a canonical m -algebra structure given by bind , so it induces a factorization in $[\mathcal{F}^{\text{op}}, \text{Distr}]$

$$\begin{array}{ccc} & \mathcal{A}lg_m & \\ \tilde{m} \nearrow & & \searrow u \\ \mathcal{J}_I & \xrightarrow{\mathcal{J}_m} & \mathcal{J}_C \end{array}$$

where u forgets the m -algebra structure and only keep the underlying arrows in \mathcal{F} . When m has an Eilenberg-Moore object, this factorization happens directly inside \mathcal{F}_v by the framed Yoneda lemma [Lem. 3.4.2](#).

In the case of the framed bicategory $\mathcal{V}\text{-Distr}$ for \mathcal{V} a suitable category for enrichment, these Eilenberg-Moore objects exists up to size conditions. Let $\mathcal{J} : \mathcal{I} \rightarrow \mathcal{C}$ be a \mathcal{V} -functor between \mathcal{V} -categories \mathcal{I}, \mathcal{C} and $M : \mathcal{I} \rightarrow \mathcal{C}$ be a \mathcal{J} -relative monad (in $\mathcal{V}\text{-Distr}$). The \mathcal{V} -category \mathcal{C}^M has as underlying set of objects pairs (c, α) composed of an object $c \in |\mathcal{C}|$ and a \mathcal{V} -natural transformation $\alpha : \mathcal{C}(\mathcal{J} -, c) \rightarrow \mathcal{C}(M -, c)$ between \mathcal{V} -presheaves. The object of morphism between (c_1, α_1) and (c_2, α_2) is obtained as the equalizer

$$\mathcal{C}^M((c_1, \alpha_1), (c_2, \alpha_2)) \dashrightarrow \mathcal{C}(c_1, c_2) \begin{array}{c} \xrightarrow{\quad} \\ \prod_{x \in |\mathcal{I}|} [\mathcal{C}(\mathcal{J} x, c_1), \mathcal{C}(M x, c_2)] \\ \xleftarrow{\quad} \end{array}$$

where the arrows on the right are obtained by currying the two maps

$$\begin{array}{ccccc}
 & \circ & \rightarrow & \mathcal{C}(\mathcal{J}x, c_2) & \xrightarrow{\alpha_2} \\
 \mathcal{C}(c_1, c_2) \otimes \mathcal{C}(\mathcal{J}x, c_1) & & \times & & \mathcal{C}(Mx, c_2) \\
 & \xrightarrow{\alpha_1} & & \mathcal{C}(c_1, c_2) \otimes \mathcal{C}(Mx, c_1) & \xrightarrow{\circ}
 \end{array}$$

Theorem 3.5.1. \mathcal{C}^M is the Eilenberg-Moore object of M in $\mathcal{V}\text{-Distr}$.

Proof. We need to show that the described \mathcal{V} -category \mathcal{C}^M is a representing object for the functor Alg_M , that is to exhibit a framed natural isomorphism $\varphi : \mathfrak{K}_{\mathcal{C}^M} \cong \text{Alg}_M$.

On object $X \in \mathcal{V}\text{-Distr}_v$, we define φ_X by projecting out the components $(uc, \alpha(c))$ out of an algebra $c \in \mathcal{C}^M$. For $f \in \mathfrak{K}_{\mathcal{C}^M}(X) = \mathcal{V}\text{-Distr}_v(X, \mathcal{C}^M)$, we define a \mathcal{J} -relative M -algebra structure on $uf : X \rightarrow \mathcal{C}$ by $\alpha(f)_{a,b} : \mathcal{C}(\mathcal{J}a, uf b) \rightarrow \mathcal{C}(Ma, uf b)$ whose naturality in b is provided by functoriality of f . A vertical 2-cell $\theta : f \rightarrow g$ is mapped to $\varphi_X(\theta) = u\theta : uf \rightarrow ug$. It is a \mathcal{J} -relative M -algebra homomorphism since each components of θ are.

Conversely, we define $\varphi^{-1}(f, \alpha) = \tilde{f} \in \mathfrak{K}_{\mathcal{C}^M}(X)$ for $(f, \alpha) \in \text{Alg}_M(X)$ by setting $\tilde{f}x = fx$ on objects $x \in X$ and obtaining the action of \tilde{f} on $X(x_1, x_2) \in \mathcal{V}$ from the universal property of the \mathcal{V} -hom of \mathcal{C}^M using the fact that α is an M -algebra structure on f :

$$\begin{array}{ccc}
 & X(x_1, x_2) & \\
 \tilde{f} \swarrow & \downarrow f & \\
 \mathcal{C}^M((f x_1, \alpha_{x_1}), (f x_2, \alpha_{x_2})) & \longrightarrow & \mathcal{C}(f x_1, f x_2) \xrightarrow{\prod_{x \in |I|} [\mathcal{C}(\mathcal{J}x, f x_1), \mathcal{C}(Mx, f x_2)]}
 \end{array}$$

For a vertical 2-cell $\theta : (f, \alpha) \rightarrow (g, \beta)$, we define the vertical 2-cell $\varphi_X^{-1}(\theta) = \tilde{\theta} : \tilde{f} \rightarrow \tilde{g}$ by another application of the universal property (where $\tilde{\theta}_x$ is the special case of the diagram below precomposed with id_x)

$$\begin{array}{ccc}
 & X(x_1, x_2) & \\
 \tilde{\theta}_{x_1} \circ f = g \circ \tilde{\theta}_{x_2} \swarrow & \downarrow \theta_{x_2} \circ f = g \circ \theta_{x_1} & \\
 \mathcal{C}^M((f x_1, \alpha_{x_1}), (g x_2, \beta_{x_2})) & \longrightarrow & \mathcal{C}(f x_1, g x_2) \xrightarrow{\prod_{x \in |I|} [\mathcal{C}(\mathcal{J}x, f x_1), \mathcal{C}(Mx, g x_2)]}
 \end{array}$$

φ_X^{-1} is indeed an inverse to φ_X by unicity of the universal property. Since the action of $\mathfrak{K}_{\mathcal{C}^M}$ and Alg_M on vertical arrows is given by precomposition and the definition of φ_X and its inverse only act on the codomain, there are natural with respect to vertical arrows. The definition of φ on horizontal arrows $M : X \rightarrow Y$ then proceeds similarly to the case of vertical 2-cells. \square

Kleisli algebras We introduce the dual notion to Eilenberg-Moore algebras for a relative monad, corresponding to a *right* module on a monad. Since they are not modules, and by lack of a suitable terminology, we call them here Kleisli algebras.

Definition 3.5.6. A Kleisli algebra is a vertical arrow $f : I \rightarrow X$ to some object $X \in \mathcal{F}$ together with a 2-cell $\alpha \in {}_I\mathcal{F}_I(\mathcal{C}(j, m), X(f, f))$ satisfying the two following equations

$$\begin{array}{c}
 \text{ret} \\
 \downarrow \\
 \alpha \\
 \uparrow \quad \downarrow \\
 f \quad f
 \end{array}
 = f$$

$$\begin{array}{ccc}
 m^*C & Cj^* & m^*C \\
 \downarrow & \downarrow & \downarrow \\
 \alpha & \alpha & \\
 \uparrow & \downarrow & \\
 f^*C & Cf^* &
 \end{array}
 =
 \begin{array}{c}
 m^*C & Cj^* & m^*C \\
 \downarrow & \downarrow & \downarrow \\
 \text{bind} & & \\
 \downarrow & & \\
 \alpha & & \\
 \uparrow & \downarrow & \\
 f^*C & Cf^* &
 \end{array}$$

Definition 3.5.7. A morphism of Kleisli algebras from (X, f, α) to (Y, g, β) is a pair of a vertical arrow $h : X \rightarrow Y$ and a 2-cell $\nu : hf \rightarrow g$ such that the following equation hold

In a similar fashion to Eilenberg-Moore algebras, a j -relative monad $m : I \rightarrow C$ in \mathcal{F} give raise to a framed functor $\mathcal{Kl}_m : \mathcal{F} \rightarrow \text{Distr}$ with the following components:

- ▷ An object $X \in \mathcal{F}$ is mapped to the category $\mathcal{Kl}_m(X)$ of m -Kleisli algebras with codomain X and m -Kleisli morphisms whose first component is the identity;
- ▷ A vertical arrow $f : X \rightarrow Y$ defines a functor $\mathcal{Kl}_m(X) \rightarrow \mathcal{Kl}_m(Y)$ by postcomposition;
- ▷ An horizontal arrow $M : X \rightarrowtail Y$ defines a profunctor $\mathcal{Kl}_m(M) : \mathcal{Kl}_m(X) \rightarrowtail \mathcal{Kl}_m(Y)$ whose component at $(f, \alpha) \in \mathcal{Kl}_m(X)$ and $(g, \beta) \in \mathcal{Kl}_m(Y)$ is the set of 2-cells $\chi \in {}_f\mathcal{F}_g(I, M)$ such that

- ▷ 2-cells $\varphi \in {}_h\mathcal{F}_{h'}(M, N)$ induce a natural transformations between profunctors by vertical postcomposition.

The framed natural transformation $\mathcal{Kl}_m : \mathcal{Kl}_C \rightarrowtail \mathcal{Kl}_I$ induced by the j -relative monad m always factors through \mathcal{Kl}_m

where λ and ρ are framed natural transformation defined at a 0-cell $X \in \mathcal{F}$ as follows

- ▷ λ forgets the Kleisli algebra structure and maps a pair $(f : I \rightarrow X, \alpha) \in \mathcal{Kl}_m(X)$ to $f \in \mathcal{Kl}_I(X) = \mathcal{F}_v(I, X)$;
- ▷ ρ sends a vertical arrow $f \in \mathcal{Kl}_C(X) = \mathcal{F}_v(C, X)$ to the Kleisli algebra (fm, α) where α is induced by the bind operation from m

We say that a j -relative monad m in \mathcal{F} has a *Kleisli object* if the framed functor \mathcal{Kl}_m is co-represented by a 0-cell $C_m \in \mathcal{F}$. In that situation, the factorization ${}^c\mathcal{K}_m = \lambda \circ \rho$ through \mathcal{Kl}_m above induces vertical arrows $l : I \rightarrow C_m$ and $r : C_m \rightarrow C$ such that $m = rl$.

In the particular case of $\mathcal{F} = \mathcal{V}\text{-Distr}$ for an enriching category \mathcal{V} , any \mathcal{J} -relative monad $M \in \mathcal{V}\text{-Distr}(\mathcal{I}, \mathcal{C})$ has a Kleisli object C_M . The explicit construction of C_M is quite standard: take $|\mathcal{I}|$ as the set of objects $|C_M|$ and for $x, y \in |\mathcal{I}|$ define $C_M(x, y) = \mathcal{C}(\mathcal{J}x, My)$ with identity given by $\text{ret}_x^M \in \mathcal{C}(\mathcal{J}x, Mx)$ and composition induced by bind^M , namely

$$\mathcal{C}(\mathcal{J}y, Mz) \times \mathcal{C}(\mathcal{J}x, My) \xrightarrow{\text{bind}^M \times \text{id}} \mathcal{C}(My, Mz) \times \mathcal{C}(\mathcal{J}x, My) \xrightarrow{\circ} \mathcal{C}(\mathcal{J}x, Mz).$$

Lemma 3.5.1. C_M is the Kleisli object of M in $\mathcal{V}\text{-Distr}$.

3.5.1 Morphisms of relative monads

Morphisms between two monads relative to the same base functor has been defined in (Altenkirch et al., 2015). Here we generalize the definition of morphisms to relative monads in a framed bicategory \mathcal{F} over possibly different base arrows.

Let \mathcal{F} be a framed bicategory and $j_1 : I_1 \rightarrow C_1, j_2 : I_2 \rightarrow C_2$ two vertical morphisms in \mathcal{F} . A morphism from $j_1 \rightarrow j_2$ is a pair of vertical cells $u_{\text{dom}} : I_1 \rightarrow I_2$ and $u_{\text{cod}} : C_1 \rightarrow C_2$ and an invertible 2-cell $\varphi : u_{\text{cod}} \circ j_1 \Rightarrow j_2 \circ u_{\text{dom}}$.

Equivalently, reinterpreting j_1 and j_2 as 2-functors from the arrow category $\mathcal{2}$ to the 2-category \mathcal{F}_v , $(u_{\text{dom}}, u_{\text{cod}}, \varphi)$ is the data of a pseudo-natural transformation.

$$\begin{array}{ccc} I_1 & \xrightarrow{j_1} & C_1 \\ u_{\text{dom}} \downarrow & \cong \varphi & \downarrow u_{\text{cod}} \\ I_2 & \xrightarrow{j_2} & C_2 \end{array}$$

Definition 3.5.8 (Morphism of relative monad). Let $u = (u_{\text{dom}}, u_{\text{cod}}, \varphi) : j_1 \rightarrow j_2$. A morphism of relative monads $\theta : m_1 \rightarrow_u m_2$ from a j_1 -relative monad m_1 to a j_2 -relative monad m_2 over u is a 2-cell $\theta : u_{\text{cod}} \circ m_1 \rightarrow m_2 \circ u_{\text{dom}}$

such that

It is not totally clear that we obtained the right definition and the following lemma shows that at the very least a property that we would expect in the classical monadic case still hold, namely that relative monad morphisms factor through the Kleisli algebras objects when those exists.⁵

⁵However, in sharp contrast with the non-relative case, it is not enough to reverse the direction of θ to obtain a notion of monad morphism factoring through the Eilenberg-Moore objects. It seems difficult to define such a notion over an arbitrary morphism u of base functors.

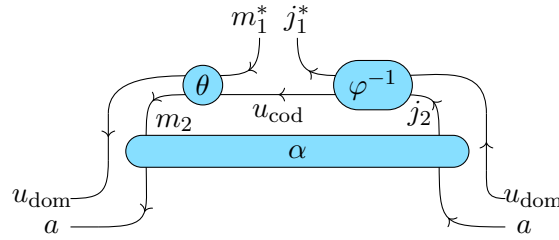
Conjecture 3.5.1. Assume the Kleisli objects C_{m_1} and C_{m_2} exists, inducing factorizations $m_1 = r_1 l_1$ and $m_2 = r_2 l_2$. Then relative monad morphisms $\theta : m_1 \rightarrow_{\mathbf{u}} m_2$ factorize as pairs of a vertical arrows $v : C_{m_1} \rightarrow C_{m_2}$ and a vertical 2-cell $\tilde{\theta} : u_{\text{cod}} r_1 \rightarrow r_2 v$ fitting in the following diagram

$$\begin{array}{ccccc} I_1 & \xrightarrow{l_1} & C_{m_1}^1 & \xrightarrow{r_1} & C_1 \\ u_{\text{dom}} \downarrow & \circlearrowleft & \downarrow v & \Downarrow \tilde{\theta} & \downarrow u_{\text{cod}} \\ I_2 & \xrightarrow{l_2} & C_{m_2}^2 & \xrightarrow{r_2} & C_2 \end{array}$$

Proof idea. Let $\theta : m_1 \rightarrow_{\mathbf{u}} m_2$ be a relative monad morphism, we define v by co-representability, so we construct a framed natural transformation

$$\nu : {}^c\mathcal{K}_{C_{m_2}^2} \cong \mathcal{Kl}_{m_2} \longrightarrow {}^c\mathcal{K}_{C_{m_1}^1} \cong \mathcal{Kl}_{m_1}.$$

Given an object X , a Kleisli algebra $(a : I_2 \rightarrow X, \alpha) \in \mathcal{Kl}_{m_2}(X)$ is sent to the Kleisli algebra $\nu_X(a, \alpha) \in \mathcal{Kl}_{m_1}(X)$ composed of $a u_{\text{dom}} : I_1 \rightarrow X$ together with the algebra structure



ν_X acts on Kleisli algebra morphisms by precomposition with u_{dom} . Given a proarrow $M : X \rightarrow Y$, ν_M sends natural transformations in $\mathcal{Kl}_{m_2}(M)((a, \alpha), (b, \beta))$ to natural transformations in $\mathcal{Kl}_{m_1}(M)(\nu_X(a, \alpha), \nu_Y(b, \beta))$ also by precomposition with u_{dom} .

In order to finish the proof we would need to exhibit $\tilde{\theta}$ but this would require us an notion of framed modification between framed natural transformation ${}^c\mathcal{K}_{\tilde{\theta}} : {}^c\mathcal{K}_{r_1} \circ {}^c\mathcal{K}_{u_{\text{cod}}} \rightarrow \nu \circ {}^c\mathcal{K}_{r_2}$ and an adequate representation theorem, which would go far beyond the purpose of this chapter. We leave this proof unfinished but note that we have a canonical candidate for this would-be ${}^c\mathcal{K}_{\tilde{\theta}}$ evaluated at a component $X \in |\mathcal{F}|$ and $f \in {}^c\mathcal{K}_{C_2}(X) = \mathcal{F}_v(C_2, X)$, namely the Kleisli algebra morphism $(f u_{\text{cod}} m_1, \text{bind}^{m_1}) \rightarrow \nu_X(f m_2, \text{bind}^{m_2}) = (f m_2 u_{\text{dom}}, \dots)$ induced by $\theta : u_{\text{cod}} m_1 \rightarrow m_2 u_{\text{dom}}$. \square

3.5.2 The 2-category of relative monads

Given a framed bicategory \mathcal{F} , we obtain a 2-category $\text{RelMon}(\mathcal{F})$ of relative monads in \mathcal{F} :

- ▷ A 0-cell $\mathbf{m} = (I, C, j, m)$ of $\text{RelMon}(\mathcal{F})$ consists of a pair of objects $I, C \in |\mathcal{F}|$, a vertical morphism $j : I \rightarrow C$ and a j -relative monad m .
- ▷ A 1-cell $\theta = (u_{\text{dom}}, u_{\text{cod}}, \varphi, \theta) : \mathbf{m}_1 \rightarrow \mathbf{m}_2$ consists of 1-cells $u_{\text{dom}} : i_1 \rightarrow i_2$, $u_{\text{cod}} : c_1 \rightarrow c_2$, an invertible 2-cells $\varphi \in {}_{u_{\text{dom}}} \mathcal{F}_{u_{\text{cod}}}(j_1, j_2)$ and a morphism of relative monad θ from m_1 to m_2 over φ .
- ▷ Finally a 2-cell $\zeta : \theta \rightarrow \theta'$, where $\theta = (u_{\text{dom}}, u_{\text{cod}}, \varphi, \theta)$, $\theta' = (v_{\text{dom}}, v_{\text{cod}}, \psi, \theta')$, is given by a pair $(\zeta_{\text{cod}}, \zeta_{\text{dom}})$ of 2-cells in \mathcal{F}

$$v_{\text{dom}} \leftarrow \zeta_{\text{dom}} \leftarrow u_{\text{dom}} \qquad v_{\text{cod}} \leftarrow \zeta_{\text{cod}} \leftarrow u_{\text{cod}}$$

such that

$$\begin{array}{c}
\begin{array}{c} v_{\text{dom}} \text{---} \zeta_{\text{dom}} \text{---} \varphi \text{---} j_1 \\ j_2 \text{---} \text{---} u_{\text{cod}} \end{array} = \begin{array}{c} v_{\text{dom}} \text{---} \psi \text{---} j_1 \\ j_2 \text{---} \text{---} \zeta_{\text{cod}} \text{---} u_{\text{cod}} \end{array} \\
\\
\begin{array}{c} v_{\text{dom}} \text{---} \zeta_{\text{dom}} \text{---} \theta \text{---} m_1 \\ m_2 \text{---} \text{---} u_{\text{cod}} \end{array} = \begin{array}{c} v_{\text{dom}} \text{---} \theta' \text{---} m_1 \\ m_2 \text{---} \text{---} \zeta_{\text{cod}} \text{---} u_{\text{cod}} \end{array}
\end{array}$$

A vertical arrow $j \in \mathcal{F}_v(I, C)$ can be alternatively seen as a (2-)functor from the category $\mathbb{2} = \bullet \rightarrow \bullet$ to the 2-category \mathcal{F}_v . This observation leads to a 2-functor $\mathcal{U} : \mathcal{R}\text{elMon}(\mathcal{F}) \rightarrow [2, \mathcal{F}_v]_{\text{ps}}$ from $\mathcal{R}\text{elMon}(\mathcal{F})$ to the functor 2-category $[2, \mathcal{F}_v]_{\text{ps}}$ of 2-functors from $\mathbb{2}$ to \mathcal{F}_v , pseudo-natural transformations and modifications. \mathcal{U} is defined by projecting out the relevant data. It has a left 2-adjoint, sending a vertical cell $j \in \mathcal{F}_v(I, C)$ to the j -relative monad j with return and bind being identities.

We will often restrict our attention to subcategories of $\mathcal{R}\text{elMon}(\mathcal{F})$ over a particular vertical arrow $j \in \mathcal{F}_v(I, C)$. We note $\mathcal{R}\text{elMon}(\mathcal{F})_j$ the full 2-subcategory of $\mathcal{R}\text{elMon}(\mathcal{F})$ on 0-cells mapped to j by \mathcal{U} and 1-cells mapped to the identity of j .

Correspondence to monads As a consistency check, we prove that relative monads in \mathcal{F} over identities are the same as monads in the 2-category \mathcal{F}_v . To do so, we first restrict $\mathcal{R}\text{elMon}(\mathcal{F})$ to the 2-category $\mathcal{R}\text{elMon}(\mathcal{F})_{\text{id}}$ of those relative monads over an identity. $\mathcal{R}\text{elMon}(\mathcal{F})_{\text{id}}$ can be built as the strict 2-pullback

$$\begin{array}{ccc}
\mathcal{R}\text{elMon}(\mathcal{F})_{\text{id}} & \dashrightarrow & \mathcal{R}\text{elMon}(\mathcal{F}) \\
\downarrow & \lrcorner & \downarrow \mathcal{U} \\
\mathcal{F}_v & \xrightarrow{\delta} & [2, \mathcal{F}_v]_{\text{ps}}
\end{array}$$

where $\delta : \mathcal{F}_v \rightarrow [2, \mathcal{F}_v]_{\text{ps}}$ is the 2-functor sending a 0-cell C to the functor corresponding to the identity $\text{id}_C : C \rightarrow C$, a 1-cell $f : C \rightarrow D$ to the pseudo-natural transformation (f, f, id_f) and a 2-cell $\nu : f \rightarrow g$ to the modification (ν, ν) .

Theorem 3.5.2. *The 2-categories $\mathcal{M}\text{nd}(\mathcal{F}_v^{\text{op}(2)})$ and $\mathcal{R}\text{elMon}(\mathcal{F})_{\text{id}}$ are isomorphic.*

Note that the direction of the 2-cells in \mathcal{F}_v has to be reversed when building the category of monads. This is because the notion of relative monad morphism we introduced corresponds to morphisms between Kleisli objects and not between Eilenberg-Moore objects.

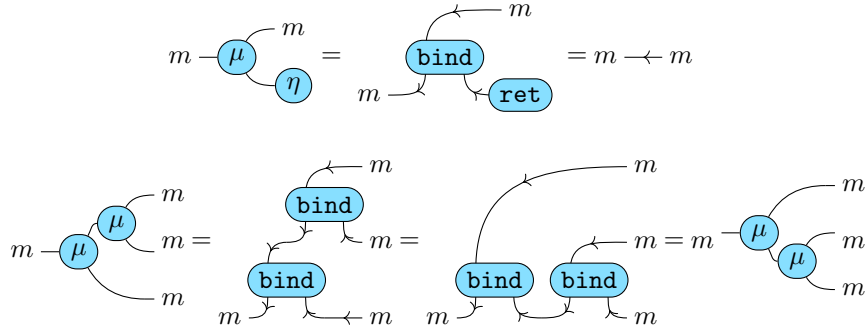
Proof. The proof is a generalization of the usual correspondence between monads and Kleisli triples, amounting to unfold the definitions and checking that everything still makes sense. We proceed by dimension of the cells.

For 0-cells. From a relative monad m over the vertical identity of an object C in \mathcal{F} , we define the following two cells in \mathcal{F}_v

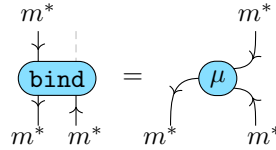
$$\begin{array}{ccc}
m \text{---} \eta = m \text{---} \text{ret} & & m \text{---} \mu = \begin{array}{c} m \text{---} \text{bind} \\ m \text{---} \end{array}
\end{array}$$

The monadic laws follow from the equations of relative monads

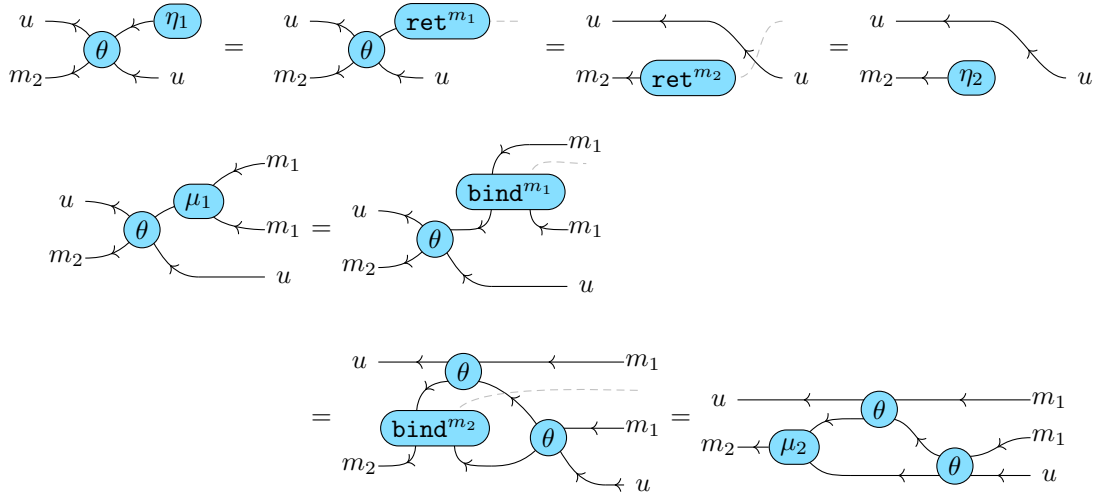
$$\begin{array}{c}
m \text{---} \mu \text{---} \eta = \begin{array}{c} \text{ret} \\ \text{bind} \end{array} = m \text{---} m
\end{array}$$



The converse, sending a monad m to a relative monad over the identity is similar, defining bind as



For 1-cells. Given a relative monad morphism $\theta = (u_{\text{dom}}, u_{\text{cod}}, \theta) : m_1 \rightarrow m_2$ between relative monads m_1 and m_2 respectively over the identity of $C_1, C_2 \in |\mathcal{F}|$, we have first a 1-cell $u = u_{\text{dom}} = u_{\text{cod}} : C_1 \rightarrow C_2$. Then the 2-cell $\theta : um_1 \rightarrow m_2u$ satisfy the simplified equations



corresponding exactly to a 1-cell in $\mathcal{Mnd}(\mathcal{F}_{\mathbf{V}}^{\text{op}(2)})$.

For 2-cells. In the same fashion, a 2-cell $\zeta = (\zeta_{\text{dom}}, \zeta_{\text{cod}}) : \theta \rightarrow \theta'$ between relative monad morphisms $\theta = (u, \theta)$ and $\theta' = (v, \theta')$ have to verify that $\zeta = \zeta_{\text{dom}} = \zeta_{\text{cod}}$ by the first equation, so that the second equation becomes $\zeta_{m_2} \circ \theta = \theta' \circ m_1 \zeta$, which is exactly the condition for a monad transformation \square

3.6 Conclusion & Related work

We briefly recalled the basic definitions of the theory of 2-categories (Bénabou, 1967) to access elements of the formal theory of monads (Kelly and Street, 1974; Lack and Street, 2002; Street, 1972), in particular the synthetic definition of a monad in a 2-category, the construction of the category of monads and the general definition of Eilenberg-Moore object (as well as Kleisli objects) as representing objects. We use a similar methodology for relative monads Altenkirch et al. (2015) in the context of framed bicategories (Shulman, 2008), defining in full generality a 2-category of relative monads and adequate notions of Eilenberg-Moore and Kleisli objects. Since this encompass enriched relative monads, the theory applies directly to specification monads. We then further prove that our extension is compatible with the formal theory of monads when restricting our attention to monads relative to identities.

We prove only modest results on relative monads in a framed bicategory, but these already demonstrate that it is possible to carry out a formal theory of relative monads at a high-level of generality and with simple proofs thanks to string diagrams. As future work, we consider extending and formalizing these results inside type theory, e.g., Coq, not only for formally (re-)proving these results, but also to use them directly to derive actual instances such as specification monads.

We now review further related work.

Other generalizations of relative monads (Fiore et al., 2018) introduce the notion of relative pseudo-monad to study the construction of category of presheaves and deal with size issues, generalizing relative monads to a bicategorical setting.

A close line of work is the study of skew-monoids: (Altenkirch et al., 2015) shows that assuming some property on a functor $\mathcal{J} : \mathcal{I} \rightarrow \mathcal{C}$, we can give to the functor category a skew-monoidal structure $\otimes_{\mathcal{J}}$ such that \mathcal{J} -relative monads coincide with skew-monoids, the adequate notion of monoids for $\otimes_{\mathcal{J}}$. The study of skew monoidal categories and related structure is a rather recent but dynamic research field (Andrianopoulos, 2017; Bourke and Lack, 2018a,b; Lack and Street, 2015; Szlachányi, 2012; Uustalu et al., 2018). In particular, skew monoids often yields construction closer in spirit to the traditional theory of monads, however important examples we consider in [chapter 6](#) do not fit immediately in that framework, whereas they do yield relative monads.

Alternative to framed bicategories We chose to work in this chapter with framed bicategories, but other choices are available such as proarrow equipments (Wood, 1982, 1985), Yoneda structures (Street and Walters, 1978) or yosegi boxes (Di Liberti and Loregian, 2019). This latter work shows that under a few hypothesis these different structures yield equivalent formal category theory. It would be interesting to understand how much of the formal theory of relative monads we sketch here could be achieved in the other settings.

Chapter 4

Mass producing monad transformers

“As a coq developer, we have no idea what we are doing[...]

PMP, CoqPL’19

We use *monad transformers* (Liang et al., 1995) to construct all kind of complex monadic objects on top of simple basic blocks: sophisticated computational monads of course, but also expressive specification monads, as well as effect observations [section 2.4](#). However, defining a monad transformer and proving that it satisfies all the expected laws requires significant effort. This is in sharp contrast with the impression that quite a few examples of monad transformers are, at least intuitively, mild generalization of naturally occurring monads, and consequently should be almost as easy to define. In this chapter, we present a methodology to reduce the definition of a monad transformer to that of a monad in an adequate metalanguage. We first review more precisely the notion of monad transformer, exploring its generalization to relative monads and then set out to define a domain specific language that we call the *Specification Metalanguage* (SM) adapted to the task of defining monad transformers. The goal of SM is to define monad transformers preserving specification monads, and we detail a few interesting point of its implementation in the Coq proof assistant. We close the chapter with a more categorical viewpoint on SM, potentially leading to future extensions.

4.1 What is a monad transformer ?

Informally, a monad transformer takes a monad as input and outputs another monad, often extended with further capabilities as demonstrated by the examples in [section 2.2](#). To be useful in practice, it must come with a way to lift computations from the original monad into the transformed monad. We expect a monad transformer to also apply to monad morphisms so that embedding between monads (so-called *subeffecting* in F^*) give rise to a monad morphism between the transformed monads. The lift has to be consistent with respect to this action on monad morphisms, resulting in the following definition.

Definition 4.1.1. A monad transformer \mathcal{T} on a category \mathcal{C} (Liang et al., 1995) consists of

- ▷ a function \mathcal{T} mapping monads $m : \mathcal{C} \rightarrow \mathcal{C}$ on \mathcal{C} to monads $\mathcal{T}m : \mathcal{C} \rightarrow \mathcal{C}$,
- ▷ equipped with a monad morphism $lift_m : m \rightarrow \mathcal{T}m$ for each monad m on \mathcal{C} ,
- ▷ assigning functorially to each monad morphism $\theta : m_1 \rightarrow m_2$ a monad morphism $\mathcal{T}\theta : \mathcal{T}m_1 \rightarrow \mathcal{T}m_2$,

$$\mathcal{T} id_m = id_{\mathcal{T}m}, \quad \mathcal{T}(\theta \cdot \theta') = \mathcal{T}\theta \cdot \mathcal{T}\theta',$$

- ▷ and such that the monad morphism lift_m is natural in m , that is for any monad morphism $\theta : m_1 \rightarrow m_2$,

$$\mathcal{T}\theta \circ \text{lift}_{m_1} = \text{lift}_{m_2} \circ \theta$$

Following L  th and Ghani (2002), we can concisely rephrase this definition by saying that

a monad transformer on \mathcal{C} is a pointed endofunctor on $\mathcal{Mnd}(\mathcal{C})$

Here, $\mathcal{Mnd}(\mathcal{C})$ is the sub-1-category of $\mathcal{Mnd}(\text{Cat})$ (see [section 3.1](#)) whose objects are sent to \mathcal{C} by the forgetful functor $U : \mathcal{Mnd}(\text{Cat}) \rightarrow \text{Cat}$ and morphisms are sent to the identity functor on \mathcal{C} . By the results of [subsection 3.5.2](#), we can equivalently see $\mathcal{Mnd}(\mathcal{C})$ as the (1-)category $\text{RelMon}(\text{Distr})_{\text{Id}_{\mathcal{C}}}$ of relative monads over the identity functor of \mathcal{C} . This observation invite us to the following generalization of monad transformers for relative monads.

Definition 4.1.2. Let \mathcal{F} be a framed category, $j \in \mathcal{F}_v(I, C)$ a vertical arrow and note $\text{RelMon}(\mathcal{F})_j$ the 1-category of j -relative monads. A j -relative monad transformer \mathcal{T} is a pointed endofunctor on $\text{RelMon}(\mathcal{F})_j$, that is

- ▷ a functor $\mathcal{T} : \text{RelMon}(\mathcal{F})_j \rightarrow \text{RelMon}(\mathcal{F})_j$
- ▷ equipped with a natural transformation $\text{lift} : \text{Id}_{\text{RelMon}(\mathcal{F})_j} \rightarrow \mathcal{T}$

To see what it means for specification monads, we unfold this definition in the case of framed bicategory of Pos -enriched categories and distributors Pos-Distr , taking as base functor $\text{Disc} : \text{Set} \rightarrow \text{Pos}$ (see [Def. 3.5.2](#) for details). Then what we could call a *specification monad transformer* \mathcal{T} consists of

- ▷ a function \mathcal{T} mapping *specification monads* $W : \text{Set} \rightarrow \text{Pos}$ to *specification monads* $\mathcal{T}W : \text{Set} \rightarrow \text{Pos}$,
- ▷ equipped with a *monotonic monad morphism* $\text{lift}_W : W \rightarrow \mathcal{T}W$ for each specification monads W ,
- ▷ acting functorially on specification monad morphisms, such lift_W is natural in W .

We will see shortly that all the examples of monad transformers in [section 2.2](#) actually lift to such specification monad transformers.

4.2 Towards a language for defining monad transformers

If we want to build a monad transformer, we could consider at a first approximation that it consists of a function taking a monad as a parameter and returning a monad. Thinking syntactically for a short while, we can describe such functions from the data of a monad in a context containing variables standing for a monad \mathbb{M} and its operations. Leveraging this simple idea, we want to design a language SM equipped with a type former $\mathbb{M}X$ for a type X standing for an abstract monad variable, such that a monad \mathbb{T} in SM naturally elaborates to a monad transformer by substituting an actual monad to the monad variable \mathbb{M} :

$$\mathcal{T}MA = \mathbb{T}[\mathbb{M}/\mathbb{M}]A \tag{4.1}$$

This language SM should build upon a base language \mathcal{L} describing the base category \mathcal{C} over which the elaborated monad transformers \mathcal{T} are defined.

What is missing in this picture for \mathcal{T} to define an actual monad transformer ? First, we need \mathcal{T} to be functorial in M . Second, we need a lifting coercion from the M to $\mathcal{T}M$. A simplistic solution for the first problem is to require all type constructors of SM to be covariant in their type arguments so that we can elaborate a functorial action for \mathcal{T} by design. For the second problem,

$$\begin{aligned}
C &::= \mathbb{M}A \mid C_1 \times C_2 \mid (x : A) \rightsquigarrow C \mid C_1 \rightarrow C_2 & A \in \text{Type}_{\mathcal{L}} \\
t &::= \text{ret} \mid \text{bind} \mid (t_1, t_2) \mid \pi_i t \mid x \mid \lambda^\diamond x. t \mid t_1 t_2 \mid \lambda x. t \mid t u & u \in \text{Term}_{\mathcal{L}}
\end{aligned}$$

Figure 4.1: Syntax of SM

we use the observation from the following lemma that a monad morphism $\text{lift} : \mathbb{M} \rightarrow \mathcal{T} \mathbb{M}$ can be equivalently provided by an \mathbb{M} -algebra structure on $\mathcal{T} \mathbb{M}$ compatible with the bind operation on $\mathcal{T} \mathbb{M}$.

Lemma 4.2.1. *Let $\mathbb{M}_1, \mathbb{M}_2$ be monads (on a category \mathcal{C}). There is a bijective correspondence between:*

1. *monad morphisms $\theta : \mathbb{M}_1 \rightarrow \mathbb{M}_2$,*
2. *liftings $L : \mathcal{C} \rightarrow \mathcal{C}^{\mathbb{M}_1}$ of \mathbb{M}_2 along $U : \mathcal{C}^{\mathbb{M}_1} \rightarrow \mathcal{C}$ the forgetful functor from the category of \mathbb{M}_1 -algebra*

$$\begin{array}{ccc}
& \mathcal{C}^{\mathbb{M}_1} & \\
L \nearrow & & \searrow U \\
\mathcal{C} & \xrightarrow{\quad \mathbb{M}_2 \quad} & \mathcal{C}
\end{array}$$

such that the multiplication of \mathbb{M}_2 lifts to an \mathbb{M}_1 -algebra morphism,

3. *for each $A \in \mathcal{C}$, assignments of \mathbb{M}_1 -algebra structure $\alpha_A : \mathbb{M}_1 \mathbb{M}_2 A \rightarrow \mathbb{M}_2 A$ such that for any $f : A \rightarrow \mathbb{M}_2 B$, $\text{bind}^{\mathbb{M}_2} f : \mathbb{M}_2 A \rightarrow \mathbb{M}_2 B$ is an \mathbb{M}_1 -algebra morphism.*

Proof. 2 and 3 can be readily seen to be in bijective correspondence by the usual correspondence between multiplication-based presentation and bind -based presentation of monads (see the proof of [Thm. 3.5.2](#)). The correspondence between 1 and 3 follows from instantiating [Lem. 4.5.2](#) to the special case of relative monads over the identity functor of \mathcal{C} in Distr . \square

Since we want such a lifting for every monad \mathbb{M} , naturally in \mathbb{M} , we should generalize the previous lemma to collection of monads, however that generalization turns out to be rather technical and we defer it to [section 4.5](#) where we develop this categorical approach in more details. The important point to keep in mind is that a type in SM should in particular be elaborated to a family consisting of \mathbb{M} -algebra for each monad \mathbb{M} in the base language \mathcal{L} (or correspondingly on the base category \mathcal{C}). This dependency on the argument monad can be naturally expressed by a dependent product and provides an important motivation for developing our language SM on top of a dependently typed language.

4.3 A DSL for specification monad transformers

In this section, we introduce the *Specification Metalanguage*, SM, and a translation from SM to correct-by-construction monad transformers in a base dependent type theory \mathcal{L} (where \mathcal{L} is a parameter of SM). More precisely, the translation takes as input a monad in SM subject to two extra conditions, *covariance* and *linearity* corresponding to the two points raised in previous subsection, and produces a specification monad transformer in \mathcal{L} .

$$\begin{array}{c}
\frac{\Delta \vdash_{\mathcal{L}}}{\Delta; \cdot \vdash_{\text{SM}}} \qquad \frac{\Delta; \Gamma \vdash_{\text{SM}} C}{\Delta; \Gamma, x : C \vdash_{\text{SM}}} \\
\\
\frac{\Delta \vdash_{\mathcal{L}} A \quad \Delta; \Gamma \vdash_{\text{SM}}}{\Delta; \Gamma \vdash_{\text{SM}} \mathbb{M}A} \qquad \frac{\Delta; \Gamma \vdash_{\text{SM}} C_1 \quad \Delta; \Gamma \vdash_{\text{SM}} C_2}{\Delta; \Gamma \vdash_{\text{SM}} C_1 \times C_2} \\
\\
\frac{\Delta, x : A; \Gamma \vdash_{\text{SM}} C}{\Delta; \Gamma \vdash_{\text{SM}} (x : A) \rightsquigarrow C} \qquad \frac{\Delta; \Gamma \vdash_{\text{SM}} C_1 \quad \Delta; \Gamma \vdash_{\text{SM}} C_2}{\Delta; \Gamma \vdash_{\text{SM}} C_1 \rightarrow C_2} \\
\\
\frac{\Delta; \Gamma \vdash_{\text{SM}} \quad A : \text{Type} \in \Delta}{\Delta; \Gamma \vdash_{\text{SM}} \text{ret} : A \rightsquigarrow \mathbb{M}A} \qquad \frac{\Delta; \Gamma \vdash_{\text{SM}} \quad A, B : \text{Type} \in \Delta}{\Delta; \Gamma \vdash_{\text{SM}} \text{bind} : \mathbb{M}A \rightarrow (A \rightsquigarrow \mathbb{M}B) \rightarrow \mathbb{M}B} \\
\\
\frac{\Delta; \Gamma \vdash_{\text{SM}} \quad x : C \in \Gamma}{\Delta; \Gamma \vdash x : C} \qquad \frac{\Delta, x : A; \Gamma \vdash_{\text{SM}} t : C}{\Delta; \Gamma \vdash_{\text{SM}} \lambda x. t : (x : A) \rightsquigarrow C} \\
\\
\frac{\Delta; \Gamma \vdash_{\mathcal{L}} u : A \quad \Delta; \Gamma \vdash_{\text{SM}} t : (x : A) \rightsquigarrow C}{\Delta \vdash_{\text{SM}} t u : C[u/x]} \qquad \frac{\Delta; \Gamma, x : C_1 \vdash_{\text{SM}} t : C_2}{\Delta; \Gamma \vdash_{\text{SM}} \lambda^\diamond x. t : C_1 \rightarrow C_2} \\
\\
\frac{\Delta; \Gamma \vdash_{\text{SM}} t_2 : C_1 \quad \Delta; \Gamma \vdash_{\text{SM}} t_1 : C_1 \rightarrow C_2}{\Delta; \Gamma \vdash_{\text{SM}} t_1 t_2 : C_2} \qquad \frac{\Delta; \Gamma \vdash_{\text{SM}} t_i : C_i}{\Delta; \Gamma \vdash_{\text{SM}} (t_1, t_2) : C_1 \times C_2} \\
\\
\frac{\Delta; \Gamma \vdash_{\text{SM}} t : C_1 \times C_2}{\Delta; \Gamma \vdash_{\text{SM}} \pi_i t : C_i}
\end{array}$$

Figure 4.2: Typing rules for SM

4.3.1 Presentation of the language SM

The design of SM, whose syntax is presented in [Figure 4.1](#), has been informed by the goal of defining monad transformers. First, since we want a mapping from monads to monads, we introduce the type constructor \mathbb{M} standing for an arbitrary base monad, as well as terms `ret` and `bind`. Second, in order to describe monads internally to SM, we add function types $(x : A) \rightarrow C[x]$ and $C_1 \rightarrow C_2$. We allow dependent function types only when the domain is in \mathcal{L} , leading to two different type formers. We write dependent abstractions as $\lambda x. t$, whereas we write the non-dependent abstraction where the domain is a type in SM as $\lambda^\diamond x. t$. In [Figure 4.2](#) we present the typing rules of SM which are mostly standard. We assume that \mathcal{L} has three judgements $\Delta \vdash_{\mathcal{L}}$, $\Delta \vdash_{\mathcal{L}} A$ and $\Delta \vdash_{\mathcal{L}} u : A$ defining respectively well-formed contexts in \mathcal{L} (that we will always note Δ), well-formed types in a context (noted A, B) and well-typed terms. We also assume that \mathcal{L} has at least one universe `Type` that we use for dependent products. We then define the judgements $\Delta; \Gamma \vdash_{\text{SM}}$ for well-formed contexts of SM, $\Delta; \Gamma \vdash_{\text{SM}} C$ for well-formed types and $\Delta; \Gamma \vdash_{\text{SM}} t : C$ for well-formed terms. We implicitly assume a conversion rule with respect to convertibility in \mathcal{L} ¹. The main rules of the equational theory of SM are given in [Figure 4.3](#). SM is expressive to define many different monads in a natural way using structure of the underlying dependent type theory \mathcal{L} ², for example

1. *reader* $\mathbb{R}d(X : \text{Type}) = \mathcal{I} \rightsquigarrow \mathbb{M}X$;

¹Because of our implementation choices, no conversion rule appear explicitly in our implementation in Coq, see [section 4.4](#)

²We assume for these examples that \mathcal{L} has (dependent) sums and products

$$\begin{array}{c}
\frac{\Delta \vdash_{\mathcal{L}} a : A \quad \Delta; \Gamma \vdash_{\text{SM}} f : A \rightsquigarrow \mathbb{M} B}{\Delta; \Gamma \vdash_{\text{SM}} \text{bind}(\text{ret } a) f \equiv f a : \mathbb{M} B} \quad \frac{\Delta; \Gamma \vdash_{\text{SM}} m : \mathbb{M} A}{\Delta; \Gamma \vdash_{\text{SM}} \text{bind } m \text{ ret} \equiv m : \mathbb{M} A} \\
\\
\frac{\Delta; \Gamma \vdash_{\text{SM}} m : \mathbb{M} A_1 \quad \Delta; \Gamma \vdash_{\text{SM}} f : A_1 \rightsquigarrow \mathbb{M} A_2 \quad \Delta; \Gamma \vdash_{\text{SM}} g : A_2 \rightsquigarrow \mathbb{M} A_3}{\Delta; \Gamma \vdash_{\text{SM}} \text{bind } m (\lambda x. \text{bind} (f x) g) \equiv \text{bind} (\text{bind } m f) g : \mathbb{M} A_3} \\
\\
\frac{\Delta; \Gamma \vdash_{\text{SM}} t_1 : C_1 \quad \Delta; \Gamma \vdash_{\text{SM}} t_2 : C_2}{\Delta; \Gamma \vdash_{\text{SM}} \pi_i (t_1, t_2) \equiv t_i : \mathbb{M} C_i} \quad \frac{\Delta; \Gamma \vdash_{\text{SM}} t : C_1 \times C_2}{\Delta; \Gamma \vdash_{\text{SM}} (\pi_1 t, \pi_2 t) \equiv t : C_1 \times C_2} \\
\\
\frac{\Delta, x : A; \Gamma \vdash_{\text{SM}} t : C \quad \Delta \vdash_{\mathcal{L}} u : A}{\Delta; \Gamma \vdash_{\text{SM}} (\lambda x. t) u \equiv t\{u/x\} : C} \quad \frac{\Delta; \Gamma \vdash_{\text{SM}} t : (x : A) \rightsquigarrow C}{\Delta; \Gamma \vdash_{\text{SM}} \lambda x. t x \equiv t : (x : A) \rightsquigarrow C} \\
\\
\frac{\Delta; \Gamma, x : C_1 \vdash_{\text{SM}} t_1 : C_2 \quad \Delta; \Gamma \vdash_{\text{SM}} t_2 : C_1}{\Delta; \Gamma \vdash_{\text{SM}} (\lambda^\diamond x. t_1) t_2 \equiv t_1\{t_2/x\} : C_2} \quad \frac{\Delta; \Gamma \vdash_{\text{SM}} t : C_1 \rightarrow C_2}{\Delta; \Gamma \vdash_{\text{SM}} \lambda^\diamond x. t x \equiv t : C_1 \rightarrow C_2} \\
\\
\frac{\Delta; \Gamma \vdash_{\text{SM}} t_1 \equiv t_2 : (x : A) \rightsquigarrow C \quad \Delta \vdash_{\mathcal{L}} u_1 \equiv u_2 : A}{\Delta; \Gamma \vdash_{\text{SM}} t_1 \equiv t_2 : C[u_1/x]}
\end{array}$$

+ reflexivity, symmetry, transitivity and congruence for all other term constructors

Figure 4.3: Equational theory of SM

2. *writer* $\mathbb{W}\mathbf{r}(X : \text{Type}) = \mathbb{M}(X \times \mathcal{O})$;
3. *exceptions* $\mathbb{E}\mathbf{x}\mathbf{c}(X : \text{Type}) = \mathbb{M}(X + \mathcal{E})$;
4. *state* $\mathbb{S}\mathbf{t}(X : \text{Type}) = \mathcal{S} \rightsquigarrow \mathbb{M}(X \times \mathcal{S})$;
5. *monotonic state* $\mathbb{M}\mathbf{on}\mathbb{S}\mathbf{t}(X) = (s_0 : \mathcal{S}) \rightsquigarrow \mathbb{M}(X \times (s_1 : \mathcal{S}) \times s_0 \preceq s_1)$, where \preceq is some preorder on states \mathcal{S} ; and
6. *continuations* $\mathbb{C}\mathbf{on}\mathbf{t}_{\mathcal{A}\mathbf{ns}}(X) = (X \rightsquigarrow \mathbb{M} \mathcal{A}\mathbf{ns}) \rightarrow \mathbb{M} \mathcal{A}\mathbf{ns}$.

The *covariance* condition states that the symbol \mathbb{M} standing for an arbitrary base monad appears only in the codomain of arrows. The more involved *linearity* condition concerns the `bind` of these monads. With the exception of continuations (see [subsection 4.3.5](#)), all these SM monads satisfy these extra conditions and thus lead to proper monad transformers, in particular all examples from [section 2.2](#) can be obtained from a definition in SM.

4.3.2 Elaborating specification monads and lift

To define a monad transformer, we use monads *internal to* SM, given by

- ▷ a type constructor $X : \text{Type}; \cdot \vdash_{\text{SM}} C[X]$;
- ▷ terms

$$\begin{array}{c}
A : \text{Type}; \cdot \vdash_{\text{SM}} \text{ret}^C : A \rightarrow C[A] \\
A, B : \text{Type}; \cdot \vdash_{\text{SM}} \text{bind}^C : (A \rightarrow C[B]) \rightarrow C[A] \rightarrow C[B];
\end{array}$$

- ▷ such that the monadic laws (see [Def. 2.1.1](#)) are derivable in the equational theory of SM.

$$\begin{aligned}
\llbracket \mathbb{M}A \rrbracket_M &= \mathbb{M}A \\
\llbracket C_1 \times C_2 \rrbracket_M &= \llbracket C_1 \rrbracket_M \times \llbracket C_2 \rrbracket_M \\
\llbracket (x : A) \rightarrow C \rrbracket_M &= (x : A) \rightsquigarrow \llbracket C \rrbracket_M \\
\llbracket C_1 \rightarrow C_2 \rrbracket_M &= (f : \llbracket C_1 \rrbracket_M \rightarrow \llbracket C_2 \rrbracket_M) \times (\forall (m_1 \leq^{C_1} m'_1). f m_1 \leq^{C_2} f m'_1) \\
\\
m \leq^{\mathbb{M}A} m' &= m \leq_A^{\mathbb{M}} m' \\
(m_1, m_2) \leq^{C_1 \times C_2} (m'_1, m'_2) &= m_1 \leq^{C_1} m'_1 \wedge m_2 \leq^{C_2} m'_2 \\
f \leq^{(x:A) \rightsquigarrow C[x]} f' &= \forall (x : A). f x \leq^{C[x]} f' x \\
f \leq^{C_1 \rightarrow C_2} f' &= \forall (m_1 : \llbracket C_1 \rrbracket_M). f m_1 \leq^{C_2} f' m_1
\end{aligned}$$

Figure 4.4: Elaboration of types from SM to \mathcal{L}

$$\begin{aligned}
\llbracket \mathbf{ret} \rrbracket_M^{\delta;\gamma} &= \mathbf{ret}^{\mathbb{M}} & \llbracket \mathbf{bind} \rrbracket_M^{\delta;\gamma} &= \mathbf{bind}^{\mathbb{M}} \\
\llbracket (t_1, t_2) \rrbracket_M^{\delta;\gamma} &= (\llbracket t_1 \rrbracket_M^{\delta;\gamma}, \llbracket t_2 \rrbracket_M^{\delta;\gamma}) & \llbracket \pi_i t \rrbracket_M^{\delta;\gamma} &= \pi_i \llbracket t \rrbracket_M^{\delta;\gamma} \\
\llbracket x \rrbracket_M^{\delta;\gamma} &= \gamma(x) & \llbracket \lambda^\diamond x^{C_1}. t \rrbracket_M &= \lambda x \llbracket C_1 \rrbracket_M. \llbracket t \rrbracket_M^{\delta;\gamma[x:=x]} \\
\llbracket t_1 t_2 \rrbracket_M^{\delta;\gamma} &= \llbracket t_1 \rrbracket_M^{\delta;\gamma} \llbracket t_2 \rrbracket_M^{\delta;\gamma} & \llbracket \lambda x^A. t \rrbracket_M^{\delta;\gamma} &= \lambda x^{A[\delta]}. \llbracket t \rrbracket_M^{\delta[x:=x];\gamma} \\
\llbracket t u \rrbracket_M^{\delta;\gamma} &= \llbracket t \rrbracket_M^{\delta;\gamma} u[\delta]
\end{aligned}$$

Figure 4.5: Denotation of SM terms

Now, given a monad C internal to SM, we want to define the corresponding monad transformer \mathcal{T}^C evaluated at a monad \mathbb{M} in the base language \mathcal{L} , essentially as the substitution of \mathbb{M} for \mathbb{M} (Equation 4.1). In order to make this statement precise, we define in Figure 4.4 a denotation $\llbracket - \rrbracket_M$ of SM types as types in \mathcal{L} equipped with an order parametrized by a specification monad \mathbb{M} . In the base case $C = \mathbb{M}A$, the order is given by the specification monad \mathbb{M} , whereas the order is given pointwise in all the other cases. The only surprise happens in the case $C = C_1 \rightarrow C_2$ where we restrict the denotation to functions monotonic with respect to the orders on C_1 and C_2 . This restriction is needed to ensure the monotonicity of the denotation of terms (Thm. 4.3.1).

The denotation of terms – or rather of typing derivations for terms – is presented in Figure 4.5. Given a derivation $\Delta; \Gamma \vdash_{\text{SM}} t : C$ and substitutions $\delta : \Delta, \gamma : \llbracket \Gamma[\delta/\Delta] \rrbracket_M$, we write $\llbracket t \rrbracket_M^{\delta;\gamma} : \llbracket C \rrbracket_M$ for the denotation of the term t in \mathcal{L} .

Provided that \mathcal{L} has extensional dependent products and pairs, meaning that surjective pairing and functional extensionality are valid in \mathcal{L} , this denotation preserves the equational theory of SM and produces monotonic terms in the following sense:

Theorem 4.3.1 (Monotonicity of denotation). *Let \mathbb{M} be an ordered monad, $\Delta; \Gamma \vdash_{\text{SM}} t : C$ a term in SM, $\vdash_{\mathcal{L}} \delta : \Delta$ a substitution for the \mathcal{L} context Δ , $(\vdash_{\mathcal{L}} \gamma_i : \llbracket \Gamma \rrbracket_M)_{i=1,2}$ substitutions for the SM context Γ such that $\forall (x : C_0) \in \Gamma. \gamma_1(x) \leq^{C_0} \gamma_2(x)$. Then $\llbracket t \rrbracket_M^{\delta;\gamma_1} \leq^C \llbracket t \rrbracket_M^{\delta;\gamma_2}$.*

The proof of preservation of the equational theory is a long but rather straightforward induction on the derivation of an equality between SM terms. We only reproduce here the proof of monotonicity.

Proof. By induction on the typing derivation of t :

Case $t = \mathbf{ret}_A : A \rightsquigarrow \mathbb{M}A$, by reflexivity

$$\llbracket \mathbf{ret}_A \rrbracket_M^{\delta;\gamma_1} = \mathbf{ret}_A^{\mathbb{M}} \leq^{A \rightsquigarrow \mathbb{M}A} \mathbf{ret}_A^{\mathbb{M}} = \llbracket \mathbf{ret}_A \rrbracket_M^{\delta;\gamma_2}$$

Case $t = \text{bind}_{A,B} : (A \rightsquigarrow \mathbb{M}B) \rightarrow (\mathbb{M}A \rightarrow \mathbb{M}B)$, by reflexivity, that holds because bind^M is monotonic

$$\llbracket \text{bind}_{A,B} \rrbracket_M^{\delta;\gamma_1} = \text{bind}_{A,B}^M \leq^{(A \rightsquigarrow \mathbb{M}B) \rightarrow (\mathbb{M}A \rightarrow \mathbb{M}B)} \text{bind}_{A,B}^M = \llbracket \text{bind}_{A,B} \rrbracket_M^{\delta;\gamma_2}$$

Case $t = (t_1, t_2) : A \times B$, by induction hypothesis

$$\llbracket t_1 \rrbracket_M^{\delta;\gamma_1} \leq^A \llbracket t_1 \rrbracket_M^{\delta;\gamma_2} \quad \llbracket t_2 \rrbracket_M^{\delta;\gamma_1} \leq^B \llbracket t_2 \rrbracket_M^{\delta;\gamma_2}$$

so

$$\llbracket (t_1, t_2) \rrbracket_M^{\delta;\gamma_1} = \left(\llbracket t_1 \rrbracket_M^{\delta;\gamma_1}, \llbracket t_2 \rrbracket_M^{\delta;\gamma_1} \right) \leq^{A \times B} \left(\llbracket t_1 \rrbracket_M^{\delta;\gamma_2}, \llbracket t_2 \rrbracket_M^{\delta;\gamma_2} \right) = \llbracket (t_1, t_2) \rrbracket_M^{\delta;\gamma_2}$$

Case $t = \pi_i t' : A_i$, by induction hypothesis and extensionality

$$\left(\pi_1 \llbracket t' \rrbracket_M^{\delta;\gamma_1}, \pi_2 \llbracket t' \rrbracket_M^{\delta;\gamma_1} \right) = \llbracket t' \rrbracket_M^{\delta;\gamma_1} \leq^{A_1 \times A_2} \llbracket t' \rrbracket_M^{\delta;\gamma_2} = \left(\pi_1 \llbracket t' \rrbracket_M^{\delta;\gamma_2}, \pi_2 \llbracket t' \rrbracket_M^{\delta;\gamma_2} \right)$$

so

$$\pi_i \llbracket t' \rrbracket_M^{\delta;\gamma_1} \leq^{A_i} \pi_i \llbracket t' \rrbracket_M^{\delta;\gamma_2}$$

Case $t = \lambda x. t : (x : A) \rightsquigarrow C$, by induction hypothesis for any $v : A$,

$$\llbracket t' \rrbracket_M^{\delta[x:=v];\gamma_1} \leq^{C[v/x]} \llbracket t' \rrbracket_M^{\delta[x:=v];\gamma_2}$$

we conclude by reduction since

$$\llbracket \lambda x. t' \rrbracket_M^{\delta;\gamma_1} v = (\lambda y. \llbracket t' \rrbracket_M^{\delta[x:=y];\gamma_1}) v = \llbracket t' \rrbracket_M^{\delta[x:=v];\gamma_1}$$

Case $t = t' v : C\{v/x\}$, by induction hypothesis

$$\forall v_0 : A. \llbracket t' \rrbracket_M^{\delta;\gamma_1} v_0 \leq^{C[v_0/x]} \llbracket t' \rrbracket_M^{\delta;\gamma_2} v_0$$

so

$$\llbracket t' v \rrbracket_M^{\delta;\gamma_1} = \llbracket t' \rrbracket_M^{\delta;\gamma_1} v \leq^{C[v/x]} \llbracket t' \rrbracket_M^{\delta;\gamma_2} v = \llbracket t' v \rrbracket_M^{\delta;\gamma_2}$$

Case $t = \lambda^\diamond x. t' : C_1 \rightarrow C_2$, for any $m_1 \leq^{C_1} m_2$, $\gamma_1[x := m_1] \leq^{\Gamma, x:C_1} \gamma_1[x := m_2]$ and by induction

$$\llbracket t' \rrbracket_M^{\delta;\gamma_1[x:=m_1]} \leq^{C_2} \llbracket t' \rrbracket_M^{\delta;\gamma_2[x:=m_2]}$$

and we conclude since for $i = 1, 2$

$$(\llbracket \lambda^\diamond x. t' \rrbracket_M^{\delta;\gamma_i[x:=y]}) m_i = (\lambda y. \llbracket t' \rrbracket_M^{\delta;\gamma_i[x:=y]}) m_i = \llbracket t' \rrbracket_M^{\delta;\gamma_i[x:=m_i]}$$

Case $t = t_1 t_2 : C_2$, by induction hypothesis applied to $t_2 : C_1$,

$$\llbracket t_2 \rrbracket_M^{\delta;\gamma_1} \leq^{C_1} \llbracket t_2 \rrbracket_M^{\delta;\gamma_2}$$

so by induction hypothesis applied to $t_1 : C_1 \rightarrow C_2$

$$\llbracket t_1 \rrbracket_M^{\delta;\gamma_1} \llbracket t_2 \rrbracket_M^{\delta;\gamma_1} \leq^{C_2} \llbracket t_1 \rrbracket_M^{\delta;\gamma_2} \llbracket t_2 \rrbracket_M^{\delta;\gamma_2}$$

From these results, we deduce that a monad C internal to \mathcal{SM} induces the following mapping from specification monads to specification monads:

$$\mathcal{T}^C : (\mathcal{M}, \text{ret}, \text{bind}) \longmapsto (\llbracket C \rrbracket_{\mathcal{M}}, \llbracket \text{ret}^C \rrbracket_{\mathcal{M}}, \llbracket \text{bind}^C \rrbracket_{\mathcal{M}})$$

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

To build the `lift` for \mathcal{T}^C , we adapt [Lem. 4.2.1](#) to the current setting. The denotation $\llbracket C \rrbracket_{\mathcal{M}}$ of an SM type C in \mathcal{L} is by design canonically endowed with an \mathcal{M} -algebra structure $\alpha_{\mathcal{M}}^C : \mathcal{M}[\llbracket C \rrbracket_{\mathcal{M}}] \rightarrow \llbracket C \rrbracket_{\mathcal{M}}$. This \mathcal{M} -algebra structure is defined by induction on the structure of the SM type C , using the free algebra when $C = \mathbb{M}A$ and the algebra defined pointwise in all the other cases. Inspecting the proof of the lemma (or rather of [Lem. 4.5.2](#)), we obtain that this \mathcal{M} -algebra structure induces a lifting function from the monad \mathcal{M} to the monad $\llbracket C \rrbracket_{\mathcal{M}}$ as follows:

$$\text{lift}_{\mathcal{M}, X}^C : \mathcal{M}(X) \xrightarrow{\mathcal{M}(\text{ret}^{\llbracket C \rrbracket_{\mathcal{M}}})} \mathcal{M}[\llbracket C \rrbracket_{\mathcal{M}}](X) \xrightarrow{\alpha_{\mathcal{M}, X}^C} \llbracket C \rrbracket_{\mathcal{M}}(X) = \mathcal{T}^C \mathcal{M}(X)$$

For instance, for the state transformer, the lift functions is given by

$$\text{lift}_{\mathcal{M}, X}^{\mathbb{S}\mathbb{t}}(m : \mathcal{M} X) = \lambda(s : \mathcal{S}). \mathcal{M}(\lambda(x : X). (x, s)) m : \mathcal{S} \rightarrow \mathcal{M}(X \times \mathcal{S}).$$

The result that SM type formers are automatically equipped with an algebra structure explains why SM features products, but not sums since the latter cannot be equipped with an algebra structure in general.

4.3.3 Elaborating the action on monad morphism

To define a monad transformer, we still need to build a functorial action mapping monad morphism $\theta : \mathcal{M}_1 \rightarrow \mathcal{M}_2$ between monads $\mathcal{M}_1, \mathcal{M}_2$ in \mathcal{L} to a monad morphism $\llbracket C \rrbracket_{\mathcal{M}_1} \rightarrow \llbracket C \rrbracket_{\mathcal{M}_2}$. However, the denotation of the arrow $C_1 \rightarrow C_2$ does not allow for such a functorial action since C_1 necessarily contains a subterm \mathbb{M} in a contravariant position. In order to get an action on monad morphisms, we first build a (logical) relation between the denotations. Given $\mathcal{M}_1, \mathcal{M}_2$ monads in \mathcal{L} and a family of relations $\mathcal{R}_A \subset \mathcal{M}_1 A \times \mathcal{M}_2 A$ indexed by types A , we build a relation $\{\llbracket C \rrbracket_{\mathcal{M}_1, \mathcal{M}_2}^{\mathcal{R}} \subset \llbracket C \rrbracket_{\mathcal{M}_1} \times \llbracket C \rrbracket_{\mathcal{M}_2}$ as follows

$$\begin{aligned} m_1 \{\llbracket \mathbb{M}A \rrbracket\} m_2 &= m_1 \mathcal{R}_A m_2 \\ (m_1, m'_1) \{\llbracket C_1 \times C_2 \rrbracket\} (m_2, m'_2) &= m_1 \{\llbracket C_1 \rrbracket\} m_2 \wedge m'_1 \{\llbracket C_2 \rrbracket\} m'_2 \\ f_1 \{\llbracket (x : A) \rightarrow C \rrbracket\} f_2 &= \forall (x : A). f_1 x \{\llbracket C x \rrbracket\} f_2 x \\ f_1 \{\llbracket C_1 \rightarrow C_2 \rrbracket\} f_2 &= \forall m_1 m_2. m_1 \{\llbracket C_1 \rrbracket\} m_2 \Rightarrow f_1 m_1 \{\llbracket C_2 \rrbracket\} f_2 m_2 \end{aligned}$$

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

Definition 4.3.1 (Monadic relation). A monadic relation³ $\mathcal{R} : \mathcal{M}_1 \leftrightarrow \mathcal{M}_2$ between monads \mathcal{M}_1 and \mathcal{M}_2 , consists of:

- ▷ a family of relations $\mathcal{R}_A : \mathcal{M}_1 A \times \mathcal{M}_2 A \rightarrow \mathbb{P}$ indexed by types A ,
- ▷ such that returned values are related $(\text{ret}^{\mathcal{M}_1} v) \mathcal{R}_A (\text{ret}^{\mathcal{M}_2} v)$ for any value $v : A$,

³To our knowledge there is no general definition of this notion; the idea underlying the definition of a relation compatible with an algebraic structure is however a recurrent one and particularly well-explained in ([Hermida et al., 2014](#))

▷ and such that sequencing of related computations is related

$$\frac{m_1 \mathcal{R}_A m_2 \quad \forall (x : A). (f_1 x) \mathcal{R}_B (f_2 x)}{(\text{bind}^{M_1} m_1 f_1) \mathcal{R}_B (\text{bind}^{M_2} m_2 f_2)}$$

for any $m_1 : M_1 A, m_2 : M_2 A, f_1 : A \rightarrow M_1 B, f_2 : A \rightarrow M_2 B$.

If moreover M_1, M_2 are specification monads, we say that \mathcal{R} is monotonic when it is compatible with the orders

$$\forall A (m_1 \leq_A^{M_1} m'_1) (m_2 \leq_A^{M_2} m'_2). \quad m_1 \mathcal{R}_A m_2 \Rightarrow m'_1 \mathcal{R}_A m'_2.$$

that is each \mathcal{R}_A is an ideal of $M_1 A \times M_2 A$.

The simplest example of monadic relation is the graph of a monad morphism $\theta : M \rightarrow W$. In the frequent case where M is a computational monad and W is a specification monad, we can consider M as equipped with a discrete order and θ induces a monotonic relation \mathcal{R}^θ defined as $m \mathcal{R}^\theta w \iff \theta m \leq^W w$. Given a monadic relation, we extend the relational translation to terms and obtain the so-called fundamental lemma of logical relations.

Theorem 4.3.2 (Fundamental lemma of logical relations). *For any monads M_1, M_2 in \mathcal{L} , monadic relation $\mathcal{R} : M_1 \leftrightarrow M_2$, term $\Gamma \vdash_{SM} t : C$ and substitutions $\gamma_1 : \llbracket \Gamma \rrbracket_{M_1}$ and $\gamma_2 : \llbracket \Gamma \rrbracket_{M_2}$, if for all $(x : C') \in \Gamma, \gamma_1(x) \llbracket C' \rrbracket_{M_1, M_2}^{\mathcal{R}} \gamma_2(x)$ then $\llbracket t \rrbracket_{M_1}^{\gamma_1} \llbracket C \rrbracket_{M_1, M_2}^{\mathcal{R}} \llbracket t \rrbracket_{M_2}^{\gamma_2}$.*

Moreover this relational interpretation preserves the order induced by the input specification monad.

Theorem 4.3.3 (Monotonicity of relational interpretation). *Let $\Delta \vdash_{SM} C$ type, M_1, M_2 two specification monads and $(\mathcal{R}_A)_A$ a family of monotonic relations $\mathcal{R}_A : M_1 A \times M_2 A \rightarrow \mathbb{P}$, then $\llbracket C \rrbracket_{M_1, M_2}^{\mathcal{R}}$ is monotonic.*

Proof. by induction on the derivation of C :

Case $C = M A$, $\llbracket M A \rrbracket_{M_1, M_2}^{\mathcal{R}} = \mathcal{R}_A$ is monotonic by assumption

Case $C = C_1 \times C_2$, suppose $(m_1, m_2) \llbracket C_1 \times C_2 \rrbracket_{M_1, M_2}^{\mathcal{R}} (n_1, n_2), (m_1, m_2) \leq^{C_1 \times C_2} (m'_1, m'_2), (n_1, n_2) \leq^{C_1 \times C_2} (n'_1, n'_2)$ then by induction hypothesis $m'_1 \llbracket C_1 \rrbracket_{M_1, M_2}^{\mathcal{R}} n'_1$ and $m'_2 \llbracket C_2 \rrbracket_{M_1, M_2}^{\mathcal{R}} n'_2$ so $(m'_1, m'_2) \llbracket C_1 \times C_2 \rrbracket_{M_1, M_2}^{\mathcal{R}} (n'_1, n'_2)$

Case $C = (x : A) \rightarrow C'$, suppose $f \llbracket (x : A) \rightarrow C' \rrbracket_{M_1, M_2}^{\mathcal{R}} g, f \leq^{(x:A) \rightarrow C'} f'$ and $g \leq^{(x:A) \rightarrow C'} g'$ then for any $v : A, (f v) \llbracket C' \{v/x\} \rrbracket_{M_1, M_2}^{\mathcal{R}} (g v), f v \leq^{C' \{v/x\}} f' v, g v \leq^{C' \{v/x\}} g' v$ so by inductive hypothesis $(f' v) \llbracket C' \{v/x\} \rrbracket_{M_1, M_2}^{\mathcal{R}} (g' v)$, hence $f' \llbracket (x : A) \rightarrow C' \rrbracket_{M_1, M_2}^{\mathcal{R}} g'$

Case $C = C_1 \rightarrow C_2$, suppose $f \llbracket C_1 \rightarrow C_2 \rrbracket_{M_1, M_2}^{\mathcal{R}} g, f \leq^{C_1 \rightarrow C_2} f'$ and $g \leq^{C_1 \rightarrow C_2} g'$, for any $m \llbracket C_1 \rrbracket_{M_1, M_2}^{\mathcal{R}} n, (f m) \llbracket C_2 \rrbracket_{M_1, M_2}^{\mathcal{R}} (g n), m \leq^{C_1} m$ and $n \leq^{C_2} n$ so $f m \leq^{C_2} f' m$ and $g n \leq^{C_2} g' n$, hence by induction hypothesis $(f' m) \llbracket C_2 \rrbracket_{M_1, M_2}^{\mathcal{R}} (g' n)$

✎

As a corollary, an internal monad C in SM induces a mapping from (monotonic) monadic relations to (monotonic) monadic relations, the relational interpretation of ret^C and bind^C providing witnesses to the preservation of the monadic structure. In particular, any monad morphism $\theta : M_1 \rightarrow M_2$ defines a monadic relation $\llbracket C \rrbracket_{M_1, M_2}^{\mathcal{R}^\theta} : \llbracket C \rrbracket_{M_1} \leftrightarrow \llbracket C \rrbracket_{M_2}$. It turns out that if C is moreover *covariant*, meaning that it does not contain any occurrence of an arrow $C_1 \rightarrow C_2$ where C_1 is a type in SM , then the relational denotation $\llbracket C \rrbracket_{M_1, M_2}^{\mathcal{R}^\theta}$ with respect to any monad morphism $\theta : M_1 \rightarrow M_2$ is actually the graph of a monad morphism.

The last missing bit in order to obtain a specification monad transformer out of C is to prove that the elaborated $\text{lift}_M^C : M \rightarrow \llbracket C \rrbracket_M$ is *natural*, that is, that the following diagram should commute:

$$\begin{array}{ccccc}
 MA & \xrightarrow{M(\text{ret}_A^C)} & M\llbracket C \rrbracket_M A & \xrightarrow{\alpha_{M,A}^C} & \llbracket C \rrbracket_M A \\
 Mf \downarrow & \circlearrowleft & \downarrow M\llbracket C \rrbracket_M f & ? & \downarrow \llbracket C \rrbracket_M f \\
 MB & \xrightarrow{M(\text{ret}_B^C)} & M\llbracket C \rrbracket_M B & \xrightarrow{\alpha_{M,B}^C} & \llbracket C \rrbracket_M B
 \end{array}$$

for any A, B and $f : A \rightarrow B$. Observe that the left square commutes automatically by the naturality of $M(\text{ret}^C)$. However, for the right square to commute $\llbracket C \rrbracket_M f = \text{bind}^{\llbracket C \rrbracket_M}(\text{ret}^{\llbracket C \rrbracket_M} \circ f)$ needs to be an M -algebra homomorphism, which is exactly the condition required by [Lem. 4.2.1](#). The next section explains how to capture this semantic condition syntactically using a linear type system. We call that syntactic condition on the monad $(C, \text{ret}^C, \text{bind}^C)$ internal to SM *the linearity of bind^C* .

To summarize the results of our approach language-base approach to monad transformers, we have:

Theorem 4.3.4 (Construction of monad transformer from SM). *Given a monad C internal to SM such that bind^C satisfies the linearity criterion, we obtain:*

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

4.3.4 Linear type system for SM

In this section, we elaborate a simplistic syntactic criterion on a monad C internal to SM ensuring the semantic condition that bind^C maps functions to M -algebra homomorphisms. To do so, we recast the homomorphism condition as a linearity condition in a modified type system for SM equipped with a *stoup*: a distinguished variable in the context such that the term typed in the judgement is linear with respect to that variable ([Egger et al., 2014](#); [Munch-Maccagnoni, 2013](#)). We note such distinguished contexts $\Gamma \mid \Xi$ where Γ is a normal SM context and Ξ is the stoup. The stoup can be either empty or containing one variable of a type C from SM. Linear types are a refinements of types from SM given by the following grammar

$$L := C \mid C_1 \multimap C_2 \mid L_1 \times L_2 \mid (x : A) \rightarrow L \mid L_1 \rightarrow L_2$$

where $A \in \text{Type}_{\mathcal{L}}$, $C, C_1, C_2 \in \text{Type}_{\text{SM}}$. In particular the linear function space $C_1 \multimap C_2$ should be understood as a subtype of $C_1 \rightarrow C_2$ whose denotation ought to be a set of homomorphisms with respect to the algebra structures on the denotations of its domain and codomain, thus cannot be nested. A linear judgement is of the form $\Delta; \Gamma \mid \Xi \vdash_{\text{lin}} t : L$ with the invariant that if Ξ is non-empty then $\Xi = x : C_1$ and $L = C_2$ for SM types $\vdash_{\text{SM}} C_1$ and $\vdash_{\text{SM}} C_2$.

The following theorem explains the aim of the linear type system:

Theorem 4.3.5 (linear terms are homomorphisms). *Let M be a monad, $\Gamma \mid - \vdash_{\text{lin}} t : C_1 \multimap C_2$ a term in SM and $\gamma : \llbracket \Gamma \rrbracket_M$, then the following diagram commutes*

$$\begin{array}{ccc}
 M\llbracket C_1 \rrbracket_M & \xrightarrow{\alpha_{M,C_1}^C} & \llbracket C_1 \rrbracket_M \\
 \downarrow M\llbracket t \rrbracket_M^\gamma & & \downarrow \llbracket t \rrbracket_M^\gamma \\
 M\llbracket C_2 \rrbracket_M & \xrightarrow{\alpha_{M,C_2}^C} & \llbracket C_2 \rrbracket_M
 \end{array}$$

$$\begin{array}{c}
\frac{}{A \mid - \vdash_{\text{lin}} \mathbf{ret} : A \rightarrow \mathbb{M}A} \quad \frac{}{A, B \mid - \vdash_{\text{lin}} \mathbf{bind} : \mathbb{M}A \multimap (A \rightarrow \mathbb{M}B) \rightarrow \mathbb{M}B} \\
\\
\frac{}{\Gamma \mid x : C \vdash x : C} \quad \frac{(x : C) \in \Gamma}{\Gamma \mid - \vdash x : C} \quad \frac{\Gamma \mid \Xi \vdash_{\text{lin}} t_i : C_i}{\Gamma \mid \Xi \vdash_{\text{lin}} (t_1, t_2) : C_1 \times C_2} \\
\\
\frac{\Gamma \mid \Xi \vdash_{\text{lin}} t : C_1 \times C_2}{\Gamma \mid \Xi \vdash_{\text{lin}} \pi_i t : C_i} \quad \frac{\Gamma, x : A \mid \Xi \vdash_{\text{lin}} t : C}{\Gamma \mid \Xi \vdash_{\text{SM}} \lambda. xt : (x : A) \rightarrow C} \\
\\
\frac{\Gamma \mid \Xi \vdash_{\mathcal{L}} u : A \quad \Gamma \mid \Xi \vdash_{\text{lin}} t : (x : A) \rightarrow C}{\Gamma \vdash_{\text{lin}} t u : C[u/x]} \quad \frac{\Gamma \mid x : C_1 \vdash_{\text{lin}} t : C_2}{\Gamma \mid - \vdash_{\text{lin}} \lambda^\diamond x. t : C_1 \multimap C_2} \\
\\
\frac{\Gamma, x : C_1 \mid \Xi \vdash_{\text{lin}} t : C_2}{\Gamma \mid \Xi \vdash_{\text{lin}} \lambda^\diamond x. t : C_1 \rightarrow C_2} \quad \frac{\Gamma \mid - \vdash_{\text{lin}} t : C_1 \multimap C_2}{\Gamma \mid - \vdash_{\text{lin}} t : C_1 \rightarrow C_2} \\
\\
\frac{\Gamma \mid \Xi \vdash_{\text{lin}} t_2 : C_1 \quad \Gamma \mid - \vdash_{\text{SM}} t_1 : C_1 \multimap C_2}{\Gamma \mid \Xi \vdash_{\text{lin}} t_1 t_2 : C_2} \\
\\
\frac{\Gamma \mid - \vdash_{\text{lin}} t_2 : C_1 \quad \Gamma \mid \Xi \vdash_{\text{SM}} t_1 : C_1 \rightarrow C_2}{\Gamma \mid \Xi \vdash_{\text{lin}} t_1 t_2 : C_2}
\end{array}$$

Figure 4.6: Typing rules for SM with linearity condition

Indeed, for an internal monad $X \vdash_{\text{SM}} C$ in SM, the linearity condition on \mathbf{bind}^C requires a derivation of

$$A, B \mid - \vdash_{\text{lin}} \mathbf{bind}^C : C\{A/X\} \multimap (A \rightarrow C\{B/X\}) \rightarrow C\{B/X\}$$

from which we can derive that

$$A, B, f : A \rightarrow B \mid - \vdash_{\text{lin}} \lambda^\diamond x. \mathbf{bind}^C x (\lambda y. \mathbf{ret}^C (f y)) : C\{A/X\} \multimap C\{B/X\}$$

that in turn proves that the right square in the diagram below commutes thanks to [Thm. 4.3.5](#):

$$\begin{array}{ccccc}
\mathbb{M} A & \xrightarrow{\mathbb{M}(\mathbf{ret}_A^C)} & \mathbb{M}[\![C]\!]_{\mathbb{M}} A & \xrightarrow{\alpha_{\mathbb{M}, A}^C} & [\![C]\!]_{\mathbb{M}} A \\
\mathbb{M} f \downarrow & & \downarrow \mathbb{M}[\![C]\!]_{\mathbb{M}} f & & \downarrow [\![C]\!]_{\mathbb{M}} f \\
\mathbb{M} B & \xrightarrow{\mathbb{M}(\mathbf{ret}_B^C)} & \mathbb{M}[\![C]\!]_{\mathbb{M}} B & \xrightarrow{\alpha_{\mathbb{M}, B}^C} & [\![C]\!]_{\mathbb{M}} B
\end{array}$$

Thus, under the assumption that \mathbf{bind}^C has a linear typing derivation (a syntactic object), we prove that its denotation is homomorphic with respect to the relevant M -algebra structure.

In order to prove the [Thm. 4.3.5](#), we need to :

- ▷ provide an interpretation of the linear types;
- ▷ show that linear derivations yield a denotation in this interpretation;
- ▷ prove using a logical relation that the linear interpretation of a term is related to the monotonic interpretation.

The interpretation of linear types is quite straightforward, mainly enforcing that the linear function space to be interpreted by homomorphisms:

$$\begin{aligned} \llbracket C \rrbracket_M &= \llbracket C \rrbracket_M & \llbracket C_1 \multimap C_2 \rrbracket_M &= \{ f : \llbracket C_1 \rrbracket_M \rightarrow \llbracket C_2 \rrbracket_M \mid f \circ \alpha_M^{C_1} = \alpha_M^{C_2} \circ M f \} \\ \llbracket L_1 \times L_2 \rrbracket_M &= \llbracket L_1 \rrbracket_M \times \llbracket L_2 \rrbracket_M & \llbracket (x : A) \rightarrow L \rrbracket_M &= (x : A) \rightarrow \llbracket L \rrbracket_M \\ \llbracket L_1 \rightarrow L_2 \rrbracket_M &= \llbracket L_1 \rrbracket_M \rightarrow \llbracket L_2 \rrbracket_M \end{aligned}$$

Theorem 4.3.6 (Denotation of linear typings). *Let M be a monad, $\Delta; \Gamma \mid \Xi \vdash_{\text{lin}} t : L$ a linear typing derivation of a term in SM , $\vdash_{\mathcal{L}} \delta : \Delta$ a substitution for the \mathcal{L} context Δ , $\vdash_{\mathcal{L}} \gamma : \langle \Gamma \rangle_M$ a substitution for the SM context Γ , and $\vdash_{\mathcal{L}} \xi : \langle \Xi \rangle_M$. Then there is a well defined denotation $\langle t \rangle_M^{\delta; \gamma; \xi} : \llbracket L \rrbracket_M$ and if $\Xi = (x : C_1)$ then $L = C_2$ and $\lambda x. \langle t \rangle_M^{\delta; \gamma; x} : \llbracket C_1 \rrbracket_M \rightarrow \llbracket C_2 \rrbracket_M$ is an M -algebra homomorphism.*

Proof. By induction on the linear typing derivation (each case corresponding to one derivation rule in Figure 4.6):

Case $t = \text{ret}$, $\langle \text{ret} \rangle_M^{\delta; \gamma; -} = \text{ret}^M : A \rightarrow MA = \langle A \rightarrow MA \rangle_M$

Case $t = \text{bind}$, $\langle \text{bind} \rangle_M^{\delta; \gamma; -} = \text{bind}^M : MA \rightarrow (A \rightarrow MB) \rightarrow MB$ with

$$MA \rightarrow (A \rightarrow MB) \rightarrow MB = \langle MA \rangle_M \rightarrow \langle (A \rightarrow MB) \rightarrow MB \rangle_M$$

and bind^M a homomorphism between the respective M -algebra structures

Case $t = x$ is linear, $\langle x \rangle_M^{\delta; \gamma; \xi} = \xi$ and the identity is an M -algebra map

Case $t = x$ is not linear, $\langle x \rangle_M^{\delta; \gamma; -} = \gamma(x)$

Case $t = (t_1, t_2)$, $\langle (t_1, t_2) \rangle_M^{\delta; \gamma; \xi} = \left(\langle t_1 \rangle_M^{\delta; \gamma; \xi}, \langle t_2 \rangle_M^{\delta; \gamma; \xi} \right)$ and $\lambda \xi. \left(\langle t_1 \rangle_M^{\delta; \gamma; \xi}, \langle t_2 \rangle_M^{\delta; \gamma; \xi} \right)$ is an M -algebra map if and only if both $\lambda \xi. \langle t_1 \rangle_M^{\delta; \gamma; \xi}$ and $\lambda \xi. \langle t_2 \rangle_M^{\delta; \gamma; \xi}$ are M -algebra maps

Case $t = \pi_i t'$, $\langle \pi_i t' \rangle_M^{\delta; \gamma; \xi} = \pi_i \langle t' \rangle_M^{\delta; \gamma; \xi}$ and $\lambda \xi. \pi_i \langle t' \rangle_M^{\delta; \gamma; \xi}$ is an M -algebra map whenever $\lambda \xi. \langle t' \rangle_M^{\delta; \gamma; \xi}$ is an M -algebra map

Case $t = \lambda x. t'$, $\langle \lambda x. t' \rangle_M^{\delta; \gamma; \xi} = \lambda x. \langle t' \rangle_M^{\delta[x:=x]; \gamma; \xi}$ and $\lambda \xi. \lambda x. \langle t' \rangle_M^{\delta[x:=x]; \gamma; \xi}$ is an M -algebra map if and only if for any $\vdash_{\mathcal{L}} x : A$, $\lambda \xi. \langle t' \rangle_M^{\delta[x:=x]; \gamma; \xi}$ is an M -algebra map

Case $t = t' v$, $\langle t' v \rangle_M^{\delta; \gamma; \xi} = \langle t' \rangle_M^{\delta; \gamma; \xi} v \{ \delta \}$ and $\lambda \xi. \langle t' \rangle_M^{\delta; \gamma; \xi} v \{ \delta \}$ is an M -algebra map whenever $\lambda \xi. \langle t' \rangle_M^{\delta; \gamma; \xi}$ is an M -algebra map

Case $t = \lambda^\diamond x. t' : C_1 \multimap C_2$, $\langle \lambda^\diamond x. t' \rangle_M^{\delta; \gamma; -} = \lambda x. \langle t' \rangle_M^{\delta; \gamma; x} : \langle C_1 \rangle_M \rightarrow \langle C_2 \rangle_M$ and it is an M -algebra map by induction hypothesis

Case $t = \lambda^\diamond x. t' : L_1 \rightarrow L_2$, $\langle \lambda^\diamond x. t' \rangle_M^{\delta; \gamma; \xi} = \lambda x. \langle t' \rangle_M^{\delta; \gamma[x:=x]; \xi}$ and $\lambda \xi. \lambda x. \langle t' \rangle_M^{\delta; \gamma[x:=x]; \xi}$ is an M -algebra map if and only if for any $\vdash_{SM} x : \langle L_1 \rangle_M$, $\lambda \xi. \langle t' \rangle_M^{\delta; \gamma[x:=x]; \xi}$ is an M -algebra

Case $t : C_1 \rightarrow C_2$ is obtained from $t : C_1 \multimap C_2$, the denotation of the term is the same, we just forget that it is an homomorphism

Case $t = t_1 t_2$, when $t_1 : C_1 \multimap C_2$, $\langle t_1 t_2 \rangle_M^{\delta; \gamma; \xi} = \langle t_1 \rangle_M^{\delta; \gamma; -} \langle t_2 \rangle_M^{\delta; \gamma; \xi}$ and $\lambda \xi. \langle t_1 \rangle_M^{\delta; \gamma; -} \langle t_2 \rangle_M^{\delta; \gamma; \xi}$ is an M -algebra map whenever $\lambda \xi. \langle t_2 \rangle_M^{\delta; \gamma; \xi}$ is an M -algebra map since $\langle t_1 \rangle_M^{\delta; \gamma; -}$ is an M -algebra map

Case $t = t_1 t_2$, **otherwise**, $\llbracket t_1 t_2 \rrbracket_M^{\delta;\gamma;\xi} = \llbracket t_1 \rrbracket_M^{\delta;\gamma;\xi} \llbracket t_2 \rrbracket_M^{\delta;\gamma;-}$ and $\lambda\xi. \llbracket t_1 \rrbracket_M^{\delta;\gamma;\xi} \llbracket t_2 \rrbracket_M^{\delta;\gamma;-}$ is an M -algebra map whenever $\lambda\xi. \llbracket t_1 \rrbracket_M^{\delta;\gamma;\xi}$ is an M -algebra map

□

Given a linear type L , we can forget all the linear annotations, obtaining a type $\vdash_{\text{SM}} |L|$ in SM. In the same fashion, given a derivation $\Delta; \Gamma \mid \Xi \vdash_{\text{lin}} t : L$, we can obtain a derivation $\Delta; |\Gamma, \Xi| \vdash_{\text{SM}} t : |L|$. In order to relate $\llbracket t \rrbracket_M$ and $\llbracket t \rrbracket_M$, we introduce the following relation $\llbracket L \rrbracket_M \subseteq \llbracket L \rrbracket_M \times \llbracket |L| \rrbracket_M$:

$$\begin{aligned} m \llbracket C \rrbracket_M m' &\iff m = m' & f \llbracket C_1 \multimap C_2 \rrbracket_M f' &\iff f = f' \\ (x_1, x_2) \llbracket L_1 \times L_2 \rrbracket_M (x'_1, x'_2) &\iff x_1 \llbracket L_1 \rrbracket_M x'_1 \wedge x_2 \llbracket L_2 \rrbracket_M x'_2 \\ f \llbracket (x : A) \rightarrow L \rrbracket_M f' &\iff \forall (x : A). f x \llbracket L \rrbracket_M f' x \\ f \llbracket L_1 \rightarrow L_2 \rrbracket_M f' &\iff (\forall x x'. x \llbracket L_1 \rrbracket_M x' \rightarrow f x \llbracket L \rrbracket_M f' x') \end{aligned}$$

We extend component-wise this relation to context, and a straightforward but tedious induction shows that for any linear derivation $\Delta; \Gamma \mid \Xi \vdash_{\text{lin}} t : L$ and context $\vdash_{\mathcal{L}} \delta : \Delta, \vdash_{\mathcal{L}} \gamma : (\Gamma \mid \Xi)_M, \gamma' : \llbracket |\Gamma, \Xi| \rrbracket_M$, if $\gamma \llbracket \Gamma \mid \Xi \rrbracket_M \gamma'$ then $\llbracket t \rrbracket_M^{\delta;\gamma} \llbracket L \rrbracket_M \llbracket t \rrbracket_M^{\delta;\gamma'}$ where the right hand side denotation is obtained from the SM derivation $\Delta; |\Gamma, \Xi| \vdash_{\text{SM}} t : |L|$. In the particular case where Ξ is empty and all types in Γ are free from linear annotations, we obtain [Thm. 4.3.5](#).

4.3.5 The Continuation Monad Pseudo-Transformer

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

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

$$\llbracket \text{Cont}_{\mathcal{A}\text{ns}} \rrbracket_{\text{Id}} \longleftarrow \{ \text{Cont}_{\mathcal{A}\text{ns}} \}_{\text{Id}, \text{Cont}_{\mathbb{P}}}^{\text{ret}} \longrightarrow \llbracket \text{Cont}_{\mathcal{A}\text{ns}} \rrbracket_{\text{Cont}_{\mathbb{P}}}$$

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

$$\begin{aligned} &m \{ \text{Cont}_{\mathcal{A}\text{ns}} \}_{\text{Id}, \text{Cont}_{\mathbb{P}}}^{\text{ret}} w \\ \iff &\forall (k : X \rightarrow \mathcal{A}\text{ns}) (w_k : X \rightarrow \text{Cont}_{\mathbb{P}}(\mathcal{A}\text{ns})). (\forall (x : X). \text{ret}(k x) = w_k x) \Rightarrow \text{ret}(m k) = w w_k \\ \iff &\forall (k : X \rightarrow \mathcal{A}\text{ns}). \text{ret}(m k) = w (\lambda x. \text{ret}(k x)) \\ \iff &\forall (k : X \rightarrow \mathcal{A}\text{ns}) (p : \mathcal{A}\text{ns} \rightarrow \mathbb{P}). w (\lambda x q. q(k x)) p = p(m k) \end{aligned}$$

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

4.4 Embedding SM in Coq

We have formalized the SM language presented in the previous section in Coq, taking Gallina as the base language \mathcal{L} and providing an implementation of the denotation of SM terms and logical relation-based elaboration to specification monad transformers. The original goals of this implementation was to provide modular construction of monad transformers helping us in the definition of specification monads, ultimately leading to constructions of Dijkstra monads as explained in [section 5.1](#). For this purpose, the equality proofs witnessing the monadic laws of specification monads have to be as simple as possible, even holding definitionally when achievable. As a consequence, our implementation of SM should:

1. generate monad transformers whose monadic operations are elaborated terms that compute inside Coq; and
2. whenever an elaborated monad transformer is applied to a specification monad whose monadic laws hold definitionally, the resulting specification monad also has definitional monadic laws.

With these goals in mind, we start by explaining our design choices for implementing SM, in particular the representation of binders. An unexpected difficulty arise when trying naively to prove equalities between SM terms, and we explain how we bypass this problem by implementing an abstract machine.

The implementation (<https://gitlab.inria.fr/kmailliar/dijkstra-monads-for-all>) uses the Equations library (Sozeau and Mangin, 2019)⁴ and covers all the previously discussed aspects of SM but the linear type system.

4.4.1 Implementation of the language SM

Two main possibilities arise when implementing a domain specific language (DSL) such as SM: either define a deep embedding, that is an object of the host language describing the syntax, or shallowly embed it in the host language itself, reusing all available features.

On the one hand, a full deep embedding of SM would require to implement a dependently typed language inside Coq and to provide all the features of Gallina that we assume in instances of monad transformers, for instance sum types. This option was dismissed as seemingly too costly.

On the other hand, an embedding using higher-order abstract syntax (Pfenning and Elliott, 1988) would take care of all the conversions in \mathcal{L} , but require some care for the $\lambda^\diamond x. t$ binders since they are not straightforwardly elaborated to Gallina's functions.

We implemented the latter, with intrinsically typed syntax, meaning that the type describing the term syntax of SM is actually a type-family indexed by the type of SM types (named `ctype` in the implementation).

An unconvincing attempt: Parametric Higher-Order Abstract Syntax Our first tentative for the term syntax of SM uses PHOAS (Chlipala, 2008) to encode the $\lambda^\diamond x. t$ binders and is presented in [Figure 4.7](#). The idea is to define the term syntax with respect to an arbitrary type of variables `VarType`, and to quantify universally over this type. A term $t : \text{forall } \{ \text{VarType} \}, \text{ctype } c$ can then be elaborated to a variety of other formats by instantiating with the right type of variables carrying some polymorphic substitution as explained in (Atkey et al., 2009). For instance, taking the type of variables to be the type-family `ctype` itself, we easily recover syntactic substitution of terms. The denotation $\llbracket t \rrbracket_M$ of an SM term with respect to a monad M can also be obtained in this fashion.

⁴An early attempt also used the library of Timany and Jacobs (2016) for categorical definitions, but it turned out to be impractical for our use-case.


```
Class VarType := var : ctype → Type.
```

```
Section CTerm.
```

```
Context {VarType}.
```

```
Inductive cterm : ctype → Type :=
```

```
| MRet : forall A, cterm (A ~> CM A)
```

```
| MBind : forall A B, cterm ((A ~> CM B) → (CM A → CM B))
```

```
| Pair : forall {c1 c2}, cterm (c1 → c2 → c1 × c2)
```

```
| Proj1 : forall {c1 c2}, cterm (c1 × c2 → c1)
```

```
| Proj2 : forall {c1 c2}, cterm (c1 × c2 → c2)
```

```
| Abs : forall A (c:forall x:A, ctype), (forall x:A, cterm (c x)) → cterm (CArr c)
```

```
| App : forall {A} {c:forall x:A, ctype} (v : A), cterm (CArr c) → cterm (c v)
```

```
| CVar : forall {c:ctype}, var c → cterm c
```

```
| CAbs : forall (c1:ctype) (c2: ctype), (var c1 → cterm c2) → cterm (c1 → c2)
```

```
| CApp : forall {c1:ctype} {c2: ctype}, cterm (c1 → c2) → cterm c1 → cterm c2.
```

```
End CTerm.
```

Figure 4.7: PHOAS definition of the term syntax of SM

We could wonder whether the type `forall {VarType}, cterm c` actually captures faithfully the syntax of SM. After all, nothing prevent us from implementing a function in that type that first inspects the given instance of `VarType` before choosing which term to return. The solution to this problem is to restrict our attention to *parametric* terms of that type. Indeed Atkey (2009) shows that the only parametric inhabitant of this type are actual pieces of syntax. This result is however external to the type theory and, in order to actually prove lemmas on the elaboration, we need to accompany every term $t : \text{forall } \{ \text{VarType} \}, \text{cterm } c$ with a proof of (unary) parametricity. Furthermore, to build a relation between two elaborations of a term, for instance between the denotations $\llbracket t \rrbracket_M$ and $\llbracket t \rrbracket_{M'}$, we need a witness of (binary) relational parametricity whose type is shown in Figure 4.8. This starts to be a bit tedious but the problem only gets worse: since we want to prove property on our (proof-relevant) relation, we need a proof that the witness of parametricity itself is parametric. The fact that writing by hand parametricity types and witnesses is hardly achievable was an expected problem and Chlipala (2008) uses an axiom providing uniformly proofs of parametricity. Beside the fact that it would make Coq inconsistent, we did not use this approach because we want to define objects that compute out of the parametricity witness, a property broken by such an axiom. We also tried using the paramcoq plugin⁵, an external tool to derive automatically the types and witnesses of parametricity. However the generated code turned out to be difficult to use in practice: for our application we only need parametricity in the parameter `VarType` whereas paramcoq provides parametricity of all parameters which results in a much more complex object. To carry out practically an approach using PHOAS, it would be convenient to work inside a type theory with internalized parametricity such as those described in Bernardy and Moulin (2013); Bernardy et al. (2015).

A mixed approach: De Bruijn meets HOAS In our second attempt, we implement SM terms using at the same time higher-order abstract syntax (HOAS) for the $\lambda x. t$ binders and De Bruijn indices for the $\lambda^\diamond x. t$ ones. This mixed approach frees us from handling explicitly dependent products $(x : A) \rightsquigarrow C$, relying instead on Gallina, while it provides the flexibility we need on the non-dependent product $C_1 \rightarrow C_2$.

⁵<https://github.com/coq-community/paramcoq>

Section CTermRel.

Context (var1 var2 : VarType) (varR : forall c, var1 c → var2 c → Type).

```

Inductive cterm_rel : forall {c}, @cterm var1 c → @cterm var2 c → Type :=
| MRetRel : forall A, cterm_rel (MRet A) (MRet A)
| MBindRel : forall A B, cterm_rel (MBind A B) (MBind A B)
| AbsRel : forall {A c ct1 ct2} (cr : forall x:A, @cterm_rel (c x) (ct1 x) (ct2 x)),
  cterm_rel (Abs A c ct1) (Abs A c ct2)
| AppRel : forall {A c} {ct1 ct2 : cterm (CArr c)} (v:A),
  cterm_rel ct1 ct2 → cterm_rel (ct1 @o v) (ct2 @o v)
| CVarRel : forall {c v1 v2}, varR c v1 v2 → cterm_rel (CVar v1) (CVar v2)
| CAbsRel : forall {c1 c2} {f1 : var1 c1 → cterm c2} {f2 : var2 c1 → cterm c2},
  (forall x1 x2, varR c1 x1 x2 → cterm_rel (f1 x1) (f2 x2)) →
  cterm_rel (CAbs c1 c2 f1) (CAbs c1 c2 f2)
| CAppRel : forall {c1 c2} {ct11 ct12 : cterm (c1 → c2)} {ct21 ct22},
  cterm_rel ct11 ct12 → cterm_rel ct21 ct22 →
  cterm_rel (ct11 @· ct21) (ct12 @· ct22).

```

End CTermRel.

Figure 4.8: Binary parametricity predicate

```

Inductive icterm : ctx → ctype → Type :=
(* ... Omitted constructors ... *)
| IAbs : forall {Γ A c}, (forall x:A, icterm Γ (c x)) → icterm Γ (CArr c)
| IApp : forall {Γ c} (H:isArr c) (v:arrDom H), icterm Γ c → icterm Γ (arrCod H v)
| IVar : forall {Γ} (n:nat) (H:in_ctx n Γ), icterm Γ (lookup H)
| ICabs : forall {Γ c1 c2}, icterm (c1::Γ) c2 → icterm Γ (c1 → c2)
| ICArr : forall {Γ c} (H:isArrC c), icterm Γ c → icterm Γ (arrCDom H) →
icterm Γ (arrCCod H).

```

Figure 4.9: Term syntax of SM using De Bruijn indices

```

Definition st_car (S A:Type) := S ~> CM (A × S).
Definition st_ret (S A:Type) : icterm nil (A ~> st_car A) :=
  IAbs (fun (a:A) => IAbs (fun (s:S) => IMRet (A × S) · (a, s))).
Definition st_bind (S A B:Type) : icterm nil (st_car A → (A ~> st_car B) → st_car B) :=
  ICabs (ICabs (IAbs (fun s0 => IMBind (IAbs (fun r => (vz · (nfst r)) ·
  (nsnd r))) ((↑vz) · [st_car A] s0)))).

```

Figure 4.10: Implementation of the state monad internally to SM

We build the functional version of the logical relation for a covariant type C , but omit the linear type system⁶. Instead, the Coq version of [Thm. 4.3.4](#) assumes a semantic hypothesis requiring that the denotation of bind is homomorphic (where $\text{ctype_alg } M c : M \llbracket c \rrbracket_M \rightarrow \llbracket c \rrbracket_M$ is the algebra structure on the elaboration):

```

Definition homomorphism c1 c2 (f :  $\llbracket c1 \rrbracket_M \rightarrow \llbracket c2 \rrbracket_M$ ) :=
  let  $\alpha_1 := \text{ctype\_alg } M c1$  in
  let  $\alpha_2 := \text{ctype\_alg } M c2$  in
  forall m, f ( $\alpha_1 m$ ) =  $\alpha_2 (f <\$> m)$ .

```

⁶Since we are working with intrinsically typed term, providing the linear type system would amount to implement yet another time a type of terms corresponding to linear type derivations.

Assuming this condition hold on the (partial) elaboration of an internal monad c , we can then derive the full monad transformer (including all the laws). In practice, this condition hold definitionally on our examples, for instance for the state monad presented in Figure 4.10, so there is no proof overhead.

The implementation of the embedding together with the necessary lemmas about the metatheory of SM (substitution, weakening lemma...), the elaborations and the logical relation, as well as a few examples of monad internal to SM amount to 4kloc, evenly separated between specifications and proofs.

4.4.2 Proving equalities between SM terms

As explained in section 4.3, the construction of monad transformer takes as input monad internal to SM. For most of the examples at the beginning of that section, the definition of the underlying type constructor and monadic operations is tedious but not difficult to encode in the deep embedded syntax. However, it turned out that providing the required equality proof witnessing the monadic laws explicitly as equational derivation is hardly manageable even for a simple example such as state (Figure 4.10). Acknowledging the difficulty, we explain here how we sidestepped this task by defining an abstract machine refining the equational theory of SM. The key idea is first to provide an evaluator for SM putting SM terms into canonical forms; and then proving the correctness of the evaluator with respect to the equational theory, generating a witness that a term is equal to its normal form as a byproduct.

A configuration of the abstract machine is a triple (t, π, σ) consisting of a term t , a stack π and an environment σ (a substitution) for SM terms. We use the notation $\langle t \parallel \pi \rangle_\sigma$ for configurations. The implementation of stacks in Coq as well as the type of the functions reducing configurations of the abstract machine and rebuilding terms out of configurations are presented in Figure 4.12. In order to describe the abstract machine we use the following notations: the empty stack is \star , a reified continuation is noted $\text{Cont}(t)$, projections are noted π_i , consing a Coq value v on top of a stack π is $v \cdot \pi$ and an SM term t is $t \diamond \pi$. The transitions of the abstract machine per-se are noted \triangleright . When the abstract machine reaches a configuration $\langle t \parallel \pi \rangle_\sigma$ where t is a returned value or a variable bound to a neutral term in the environment, it switches to the rebuilding procedure noted \triangleright . The rebuilding phase apply to configurations $t \bullet \pi$, dismantling the stack π – essentially corresponding to an inside-out one-hole term context $C_\pi[-]$ – to reconstruct a term $t' \equiv C_\pi[t]$. Since abstract machine configurations ultimately produce a term, we make a small abuse of notations in the rule reducing $@$, placing directly the result of evaluating the configuration $\langle t_2 \parallel \star \rangle_{\text{id}}$ on the top stack.

As shown in the declared type of `reduce`, the implementation in Coq uses step-indexing (the argument `(fuel:nat)`) to enforce termination. We introduced this abstract machine for a pragmatic purpose, and in that respect this trick achieve the goal successfully at a low implementation cost. Nonetheless, termination of the abstract machine should be provable without, since the part of SM the abstract machine is reducing is essentially simply typed.

4.5 Towards a categorical approach to relative monad transformers

Beside the implemented version of SM, we wish to have a more conceptual understanding of the limitations and potential extensions of the SM language. In particular, the Coq development of SM is specialized to produce specification monad transformers, however most of the proofs seem to be of general nature, enabling an extension to other kind of relative monad transformers. In this section, extend Lem. 4.2.1 in two different directions. First, we extend it to account for a collection of monads, or rather to natural transformations $\theta : F \dashrightarrow G$ between functors $F, G : K \rightarrow \mathcal{Mnd}(\mathcal{C})$ from a category K to monads on a category \mathcal{C} , objects that we call $(K\text{-})$ indexed

$\langle \text{ret } v \parallel \pi \rangle_\sigma$	\triangleright	$\text{ret } v[\sigma] \bullet \pi$	$t \bullet *$	\blacktriangleright	t
$\langle \text{bind } t_1 t_2 \parallel \pi \rangle$	\triangleright	$\langle t_1 \parallel \text{Cont}(\lambda x. \langle t_2 x \parallel \pi \rangle) \rangle$	$t \bullet \text{Cont}(f)$	\blacktriangleright	$f^\dagger t$
$\langle (t_1, t_2) \parallel \pi_i \cdot \pi \rangle$	\triangleright	$\langle t_i \parallel \pi \rangle$	$t \bullet \pi_i \cdot \pi$	\blacktriangleright	$\pi_i t \bullet \pi$
$\langle \pi_i t \parallel \pi \rangle$	\triangleright	$\langle t \parallel \pi_i \cdot \pi \rangle$	$t \bullet v \cdot \pi$	\blacktriangleright	$t @ \circ v \bullet \pi$
$\langle \lambda x. t \parallel v \cdot \pi \rangle$	\triangleright	$\langle t[x/v] \parallel \pi \rangle$	$t \bullet t_1 \diamond \pi$	\blacktriangleright	$t @ \cdot v \bullet \pi$
$\langle t @ \circ v \parallel \pi \rangle$	\triangleright	$\langle t \parallel v \cdot \pi \rangle$			
$\langle x \parallel \pi \rangle_\sigma$	\triangleright	$\sigma(x) \bullet \pi$ when $\sigma(x)$ is neutral			
$\langle x \parallel \pi \rangle_\sigma$	\triangleright	$\langle \sigma(x) \parallel \pi \rangle_{\text{id}}$ otherwise			
$\langle \lambda^\circ x. t \parallel t_1 \diamond \pi \rangle_\sigma$	\triangleright	$\langle t \parallel \pi \rangle_{\sigma[x:=t_1]}$			
$\langle t_1 @ \cdot t_2 \parallel \pi \rangle_\sigma$	\triangleright	$\langle t_1 \parallel \langle t_2 \parallel \star \rangle_{\text{id}} \diamond \pi \rangle_\sigma$			

Figure 4.11: Reduction of abstract machine configurations

```

Inductive stack {Γ c} : ctype → Type :=
| StckNil : stack c
| StckCont : forall A, (A → icterm Γ c) → stack (CM A)
| StckProj1 : forall c' (H : isProd c'), stack (prodProj1 H) → stack c'
| StckProj2 : forall c' (H : isProd c'), stack (prodProj2 H) → stack c'
| StckArg : forall c' (H : isArr c') (v : arrDom H), stack (arrCod H v) → stack c'
| StckCArg : forall c' (H : isArrC c'), icterm Γ (arrCDom H) → stack (arrCCod H) → stack c'.

```

Definition reduce $(\Gamma_0 : \text{ctx}) (\text{fuel} : \text{nat}) :$
 $\text{forall } c0 \Gamma c (t : \text{icterm } \Gamma c) (\pi : @\text{stack } \Gamma_0 c0 c) (\sigma : \text{isubstitution } \Gamma_0 \Gamma), \text{icterm } \Gamma_0 c0.$

Definition rebuild $\Gamma \{c0 c\} (\pi : @\text{stack } \Gamma c0 c) : \text{forall } (t : \text{icterm } \Gamma c), \text{icterm } \Gamma c0.$

Figure 4.12: Stacks and Abstract machine reduction for SM

monads in (Maillard and Melliès, 2015). Second, we explain how to generalize this lemma to the case of relative monads in a framed bicategory, closing the loop with the setting of [chapter 3](#).

As a motivation, observe that a simple special case of natural transformations between indexed monads recover the notion of monad transformers. Given categories K, \mathcal{C} and functors $F, G : K \rightarrow \mathcal{Mnd}(\mathcal{C})$, a natural transformation $\theta : F \rightarrow G$ is a collection of monad morphisms $\theta_k : F k \rightarrow G k$ for $k \in K$. Taking K to be the category $\mathcal{Mnd}(\mathcal{C})$ of monads on \mathcal{C} and F to be the identity functor $\text{Id}_{\mathcal{Mnd}(\mathcal{C})}$, the data of a pair (G, θ) correspond exactly to a monad transformer on \mathcal{C} .

In order to state the generalization of [Lem. 4.2.1](#) for indexed monads, we introduce the following notations. Given categories K, \mathcal{C} , we note $\bar{\mathcal{C}} : K \rightarrow \mathcal{Cat}$ the constant functor with value \mathcal{C} . If $G : K \rightarrow \mathcal{Mnd}(\mathcal{C})$ is a K -indexed monad, we define a transformation $t^G : \bar{\mathcal{C}} \rightarrow \bar{\mathcal{C}}$, with component at $k \in K$ set to $t_k^G = G k : \mathcal{C} \rightarrow \mathcal{C}$. t^G is **not** natural but *lax-natural*, which means that for $f \in K(k, k')$, the following naturality square is filled by a (not necessarily invertible) 2-cell $t_f^G = G f$

$$\begin{array}{ccc}
\mathcal{C} & \xrightarrow{t_k^G = G k} & \mathcal{C} \\
\bar{c}_f \parallel & \Downarrow t_f^G & \parallel \bar{c}_f \\
\mathcal{C} & \xrightarrow{t_{k'}^G = G k'} & \mathcal{C}
\end{array}$$

Moreover, there are modifications $\eta^G : \text{id}_{\bar{\mathcal{C}}} \rightarrow t^G$ and $\mu^G : t^G \circ t^G \rightarrow t^G$ induced by the

pointwise monad structure of t_k^G and the fact that t_f^G are monad morphisms. In details, for each $k \in K$ we have natural transformations $\eta_k^G : \text{Id}_{\mathcal{C}} \rightarrow Gk$, $\mu_k^G : (Gk)^2 \rightarrow Gk$, respectively the unit and multiplication of the monad Gk , satisfying for any $f \in K(k, k')$ the identities

$$\begin{array}{ccc} \begin{array}{c} \mathcal{C} \\ \Downarrow \eta_k^G \\ \mathcal{C} \\ \Downarrow t_f^G \\ \mathcal{C} \\ \Downarrow t_k^G \end{array} & = & \begin{array}{c} \mathcal{C} \\ \Downarrow \eta_k^G \\ \mathcal{C} \\ \Downarrow t_k^G \end{array} \end{array} \quad \begin{array}{ccc} \begin{array}{c} \mathcal{C} \\ \Downarrow \mu_k^G \\ \mathcal{C} \\ \Downarrow t_f^G \\ \mathcal{C} \\ \Downarrow t_k^G \end{array} & = & \begin{array}{c} \mathcal{C} \\ \Downarrow t_f^G \circ t_f^G \\ \mathcal{C} \\ \Downarrow \mu_k^G \\ \mathcal{C} \\ \Downarrow t_k^G \end{array} \end{array}$$

For $F : K \rightarrow \mathcal{Mnd}(\mathcal{C})$, we note $\mathcal{C}^F = \mathcal{Alg} \circ F^{\text{op}} : K^{\text{op}} \rightarrow \mathcal{Cat}$ the functor assigning to $k \in K$ the category of Eilenberg-Moore algebras \mathcal{C}^{Fk} . There is a *natural* transformation $u : \mathcal{C}^F \rightarrow \bar{\mathcal{C}}$ given at each component k by the relevant forgetful functor sending an algebra in \mathcal{C}^{Fk} to its carrier in $\bar{\mathcal{C}}k = \mathcal{C}$. Since u is natural, it acts by post-composition on lax-natural transformations and modifications, inducing in particular a functor (of 1-categories)

$$u_* : [K^{\text{op}}, \mathcal{Cat}]_{\text{lax}}(\bar{\mathcal{C}}, \mathcal{C}^F) \longrightarrow [K^{\text{op}}, \mathcal{Cat}]_{\text{lax}}(\bar{\mathcal{C}}, \bar{\mathcal{C}}).$$

Lemma 4.5.1. *Let \mathcal{C}, K be categories, and $F, G : K \rightarrow \mathcal{Mnd}(\mathcal{C})$ be functors to monads on \mathcal{C} . There is a bijective correspondence between*

1. *natural transformations $\theta : F \rightarrow G$,*
2. *lax-natural liftings $\tilde{t}^G : \bar{\mathcal{C}} \rightarrow \mathcal{C}^F$ of t^G through u such that the modification μ^G also lifts to $\tilde{\mu}^G : \tilde{t}^G \circ t^G \rightarrow \tilde{t}^G$.*

Proof. (1 \Rightarrow 2) The component at $k \in K$ of the natural transformation $\theta : F \rightarrow G$ provides a monad morphism $\theta_k : Fk \rightarrow Gk$. By **Lem. 4.2.1** point 2, we obtain for each $k \in K$ a lifting $\tilde{t}_k^G : \bar{\mathcal{C}}k \rightarrow \mathcal{C}^{Fk}$ of $t_k^G = Gk : \bar{\mathcal{C}}k \rightarrow \bar{\mathcal{C}}k$ through $u_k : \mathcal{C}^{Fk} \rightarrow \mathcal{C}$ such that μ_k^G lifts as $\tilde{\mu}_k^G$. Now, given $f \in K(k, k')$, we need to show that $t_f^G = Gf$ lifts as a natural transformation \tilde{t}_f^G filling the naturality diagram below on the left, that is Gf should be an Fk -algebra morphism. This is indeed the case since the diagram on the right commutes by naturality of θ_k and μ^G . Moreover, $\tilde{\mu}^G$ is a modification since Gf is a monad morphism.

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{\tilde{t}_k^G = Gk} & \mathcal{C}^{Fk} \\ \parallel & \Downarrow \tilde{t}_f^G & \uparrow \mathcal{C}^{Ff} \\ \mathcal{C} & \xrightarrow{\tilde{t}_{k'}^G = Gk'} & \mathcal{C}^{Fk'} \end{array} \quad \begin{array}{ccc} Fk Gk(c) & \xrightarrow{Fk Gf(c)} & Fk Gk'(c) \\ \downarrow \theta_{k, Gk(c)} & & \downarrow Ff_{Gk'(c)} \\ & & Fk' Gk'(c) \\ & & \downarrow \theta_{k', Gk'(c)} \\ Gk Gk(c) & \dashrightarrow & Gk' Gk'(c) \\ \downarrow \mu_k^G & & \downarrow \mu_{k'}^G \\ Gk(c) & \xrightarrow{Gf(c)} & Gk'(c) \end{array}$$

(2 \Rightarrow 1) Conversely, another application of **Lem. 4.2.1** provides the definition of θ_k at each $k \in K$. Explicitly, θ_k is obtained by extending η_k^G to an Fk homomorphism thanks to the Fk -algebras structure on Gk (see the proof of **Lem. 4.5.2** below). We still need to show that θ is natural in k . Since \tilde{t}^G is a lifting of \tilde{t} , for any $f \in K(k, k')$, the underlying natural transformation of \tilde{t}_f^G is necessarily Gf , and unfolding the lax-naturality condition, Gf is an Fk -algebra homomorphism, justifying the following computation (where we write α_k for the Fk -algebra structure on Gk):

$$\begin{aligned}
Gk - \textcircled{Gf} - \textcircled{\theta_k} - Fk &= Gk - \textcircled{Gf} - \textcircled{\alpha_k} \begin{array}{l} \circ \eta_k^G \\ \text{---} Fk \end{array} \\
&= Gk - \textcircled{\alpha_{k'}} \begin{array}{l} \textcircled{Gf} \circ \eta_k^G \\ \textcircled{Ff} \text{---} Fk \end{array} \\
&= Gk - \textcircled{\alpha_{k'}} \begin{array}{l} \circ \eta_{k'}^G \\ \text{---} Ff \text{---} Fk \end{array} = Gk - \textcircled{\theta_{k'}} - \textcircled{Ff} - Fk
\end{aligned}$$

□

How does [Lem. 4.5.1](#) help us understanding SM ? We can interpret covariant type constructors in SM as lax natural transformations from \bar{C}^n to \bar{C} where n is the number of type arguments. For instance, the unary type constructor \mathbb{M} should be interpreted by $t^{\text{Id}, \mathcal{M}\text{nd}(\mathcal{C})}$. The choice of type formers accepted in SM should then be induced by the lifting condition along u . Finally, the condition that the multiplication should lift as well can be seen as a counterpart of the linearity condition.

We now explain how to extend [Lem. 4.2.1](#) to the relative monad setting. We will not attempt to generalize [Lem. 4.5.1](#) to the relative setting. Even though we expect some generalization to the relative setting to hold, introducing the structures corresponding to lax-natural transformations and modifications in the relative monad setting would bring us too far.

Lemma 4.5.2. *Let \mathcal{F} be a framed bicategory, $j : I \rightarrow C$ a vertical arrow in \mathcal{F} and m_1 a j -relative monad. For any j -relative monad m_2 , there is a bijective correspondence between:*

- ▷ *relative monad morphisms $m_1 \rightarrow m_2$ over the identity of j , and*
- ▷ *m_1 -algebra structures on m_2 such that bind^{m_2} lifts to an m_1 -algebra morphism, that is $\text{bind}^{m_2} \in \text{Alg}_{m_1}(m_2^* C j^*)$.*

Proof. Given a relative monad morphism $\theta : m_1 \rightarrow m_2$ over the identity of j , we equip m_2 with an m_1 -algebra structure α_θ defined as

$$\begin{array}{c} m_2^* \quad j^* \\ \downarrow \quad \uparrow \\ \textcircled{\alpha_\theta} \\ \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} = \begin{array}{c} m_2^* \quad j^* \\ \downarrow \quad \uparrow \\ \textcircled{\text{bind}^{m_2}} \\ \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} \begin{array}{c} \text{---} \textcircled{\theta} \text{---} \\ \text{---} m_1^* \end{array}$$

The 2-cell α_θ is indeed an m_1 -algebra structure as shown by the two following computations using the fact that θ preserves ret and bind .

$$\begin{array}{c} m_2^* \quad j^* \\ \downarrow \quad \uparrow \\ \textcircled{\alpha_\theta} \\ \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} \begin{array}{c} \text{---} \textcircled{\text{ret}^{m_1}} \text{---} \\ \text{---} j \end{array} = \begin{array}{c} m_2^* \quad j^* \\ \downarrow \quad \uparrow \\ \textcircled{\text{bind}^{m_2}} \\ \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} \begin{array}{c} \text{---} \textcircled{\theta} \text{---} \\ \text{---} \textcircled{\text{ret}^{m_1}} \text{---} \\ \text{---} j \end{array} = \begin{array}{c} m_2^* \quad j^* \\ \downarrow \quad \uparrow \\ \textcircled{\text{bind}^{m_2}} \\ \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} \begin{array}{c} \text{---} \textcircled{\text{ret}^{m_2}} \text{---} \\ \text{---} j \end{array} = \begin{array}{c} m_2^* \quad j^* \\ \downarrow \quad \uparrow \\ \text{---} \\ \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} \begin{array}{c} \text{---} \\ \text{---} j \end{array}$$

$$\begin{aligned}
& \begin{array}{c} m_2^* \quad j^* \quad m_2^* \quad j^* \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ \alpha_\theta \quad \text{bind}^{m_1} \\ \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} = \begin{array}{c} m_2^* \quad j^* \quad m_2^* \quad j^* \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ \text{bind}^{m_2} \quad \text{bind}^{m_1} \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} = \begin{array}{c} m_2^* \quad j^* \quad m_2^* \quad j^* \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ \text{bind}^{m_2} \quad \text{bind}^{m_2} \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} \\
& \begin{array}{c} m_2^* \quad j^* \quad m_2^* \quad j^* \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ \text{bind}^{m_2} \quad \text{bind}^{m_2} \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} = \begin{array}{c} m_2^* \quad j^* \quad m_2^* \quad j^* \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ \alpha_\theta \quad \text{bind}^{m_2} \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} = \begin{array}{c} m_2^* \quad j^* \quad m_2^* \quad j^* \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ \alpha_\theta \\ \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array}
\end{aligned}$$

Moreover bind^{m_2} is a monad morphism in the following sense

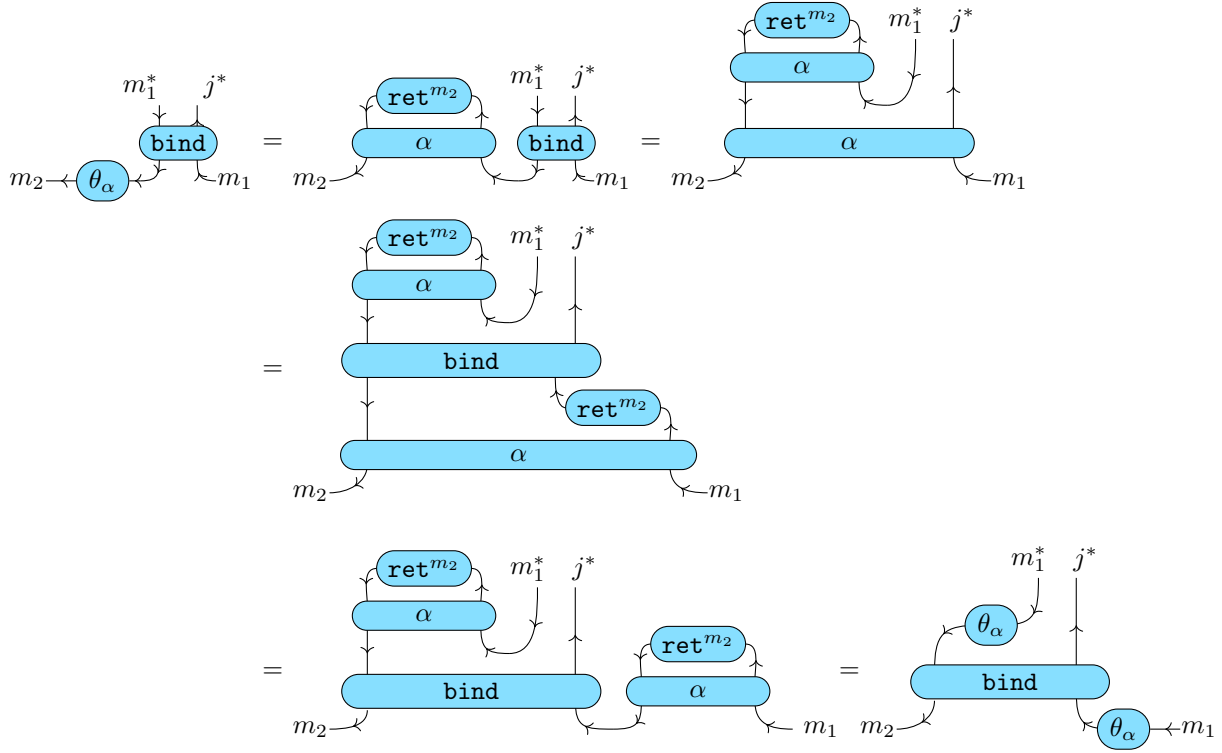
$$\begin{aligned}
& \begin{array}{c} m_2^* \quad j^* \quad m_2^* \quad j^* \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ \text{bind}^{m_2} \quad \alpha_\theta \\ \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} = \begin{array}{c} m_2^* \quad j^* \quad m_2^* \quad j^* \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ \text{bind}^{m_2} \quad \text{bind}^{m_2} \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} \\
& \begin{array}{c} m_2^* \quad j^* \quad m_2^* \quad j^* \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ \text{bind}^{m_2} \quad \text{bind}^{m_2} \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array} = \begin{array}{c} m_2^* \quad j^* \quad m_2^* \quad j^* \\ \downarrow \quad \uparrow \quad \downarrow \quad \uparrow \\ \text{bind}^{m_2} \quad \alpha_\theta \\ \downarrow \quad \uparrow \\ m_2^* \quad m_1^* \end{array}
\end{aligned}$$

In the other direction, if α is an m_1 -algebra structure on m_2 , we define the natural transformation θ_α as

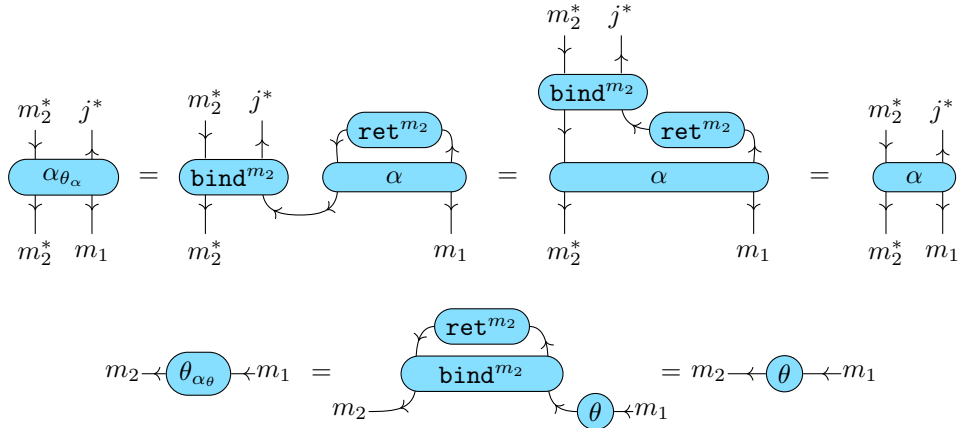
$$m_2 \leftarrow \theta_\alpha \leftarrow m_1 = \begin{array}{c} \text{ret}^{m_2} \quad j \\ \downarrow \quad \uparrow \\ \alpha \\ \downarrow \quad \uparrow \\ m_2 \quad m_1 \end{array}$$

θ_α is actually a monad morphism using the algebra laws and, for the last step of the second computation, the fact that bind^{m_2} is an algebra morphism

$$\begin{aligned}
& m_2 \leftarrow \theta_\alpha \leftarrow \text{ret}^{m_1} \leftarrow j = \begin{array}{c} \text{ret}^{m_2} \quad j \\ \downarrow \quad \uparrow \\ \alpha \\ \downarrow \quad \uparrow \\ m_2 \quad \text{ret}^{m_1} \quad j \end{array} \\
& \begin{array}{c} \text{ret}^{m_2} \quad j \\ \downarrow \quad \uparrow \\ \alpha \\ \downarrow \quad \uparrow \\ m_2 \quad \text{ret}^{m_1} \quad j \end{array} = \begin{array}{c} \text{ret}^{m_2} \quad j \\ \downarrow \quad \uparrow \\ m_2 \quad j \end{array} = m_2 \leftarrow \text{ret}^{m_1} \leftarrow j
\end{aligned}$$



Finally, the two constructions are inverse of each other:



□

4.6 Conclusion & Related work

In this chapter we presented the notion of monad transformers, introduced for by (Liang et al., 1995), and extended it to relative monads. In order to derive correct monad transformers and specification monad transformers, we introduced a metalanguage SM. The types of this metalanguage capture algebras with respect to an abstract monad \mathbb{M} , a notion reminiscent of *modular handlers* introduced in (Schrijvers et al., 2019) to compare monad transformers with the extensible algebraic effects with handlers of (Kiselyov et al., 2013). The elaboration of monads internal to SM to monad transformers employ proof-relevant logical relations to provide the action on monad morphisms, a technique inspired by the work of Kaposi et al. (2019) on models of dependent signatures. An implementation of this metalanguage in Coq brings the convenience of defining monad transformers from a small standard monad specification to a practical level. We

sketched some ideas on how the theory behind SM could be extended to relative monad transformers beyond the case of specification monad transformers. Designing and implementing such an extension is left as an interesting but challenging future work.

At a theoretical level, the work of [Jaskelioff and Moggi \(2010\)](#) where a thorough study on monoid transformers is carried on is closely related to ours. Monoid transformers on an arbitrary monoidal category are by essence more general than monad transformers, and by consequence harder to describe syntactically in general. They tackle this problem by classifying monadic operations in various classes of expressiveness and provide more or less structure on the monoid transformers depending on the well-behavedness of these operations. For instance, they derive a monad transformer for continuations without a functorial action on monad morphism, whereas we extend it to monadic relations. A tempting future work would consist in extending their work on monoid transformers to consider relations between monoids preserving the monoid structure.

Chapter 5

Dijkstra monads

“One Ring to rule them all, One Ring to find them,
One Ring to bring them all and in the darkness bind them[...]

J.R.R. Tolkien, *The Lord of the Rings*, The Fellowship of the Ring

Having examples, theoretical concepts and practical ways to build computational monads, specification monads and effect observations, we now turn to the question of verifying code in practice. Effect observations by themselves provide a rudimentary way to prove properties of programs: given a program $c : M A$ and an effect observation $\theta : M \rightarrow W$, we can prove properties on c by exploiting its assigned specification $\theta(c) : W A$. However, directly applying θ amounts to run the program with respect to the semantics given by θ . This may lead to a complex, hardly modular specification.

This chapter is dedicated to the study of a methodology for verifying unary programs called Dijkstra monads. Dijkstra monads provide a practical and automatable verification technique in dependent type theories like F^* (Swamy et al., 2016), where they are a primitive notion, and Coq, where they can be embedded via dependent types. A Dijkstra monad $\mathcal{D} A w$ refine a computational monad with a specification index picked out of a specification monad. We open this chapter by defining more formally these objects and then show with examples how a Dijkstra monad can be obtained from a computational monad, a specification monad, and an effect observation relating them, providing a methodology for actual verification. The second section then proves that effect observations and Dijkstra monad are categorically equivalent, providing a principled approach to the construction of Dijkstra monads in the examples. We close the chapter on a brief comparison with *graded monads*, another indexed monad structure used to capture for instance resource bounds on computations (Katsumata, 2014).

5.1 Definition & examples

Definition 5.1.1 (🐦). A Dijkstra monad over a specification monad W is given by

- ▷ a type $\mathcal{D} A w$ for each type A and specification $w : W A$,
- ▷ return and bind functions where the index is provided respectively by the return and bind of W

$$\begin{aligned} \text{ret}^{\mathcal{D}} & : (x : A) \rightarrow \mathcal{D} A (\text{ret}^W x) \\ \text{bind}^{\mathcal{D}} & : \mathcal{D} A w_1 \rightarrow ((x : A) \rightarrow \mathcal{D} B w_2(x)) \rightarrow \mathcal{D} B (\text{bind}^W w_1 w_2) \end{aligned}$$

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

$$\begin{aligned}\text{bind}^{\mathcal{D}} m \text{ret}^{\mathcal{D}} &= m \\ \text{bind}^{\mathcal{D}} (\text{ret}^{\mathcal{D}} x) f &= f x \\ \text{bind}^{\mathcal{D}} (\text{bind}^{\mathcal{D}} m f) g &= \text{bind}^{\mathcal{D}} m (\lambda x. \text{bind}^{\mathcal{D}} (f x) g)\end{aligned}$$

where $m : \mathcal{D} A w^m, x : A, f : (x : A) \rightarrow \mathcal{D} B (w^f x), g : (y : B) \rightarrow \mathcal{D} C (w^g y)$ for A, B, C any types and $w^m : W A, w^f : (x : A) \rightarrow W B, w^g : (y : B) \rightarrow W C$.

- ▷ Together with a weakening structure reflecting the order on the specification monad W

$$\text{weaken} : w_1 \leq_A w_2 \times \mathcal{D} A w_1 \longrightarrow \mathcal{D} A w_2$$

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

$$\begin{aligned}\text{weaken} \langle w \leq w, m \rangle &= m, \\ \text{weaken} \langle w_1 \leq w_2 \leq w_3, m \rangle &= \text{weaken} \langle w_2 \leq w_3, \text{weaken} \langle w_1 \leq w_2, m \rangle \rangle, \\ \text{bind}^{\mathcal{D}} (\text{weaken} \langle w_m \leq w'_m, m \rangle) (\lambda a. \text{weaken} \langle w_f a \leq w'_f a, f a \rangle) &= \\ &\text{weaken} \langle \text{bind}^W w_m w_f \leq \text{bind}^W w'_m w'_f, \text{bind}^{\mathcal{D}} m f \rangle.\end{aligned}$$

Intuitively, the type $\mathcal{D} A w$ correspond to “computations specified by w ” and the weakening structure allow to coerce a computation from a stronger to a weaker specification as needed.

Note that the Dijkstra monad equations are well-typed only if W satisfy the monadic laws. In HoTT terminology (Univalent Foundations Program, 2013), these equations are actually paths over the corresponding equations for W . This has no incidence in an extensional type theory such as F^* , but it means for our Coq development that we need to pay attention to the equality proofs for our specification monads. It explains why we are so often relying on the backward predicate transformer specification monad (see subsection 2.3.4) since it has the good taste to satisfy its monad laws definitionally.

In order to use seamlessly multiple Dijkstra monads, that is multiple effects, in a single program, we need a way to coerce computations – and specifications – from one effect to another. F^* uses the concept of *subeffecting* to achieve this. In the implementation, the subeffecting relation is an order on Dijkstra monads generated by a choice of at most one Dijkstra monad morphism between two different Dijkstra monads. Such a Dijkstra monad morphism must hence coerce both computations and specifications.

Definition 5.1.2. A morphism of Dijkstra monads from $\mathcal{D}_1 A (w_1 : W_1 A)$ to $\mathcal{D}_2 A (w_2 : W_2 A)$ consists of:

- ▷ a specification monad morphism $\Theta^W : W_1 \rightarrow W_2$ and
▷ a family of maps

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

indexed by types A and specifications $w_1 : W_1 A$,

- ▷ satisfying the following identities

$$\Theta^{\mathcal{D}} (\text{ret}^{\mathcal{D}_1} a) = \text{ret}^{\mathcal{D}_2} a, \quad \Theta^{\mathcal{D}} (\text{bind}^{\mathcal{D}_1} m f) = \text{bind}^{\mathcal{D}_2} (\Theta^{\mathcal{D}} m) (\Theta^{\mathcal{D}} \circ f),$$

$$\Theta^{\mathcal{D}} (\text{weaken} (w \leq w', m)) = \text{weaken} (\Theta^W w \leq \Theta^W w', \Theta^{\mathcal{D}} m).$$

for any types A, B , and terms $a : A, w, w' : W A, m : \mathcal{D} A w, w^f : A \rightarrow W B, f : (a : A) \rightarrow \mathcal{D} B (w^f a)$.

Considering Dijkstra monads and Dijkstra monad morphisms together, we obtain a category that we will note \mathcal{DMon} .

5.1.1 Using Dijkstra monads for verifying programs

We explain the general methodology for proving code using Dijkstra monads. Consider the following piece of F^* code defining a function mapping a natural number $k : \mathbb{N}$ to the k -th element of Fibonacci sequence.

```
let rec fib (n:ℕ) : Pure ℕ(requires  $\top$ ) (ensures  $(\lambda r. r \geq n \wedge r \geq 1)$ ) =
  if  $n \leq 1$  then 1 else fib (n-1) + fib (n-2)
```

This code does not need any effect¹ and uses implicitly the Dijkstra monad Pure of pure functions provably terminating on the domain given by specifications drawn from W^{Pure} . Translating the `let ... in` constructs to their explicit monadic variant and inserting return operation where needed, the definition of `fib` becomes:

```
let rec fib (n:ℕ) : Pure ℕ(requires  $\top$ ) (ensures  $(\lambda r. r \geq n \wedge r \geq 1)$ ) =
  if  $n \leq 1$  then
    retPure 1
  else
    bindPure (fib (n-1)) ( $\lambda r1.$ 
      bindPure (fib (n-2)) ( $\lambda r2.$ 
        retPure (r1 + r2)))
```

By type inference, the type of the body of `fib` is $\text{Pure } \mathbb{N} \ w_{\text{body}}$ where

```
wbody n = if  $n \leq 1$  then retPure 1
           else bindPure (wfib (n-1)) ( $\lambda r1.$  bindPure (wfib (n-2)) ( $\lambda r2.$  retPure (r1 + r2)))
wfib n =  $\lambda post. \top \wedge \forall r. r \geq n \wedge r \geq 1 \implies post \ r$ 
```

the second specification being derived² from the declared require and ensure clause of the function `fib` above. For the function `fib` to be well-typed, the following *verification condition* (VC) must hold:

$$\forall n, w_{\text{body}} \ n \leq^{W^{\text{Pure}}} w_{\text{fib}} \ n.$$

Formally, it corresponds to wrapping the body of `fib` with a *weaken* operation and providing the proof of the VC as argument. This last step is performed as part of subeffecting in F^* 's type inference/type checking mechanism. When using Dijkstra monads in Coq – or more generally in any dependent type theory where Dijkstra monads are not a primitive notion –, these weakening must be written explicitly.

How is this methodology any better than just applying an effect observation to the code? Observe that the specification w_{body} obtained by type inference is close to what we could obtain when applying an effect observation θ to the body, the difference being that at the leaves of the specification, we have occurrences of w_{fib} instead of θ applied to some recursive occurrences of `fib`. In this small example it might seem to be a benign difference, but it means that we have some control over the specifications that are used and can abstract away irrelevant implementation details. This is an important, albeit rather simple, form of modularity.

Of course, this methodology comes with an important drawback: when defining a function, we need to come up with *the* right specification that strikes a good balance between being simple and complete enough.

5.1.2 Implementing Dijkstra monads in type theory

The concrete definition for the type of a Dijkstra monad can vary according to the underlying type theory. For instance, in our Coq development, we define it (roughly) as a dependent pair of a computation $c : \text{St } A$ and a proof that c is correctly specified by w . In F^* , it is instead a primitive notion.

¹Beside the fixpoint that can be shown to be total; we return to this point when reconstructing Pure .

²The transformation from pairs of pre/postconditions to backward predicate transformer is actually part of the adjunction described in [section 2.3](#)

5.1.3 The Dijkstra monad St of stateful computations

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

$$\begin{aligned}\theta^{\text{St}} &: \text{St} \longrightarrow \text{W}^{\text{St}} \\ \theta^{\text{St}}(m) &= \lambda \text{post } s_0. \text{post } (m \ s_0)\end{aligned}$$

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

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

$$\begin{aligned}\text{ret}^{\text{ST}} &= \text{ret}^{\text{St}} : (v : A) \rightarrow \text{ST } A \ (\text{ret}^{\text{W}^{\text{St}}} v) \\ \text{bind}^{\text{ST}} &= \text{bind}^{\text{St}} : (c : \text{ST } A \ w_c) \rightarrow (f : (x : A) \rightarrow \text{ST } B \ (w_f \ x)) \rightarrow \text{ST } B \ (\text{bind}^{\text{W}^{\text{St}}} w_c \ w_f)\end{aligned}$$

which, after unfolding the state-specific definitions becomes:

$$\begin{aligned}\text{ret}^{\text{ST}} &= \text{ret}^{\text{St}} : (v : A) \rightarrow \text{ST } A \ (\lambda \text{post } s_0. \text{post } (v, \ s_0)) \\ \text{bind}^{\text{ST}} &= \text{bind}^{\text{St}} : (c : \text{ST } A \ w_c) \rightarrow (f : (x : A) \rightarrow \text{ST } B \ (w_f \ x)) \\ &\quad \rightarrow \text{ST } B \ (\lambda p \ s_0. w_c \ (\lambda (x, \ s_1). w_f \ x \ p \ s_1) \ s_0)\end{aligned}$$

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

$$\text{op}^{\text{ST}} = \text{op}^{\text{St}} : (x_1 : A_1) \rightarrow \dots \rightarrow (x_n : A_n) \rightarrow \text{ST } B \ (\theta^{\text{St}}(\text{op}^{\text{St}} \ x_1 \ \dots \ x_n))$$

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

$$\text{get} : \text{ST } S \ (\lambda p \ s_0. p \ (s_0, \ s_0)), \quad \text{put} : (s : S) \rightarrow \text{ST } \mathbb{1} \ (\lambda p \ s_0. p \ (*, \ s)).$$

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

$$\text{ST } \mathbb{1} \ (\text{bind}^{\text{W}^{\text{St}}} (\lambda p \ s_0. p \ (s_0, \ s_0)) (\lambda s \ p \ s_0. p \ (*, \ f s))) = \text{ST } \mathbb{1} \ (\lambda p \ s_0. p \ (*, \ f s_0))$$

which precisely describes the behavior of modify , both in terms of the returned value and of its effect on the state. Program verification then amounts to proving that, given a programmer-provided type-annotation $\text{ST } \mathbb{1} \ w$ for $\text{modify } f$, the specification w is weaker than the inferred specification.

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

Aparté: State in real life The basic idea of a Dijkstra monad for state can be extended to apply to more realistic situations using a curated memory model closer to an actual implementation. We briefly explain the model used in F^{*} targetting generation of efficient low-level C code to give an idea of how to step up from a toy model to an actual tool for program verification. A more complete account of this exposition can be found in (Protzenko et al., 2017).

We already explain in chapter 2 how the state S can be instantiated by a store $S = \text{Loc} \rightarrow \text{Val}$ to accommodate for multiple memory cells. Pushing this idea further, we can structure the state as a tree of regions, each region holding its own set of memory cells. This model called *hyper-heap* in F^{*} provides a primitive variant of separation and framing, an important tool to prove that a program does not interfere with regions it does not touch. Specializing some of the regions to reflect the differences between the heap and the stack in the C memory model, we obtain the *hyper-stack* model. In more details, the stack is a list-shaped sub-tree of regions, each of these region corresponding to a stack frame, maintaining liveness condition satisfying the stack discipline, e.g., memory cells on the stack cannot outlive their stack frame. The heap on the other hand has a much more liberal discipline.

The hyper-stack model per-se does not use anything F^{*}-specific. However to reason efficiently about stateful arguments, monotonicity arguments are heavily used in F^{*} (Ahman et al., 2018), for instance to discharge the hypothesis that a garbage collected reference that is reachable in a program is always alive, that is both allocated and not freed. If it seems achievable to provide a Dijkstra monad for monotonic state⁴, it does not seem possible to obtain the crucial operations witness and recall. In the setting of Coq, the Iris framework (Jung et al., 2018) may be an interesting way to obtain similar reasoning methods.

5.1.4 The demonic Dijkstra monad ND_{X}

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

$$\text{pick}^{\text{ND}_{\text{X}}} : \text{ND}_{\text{X}} \mathbb{B} (\lambda p. p \text{ true} \wedge p \text{ false}) \quad \text{fail}^{\text{ND}_{\text{X}}} : \text{ND}_{\text{X}} () (\lambda p. \top)$$

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

```

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

let guard (b: ℤ) : NDD unit (λ p → b ⇒ p ()) = if b then () else fail ()

```

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

```

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

```

⁴Using the monotonic state transformer from section 4.3.

5.1.5 Interacting with the outer world: the IO Dijkstra monad family

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

$$\begin{aligned} \text{read}^{\text{IOFr}} &: \text{IOFree } I (\lambda p. \forall (i : I). p \langle i, [\text{In } i] \rangle) \\ \text{write}^{\text{IOFr}} &: (o : O) \rightarrow \text{IOFree } \mathbb{1} (\lambda p. p \langle *, [\text{Out } o] \rangle) \end{aligned}$$

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

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

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

$$\begin{aligned} \text{read}^{\text{IOHist}} &: \text{IOHist } I (\lambda p h. \forall i. p \langle i, [\text{In } i] \rangle) \\ \text{write}^{\text{IOHist}} &: (o : O) \rightarrow \text{IOHist } \mathbb{1} (\lambda p h. p \langle *, [\text{Out } o] \rangle) \end{aligned}$$

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

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

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

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

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

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

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

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

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

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

5.1.6 Provably terminating recursion in Pure

The Pure effect is the most basic effect, containing only pure, provably total computations. We implement a model of Pure in Coq over the specification monad $W^{\text{Pure}} A = \text{Cont}_{\mathbb{P}} A$ (or rather its monotonic refinement). The underlying representation of a pure computation is given by

$$\text{Pure } A \, w = (p : A \rightarrow \mathbb{P}) \rightarrow w \, p \rightarrow (a : A) \times p \, a$$

That is, given a postcondition $p : A \rightarrow \mathbb{P}$, a pure computation $c : \text{Pure } A \, w$ is a total function taking as input a proof that the precondition $w \, p$ holds and returning values $a : A$ such that $p \, a$ holds.

We use this simple Dijkstra monad to explain how to combine the ideas from (McBride, 2015) with effect observations to yield a presentation of provably terminating recursive functions close to what can be found in F^* . We start by briefly explaining how termination proof work in F^* . In order to define a Pure recursive function f taking arguments of type Arg and returning a result of type R according to the specification $w : W^{\text{Pure}} R$, an F^* user have to specify a *measure* l which is an arbitrary total expression depending on the arguments of the function, here $args$.

```
let rec f(args:Arg) : Pure R w (decreases l) =
  (* body of f containing recursive calls *)
  ... f args' ...
```

This measure acts on the specification of f inside its body, inducing the following signature (note that $args$ correspond to the top-level argument and is bound inside the body):

```
val f : (args' : Arg) → Pure R (λ p. l[args'/args] < l ∧ w[args'/args] p)
```

Here $<$ is a well-founded relation between arbitrary F^* terms generated by the subterm order on inductive types⁵. The overall result is that every recursive are guarded in the sense that the function f can only be called on arguments for which the measure provably decreases.

In order to emulate this mechanism in Coq for a recursive function with fixed domain Arg , codomain R and specification $w : W^{\text{Pure}} R$, we construct a Dijkstra monad from effect observations on top of the monad GenRec for general recursion presented in subsection 2.1.2. We assume that Arg comes equipped with a well-founded relation $<$. The specification monad we use is simply W^{Pure} , however instead of fixing a single effect observation, we define a family of effect observations θ_{args} parametrized by the top-level parameter $(args:Arg)$. Recall that the monad $\text{GenRec } Arg \, R$ is free on a single operation $\text{call} : (args' : Arg) \rightsquigarrow R$ and that an effect observation from a free monad is fully defined by specifications for each operations subsection 2.4.5, in the present case a single function $w_{\text{call}} : (args' : Arg) \rightarrow W^{\text{Pure}} R$. Putting these observation together we define the family of effect observations $\theta_{args} : \text{GenRec } Arg \, R \, X \rightarrow W^{\text{Pure}} X$ as induced by $w_{\text{call}} \, args$

```
let w_call (args: Arg) : (args' : Arg) → W^Pure R = λ(p : R → ℙ). args' < args ∧ w[args'/args] p
```

This induces a family of Dijkstra monads $\text{GenREC}_{args} X (w : W^{\text{Pure}})$ parametrized by $args : Arg$. Now, to close the loop, we can define a handling construct fix from this newly defined Dijkstra monad to Pure:

```
val fix : ((args : Arg) → GenREC_args R w) → (args:Arg) → Pure R w
```

The role of fix is to gather the proofs of well-foundedness carried by each recursive call inside its first argument thanks to the specification w_{call} and to transform it into a witness that the putative fixpoint is globally well-defined.

We presented a program transformation translating F^* source syntax to describe provably terminating recursive functions into a form admissible in Coq.

⁵The actual relation in F^* also take in the decreasing order on natural numbers and a construction for lexicographic order.

5.1.7 Effect polymorphic functions

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

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

When working with Dijkstra monads, we can use the `mapW` function as a generic specification for the same computation when expressed using an arbitrary Dijkstra monad D indexed by W .⁶

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

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

Analogously, we can implement a generic iterator combinator provided we have an invariant $w : W \text{ unit}$ for the loop `body` : $\mathbb{N} \rightarrow D \text{ unit } w$ such that the invariant satisfies $\text{bind } w (\lambda () \rightarrow w) \leq w$:

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

Here we use not only the monadic operations but also the possibility to weaken the specification $\text{bind } w (\lambda () \rightarrow w)$ computed from the second branch of the `match` to the specification w by assumption.

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

5.2 Equivalence with effect observations

As illustrated with examples in the previous section, Dijkstra monads can be obtained from effect observations $\theta : M \rightarrow W$ between computational and specification monads. As we shall see this construction is generic and leads to a categorical equivalence between Dijkstra monads and effect observations. In order to compare this notion of Dijkstra monads to effect observations, we also introduce a category of monadic relations MonRel and show that there is an adjunction

$$\int \dashv \text{pre} : \text{MonRel} \longrightarrow \mathcal{D}\text{Mon}. \quad (5.1)$$

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

⁶These effect polymorphism last examples are written in F^* syntax, but only implemented in Coq, since Dijkstra monads are not first class in F^* .

The discrete setting In this paragraph we take all specification monads W to be slightly degenerated, namely discrete. Given a monadic relation $\mathcal{R} : M \leftrightarrow W$ (Def. 4.3.1) between a computational monad M and a specification monad W , we construct a Dijkstra monad $\text{pre } \mathcal{R}$ on W as follows:

$$(\text{pre } \mathcal{R}) A (w : W A) = (m : M A) \times m \mathcal{R}_A w \quad (5.2)$$

That is $(\text{pre } \mathcal{R}) A w$ consists of those elements m of $M A$ that are related by \mathcal{R} to the specification w . When \mathcal{R} is the graph of a monad morphism θ (or equivalently, \mathcal{R} is functional), $\text{pre}(\mathcal{R} : M \leftrightarrow W)$ maps an element $w : W A$ to its *preimage* $\theta^{-1}(w) = \{m : M A \mid \theta(m) = w\}$. The return and bind operations of $\text{pre } \mathcal{R}$ are given by the return and bind operations of M , using the compatibility of \mathcal{R} with respect to the monad operations of M and W . The weakening operation is just the identity since the order on W is assumed to be trivial.

Conversely, any Dijkstra monad \mathcal{D} over a specification monad W with discrete order yields a monad structure on

$$\int \mathcal{D} A = (w : W A) \times \mathcal{D} A w \quad (5.3)$$

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

In order to explain the relation between these two operations pre and $\int -$, we introduce the category MonRel of monadic relations.

Definition 5.2.1. *The category MonRel of monadic relations consists of:*

- ▷ *An object of MonRel is a pair of monads M, W together with a monadic relation $\mathcal{R} : M \leftrightarrow W$ between them.*
- ▷ *A morphism between $\mathcal{R}^1 : M_1 \leftrightarrow W_1$ and $\mathcal{R}^2 : M_2 \leftrightarrow W_2$ is a pair (Θ^M, Θ^W) of a monad morphism $\Theta^M : M_1 \rightarrow M_2$ and a specification monad morphism $\Theta^W : W_1 \rightarrow W_2$ such that*

$$\begin{array}{ccc} M_1 A & \xrightarrow{\Theta_A^M} & M_2 A \\ \mathcal{R}_A^1 \updownarrow & \Rightarrow & \downarrow \mathcal{R}_A^2 \\ W_1 A & \xrightarrow{\Theta_A^W} & W_2 A \end{array}$$

$$\Theta_A^{\mathcal{R}} : \forall (m : M A) (w : W A). m \mathcal{R}_A^1 w \implies \Theta^M(m) \mathcal{R}_A^2 \Theta^W(w). \quad (5.4)$$

The construction pre extends to a functor on MonRel : it sends a morphism $(\Theta^W, \Theta^M) : \mathcal{R}^1 \rightarrow \mathcal{R}^2$ between monadic relations to a Dijkstra monad morphism $(\Theta^W, \Theta^{\mathcal{D}}) : \text{pre } \mathcal{R}^1 \rightarrow \text{pre } \mathcal{R}^2$, where $\Theta_{A,w}^{\mathcal{D}}$ is defined as the restriction of Θ_A^M to the appropriate domain

$$\Theta_{A,w} : \begin{cases} (m : M_1 A) \times m \mathcal{R}_A^1 w & \longrightarrow & (m : M_2 A) \times m \mathcal{R}_A^2 (\Theta^W w) \\ (m, pf) & \longmapsto & (\Theta^M m, \Theta^{\mathcal{R}} pf) \end{cases}$$

Conversely, \int packs up a Dijkstra monad morphism $(\Theta^W, \Theta^{\mathcal{D}}) : (W_1, \mathcal{D}_1) \rightarrow (W_2, \mathcal{D}_2)$ as monadic relation morphism (Θ^W, Θ^M) , where

$$\Theta_A^M : \begin{cases} (w : W_1 A) \times \mathcal{D}_1 A w & \longrightarrow & (w : W_2 A) \times \mathcal{D}_2 A w \\ (w, m) & \longmapsto & (\Theta_A^W w, \Theta_{A,w}^{\mathcal{D}} m) \end{cases}$$

Since Θ^M maps the inverse image of w – pairs whose first component is w – to the inverse image of $\Theta^W(w)$ – pairs whose first component is $\Theta^W(w)$ –, condition (5.4) holds. Moreover, this gives rise to a natural bijection

$$\text{MonRel}(\int \mathcal{D}, \mathcal{R}) \cong \mathcal{D}\text{Mon}(\mathcal{D}, \text{pre } \mathcal{R})$$

that establishes the adjunction (5.1). We can restrict (5.1) to an equivalence by considering only those objects for which the unit (resp. counit) of the adjunction is an isomorphism. Every Dijkstra monad \mathcal{D} is isomorphic to its image $\text{pre}(\int \mathcal{D})$, whereas a monadic relation \mathcal{R} is isomorphic to $\int(\text{pre } \mathcal{R})$ if and only if it is functional, i.e., a monad morphism. This way we obtain an equivalence of categories between \mathcal{DMon} and the category of effect observations on monads with discrete preorder.

The ordered setting We now consider the general case of specification monads equipped with an arbitrary order, that is monadic relations $\mathcal{R} : M \leftrightarrow W$ where M is a (plain,discrete) monad and W a specification monad. The definition of pre (5.2) still makes sense but for the weakening operation: the Dijkstra monad $\text{pre } \mathcal{R}$ has a weakening operation exactly when \mathcal{R} is *monotonic* Def. 4.3.1 with respect to the order. Hence, we restrict our attention to the category MonRel^{\leq} of monotonic monadic relations. Doing so, we obtain a functor $\text{pre} : \text{MonRel}^{\leq} \rightarrow \mathcal{DMon}$ from the category of monotonic monadic relations to the category of Dijkstra monads.

The special case of a monad morphism $\theta : M \rightarrow W$ fits well in this picture as long as we consider the associated monotonic monadic relation \mathcal{R}_θ defined by $m \mathcal{R}_\theta w \iff \theta m \leq^W w$. The corresponding weakening structure on $\text{pre } \mathcal{R}_\theta$ is given by

$$\text{weaken}\langle w_1 \leq w_2, \langle m, \theta(m) \leq w_1 \rangle \rangle = \langle m, \theta(m) \leq w_1 \leq w_2 \rangle.$$

Building and explicitly describing a left adjoint \int to pre turns out to be slightly more difficult. To explain where the problem lies, consider the case of a monad morphism $\theta : M \rightarrow W$ for which we expect $\int(\text{pre } \theta)$ to be isomorphic to θ . However, using straightforwardly the previous definition of \int , $\int(\text{pre } \theta)$ is just $(\Sigma M, W, \pi_1)$ where $\Sigma M A = (w : W A) \times (m : M A) \times \theta_A(m) \leq_A w$, which is far from being isomorphic to M . The problem is that we get one copy of m for each admissible specification $w : W A$. These copies, however, are non-essential since the weakening structure of $\text{pre } \theta$ identifies them. As such, a reasonable definition of \int in the ordered setting need to further quotient them⁷. Consequently, we define $\int \mathcal{D}$ as the monotonic monadic relation $\int \mathcal{D} = (\int \mathcal{D}, W, \mathcal{R}_{\int \mathcal{D}})$ with

$$\int \mathcal{D} A = ((w : W A) \times \mathcal{D} A w) / \sim \quad [w, c] \mathcal{R}_{\int \mathcal{D}} w' \iff w \leq w'$$

where \sim is the equivalence relation induced by $(w, c) \sim (w', \text{weaken}(w \leq w', c))$ and $[w, c]$ is the equivalence class of the pair (w, c) in $\int \mathcal{D}$. The monad structure on $A \mapsto (w : W A) \times \mathcal{D} A w$ induces a monad structure on the quotient $\int \mathcal{D}$ because bind^W is monotonic in both arguments and $\text{bind}^{\mathcal{D}}$ is compatible with the weakening structure in both arguments as well. This definition reduces to Equation 5.3 when the order on W is discrete.

Theorem 5.2.1. *The categories of Dijkstra monads and monadic relations are connected by an adjunction*

$$\int \dashv \text{pre} : \text{MonRel}^{\leq} \rightarrow \mathcal{DMon}.$$

Moreover, restricting our attention to specification monads W such that any two elements in $W A$ has an upper bound for \leq^W , the adjunction induces an equivalence in the following cases:

- ▷ the counit $\varepsilon_{\mathcal{R}} : \int(\text{pre } \mathcal{R}) \rightarrow \mathcal{R}$ is invertible if $\mathcal{R} = \mathcal{R}_\theta$ for $\theta : M \rightarrow W$ a (lax) monad morphism⁸.
- ▷ the unit $\eta_{\mathcal{D}} : \mathcal{D} \rightarrow \text{pre}(\int \mathcal{D})$ is invertible whenever $\text{weaken}^{\mathcal{D}}(w \leq^W w') : \mathcal{D} A w \rightarrow \mathcal{D} A w'$ is one-to-one for any A and $w \leq w' : W A$.

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

⁸By a lax monad morphism $\theta : M \rightarrow W$, we mean a lax natural transformation θ – that is such that the naturality square commutes up to the order on W – preserving the monadic operations in the weaker sense that $\theta(\text{ret}^M) \leq \text{ret}^W$ and $\theta(\text{bind}^M m f) \leq \text{bind}^W(\theta m)(\theta \circ f)$.

Proof. The definition of f given on Dijkstra monads extends Dijkstra monad morphisms: $(\Theta^W, \Theta^D) : (W_1, \mathcal{D}_1) \rightarrow (W_2, \mathcal{D}_2)$ pairs up to provide a monad morphism $\langle \Theta_A^W, \Theta_A^D \rangle : ((w : W_1 A) \times \mathcal{D}_1 A w) \rightarrow ((w : W_2 A) \times \mathcal{D}_2 A w)$, and since θ^W is monotonic with respect to the orders on W_1, W_2 , this natural transformation descend to the quotient by \sim as $\Theta^{fD} : f\mathcal{D}_1 \rightarrow f\mathcal{D}_2$ and we define $f(\Theta^W, \Theta^D) = (\Theta^W, \Theta^{fD})$, the condition (5.4) being immediate. We check that this assignment is functorial, giving rise to a functor $f : \mathcal{DMon} \rightarrow \mathcal{MonRel}^{\leq}$.

We now turn to the construction of natural transformations

$$\begin{array}{ccc} \varphi & : & \mathcal{MonRel}(f\mathcal{D}_1, \mathcal{R}_2) \xrightarrow{\sim} \mathcal{DMon}(\mathcal{D}_1, \mathbf{pre} \mathcal{R}_2) \\ & & \mathcal{MonRel}(f\mathcal{D}_1, \mathcal{R}_2) \xleftarrow{\sim} \mathcal{DMon}(\mathcal{D}_1, \mathbf{pre} \mathcal{R}_2) : \psi \end{array}$$

for (W_1, \mathcal{D}_1) a Dijkstra monad and $(M_2, W_2, \mathcal{R}_2)$ a monadic relation.

For $(\Theta^M, \Theta^W) \in \mathcal{MonRel}(f\mathcal{D}_1, \mathcal{R}_2)$, that is $\Theta^M : f\mathcal{D}_1 \rightarrow M_2$ and $\Theta^W : W_1 \rightarrow W_2$, we set $\varphi(\Theta^M, \Theta^W) = (\Theta^W, \Theta^D)$ where Θ^D is the Dijkstra monad morphism defined by

$$\begin{array}{ccc} \Theta_{A,w}^D & : & \mathcal{D}_1 A w \longrightarrow (m : M_2 A) \times m \mathcal{R}_2 (\Theta^W w) \\ c & \longmapsto & (\Theta^M[w, c], _) \end{array}$$

In this definition and the proof of the Dijkstra monad morphism laws below, we leave implicit the witness that the relation \mathcal{R}_2 hold obtained from the condition (5.4).

$$\Theta_{A, \mathbf{ret}^{W_1} a}^D (\mathbf{ret}^{\mathcal{D}_1} a) = \Theta^M[\mathbf{ret}^{W_1} a, \mathbf{ret}^{\mathcal{D}_1} a] = \Theta^M(\mathbf{ret}^{f\mathcal{D}_1} a) = \mathbf{ret}^{M_2} a$$

$$\begin{aligned} \Theta_{A, \mathbf{bind}^{W_1} w^m w^f}^D (\mathbf{bind}^{\mathcal{D}_1} m f) &= \Theta^M[\mathbf{bind}^{W_1} w^m w^f, \mathbf{bind}^{\mathcal{D}_1} m f] \\ &= \Theta^M(\mathbf{bind}^{f\mathcal{D}_1} [w^m, m] (\lambda x. [w^f x, f x])) \\ &= \mathbf{bind}^{f\mathcal{D}_1} \Theta^M[w^m, m] (\lambda x. \Theta^M[w^f x, f x]) \\ &= \mathbf{bind}^{f\mathcal{D}_1} \Theta_{A, w^m}^D (m) (\lambda x. \Theta_{A, w^f x}^D (f x)) \end{aligned}$$

$$\begin{aligned} \Theta_{A, w'}^D (\mathbf{weaken} (w \leq w', m)) &= \Theta^M[w', \mathbf{weaken} (w \leq w', m)] \\ &= \mathbf{weaken} (\Theta^W w \leq \Theta^W w', \Theta^M[w, m]) \\ &= \mathbf{weaken} (\Theta^W w \leq \Theta^W w', \Theta_{A, w}^D m). \end{aligned}$$

In the other direction, a Dijkstra monad morphism $(\Theta^W, \Theta^D) \in \mathcal{DMon}(\mathcal{D}_1, \mathbf{pre} \mathcal{R}_2)$ is sent to the morphism of monadic relations $\psi(\Theta^W, \Theta^D) = (\Theta^M, \Theta^W)$ where Θ^M is the monad morphism mapping an equivalence class $[w, c] \in f\mathcal{D}_1 A$ to $\Theta^M[w, c] = \pi_1(\Theta_{A, w}^D c)$ which is well defined because Θ^W is monotonic and Θ^D is compatible with \mathbf{weaken} . The condition (5.4) is provided by the second component of Θ^D .

Checking that φ and ψ are inverse to each other is straightforward (in an intensional setting, such as Coq, extensionality of functions and products are needed, and we also assumed uniqueness of propositions).

From this concrete description of the adjunction $f \dashv \mathbf{pre}$, we obtain the explicit formula for the unit η and counit ε . Considering respectively a Dijkstra monad (W, \mathcal{D}) and a monadic relation (M, W, \mathcal{R}) , we have $\eta_{\mathcal{D}} = (\text{Id}_W, \eta_{\mathcal{D}}^D)$ and $\varepsilon_{\mathcal{R}} = (\varepsilon_{\mathcal{R}}^M, \text{Id}_W)$ where

$$\begin{array}{ccc} \eta_{\mathcal{D}}^D & : & \mathcal{D} A w \longrightarrow \mathbf{pre} (f\mathcal{D}) A w & \varepsilon_{\mathcal{R}}^M & : & f(\mathbf{pre} \mathcal{R}) A \longrightarrow M A \\ c & \longmapsto & ([w, c], _) & & & [w, (m, _)] \longmapsto m \end{array}$$

In the case of $\varepsilon_{\mathcal{R}}$, if $\mathcal{R} = \mathcal{R}_{\theta}$ is the monadic relation associated to a possibly lax monad morphism, then we have a section $(\varepsilon_{\mathcal{R}}^M)^{-1}$ mapping $m : M A$ to $[\theta(m), (m, _)]$. Taking $c = [w, (m, _)] \in f(\mathbf{pre} \mathcal{R}) A$, we can find $w' \in W A$ such that $w \leq^W w'$ and $w \leq^W \theta m$ so we have $c = [w, (m, _)] = [\theta m, (m, _)] = (\varepsilon_{\mathcal{R}}^M)^{-1} m$ and $(\varepsilon_{\mathcal{R}}^M)^{-1}$ is onto, so $\varepsilon_{\mathcal{R}}$ is invertible.

For the unit, $\eta_D^{\mathcal{D}}$ is clearly onto. It is also one-to-one when **weaken** is: if $([w, c], _) = \eta_D^{\mathcal{D}}(c) = \eta_D^{\mathcal{D}}(c') = ([w, c'], _)$ for $c, c' \in \mathcal{D} A w$ and $w \in W A$, then a straightforward induction on the length of the witness that $(w, c) \sim (w, c')$ using the fact that $W A$ has upper bounds of pairs of elements prove that there exists a $w' \geq w$ such that $\text{weaken}(w \leq w', c) = \text{weaken}(w \leq w', c')$, so $c = c'$ by injectivity of **weaken**. \square

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

5.3 Dijkstra monads as relative monads, connection to graded monads

In the same way specification monads can be understood as a particular kind of order-enriched relative monads (see Def. 3.5.2), we explain in this section how Dijkstra monads themselves can be framed as (order enriched) relative monads. This point of view on Dijkstra monads provide an interesting bridge to the notion of graded monad (Fujii et al., 2016), another algebraic structure refining monads with an index, capturing for instance resource usage (cost analysis). The latter can be used to model type-and-effects system (Katsumata, 2014).

In order to introduce the relative monad presentation of Dijkstra monads, we first reformulate Def. 5.1.1 in terms of indexed families. A family indexed by a set A is a function $B \rightarrow A$, the component B_a at index $a \in A$ being the fiber of this function at a . index $a \in A$ being the fiber of this function at a . A Dijkstra monad \mathcal{D} over a specification monad W induces an indexed family $\partial_A : \mathcal{D} A \rightarrow W A$ where $\mathcal{D} A = (w : W A) \times \mathcal{D} A w$ and ∂_A is the first projection. Then the for any type A , the return operations of W and \mathcal{D} make the diagram on the left commute, whereas bind^W and $\text{bind}^{\mathcal{D}}$ induce the function on the right:

$$\begin{array}{ccc}
 A \xrightarrow{\text{ret}^{\mathcal{D}}} \mathcal{D} A & & A \xrightarrow{f} \mathcal{D} B \\
 id \downarrow & & id \downarrow \\
 A \xrightarrow{\text{ret}^W} W A & & A \xrightarrow{w_f} W B
 \end{array}
 \quad \mapsto \quad
 \begin{array}{ccc}
 \mathcal{D} A \xrightarrow{\text{bind}^{\mathcal{D}} f} \mathcal{D} B & & \\
 id \downarrow & & \downarrow \partial_B \\
 W A \xrightarrow{\text{bind}^W w_f} W B & &
 \end{array}$$

Taking into account the order on W and the **weaken** operation, the Dijkstra monad \mathcal{D} can be equivalently described by giving the following data:

- ▷ for each type A , an indexed family $\partial_A : \mathcal{D} A \rightarrow W A$ between orders such that ∂_A is monotonic and its fibers discrete,
- ▷ for each type A , a map of indexed families $(\text{ret}^W, \text{ret}^{\mathcal{D}}) : id_A \rightarrow \partial_A$,
- ▷ for each pair of type A, B and map of families $\mathbf{f} = (w_f, f) : id_A \rightarrow \partial_B$, a monotonic extension $(\text{bind}^W w_f, \text{bind}^{\mathcal{D}} f) : \partial_A \rightarrow \partial_B$ also monotonic in \mathbf{f} (for the pointwise order on $id_A \rightarrow \partial_B$),
- ▷ satisfying laws analogous to the monadic laws

This reformulation exhibits a Dijkstra monads \mathcal{D} over the monad W as a relative monad where the domain category consists of sets – the type A of returned values – and the codomain consists of families indexed over a preorder, namely $W A$. To be more precise, let DiscPosFib be the full subcategory of Pos^{\rightarrow} whose objects are triples $(E, B, f : E \rightarrow B)$ consisting of ordered

sets E, B and a discrete fibration f between them, that is a monotone map f such that for any $b_1 \leq b_2$ and $e_1 \in f^{-1}(b_1)$ there is a unique $e_2 \geq e_1$ with $f(e_2) = b_2$ – in particular the fibers $f^{-1}(b)$ are discrete for all $b \in B$. Explicitly, the morphisms of DiscPosFib between (E, B, f) and (E', B', f') is a pair of monotone functions $(h^{\text{dom}} : E \rightarrow E', h^{\text{cod}} : B \rightarrow B')$ such that the following square commutes:

$$\begin{array}{ccc} E & \xrightarrow{h^{\text{dom}}} & E' \\ f \downarrow & & \downarrow f' \\ B & \xrightarrow{h^{\text{cod}}} & B' \end{array}$$

The Dijkstra monad \mathcal{D} then induces a functor $\mathcal{F}^{\mathcal{D}}$ with a monad structure relative to the base functor \mathcal{J}^{id} , all these objects being enriched over Pos (in the terminology of [chapter 3](#), we are working inside the framed bicategory Pos-Distr).

$$\mathcal{J}^{\text{id}} : \begin{cases} \text{Set} & \longrightarrow & \text{DiscPosFib} \\ A & \longmapsto & (id_A : A \rightarrow A) \end{cases} \quad \mathcal{F}^{\mathcal{D}} : \begin{cases} \text{Set} & \longrightarrow & \text{DiscPosFib} \\ A & \longmapsto & (\pi_1 : (w : W A) \times \mathcal{D} A w \rightarrow W A) \end{cases}$$

Moreover this presentation of \mathcal{D} as relative monad can be closely related to the relative monad structure on W (since it is a specification monad). Consider the projection functor $\text{cod} : \text{DiscPosFib} \rightarrow \text{Pos}$ sending a family (E, B, f) to its indexing base B . Together with the identity functor on Set , it fits into a morphism of base functor $(\text{Id}_{\text{Set}}, \text{cod}) : \mathcal{J}^{\text{id}} \rightarrow \text{Disc}$ (illustrated on the left) and the identity natural transformation is a relative monad morphism from $\mathcal{F}^{\mathcal{D}}$ to W over $(\text{Id}_{\text{Set}}, \text{cod})$.

$$\begin{array}{ccc} \text{Set} & \xrightarrow{\mathcal{J}^{\text{id}}} & \text{DiscPosFib} \\ \text{Id} \downarrow & & \downarrow \text{cod} \\ \text{Set} & \xrightarrow{\text{Disc}} & \text{Pos} \end{array} \quad \begin{array}{ccc} \text{Set} & \xrightarrow{\mathcal{F}^{\mathcal{D}}} & \text{DiscPosFib} \\ \text{Id} \downarrow & & \downarrow \text{cod} \\ \text{Set} & \xrightarrow{W} & \text{Pos} \end{array} \quad (5.5)$$

Taking inspiration from ([Katsumata, 2005, 2013](#)) where the case of monads is studied, we say that $\mathcal{F}^{\mathcal{D}}$ is a *lifting* of the relative monad W along $(\text{Id}_{\text{Set}}, \text{cod}) : \mathcal{J}^{\text{id}} \rightarrow \text{Disc}$ to mean that $\mathcal{F}^{\mathcal{D}}$ is a \mathcal{J}^{id} -relative monad such that the identity is a relative monad morphism over $(\text{Id}_{\text{Set}}, \text{cod})$ to W .

Lemma 5.3.1. *A Dijkstra monad \mathcal{D} over a specification monad W is equivalent to a relative lifting of W along $(\text{Id}_{\text{Set}}, \text{cod}) : \mathcal{J}^{\text{id}} \rightarrow \text{Disc}$.*

Proof. The previous discussion shows that a Dijkstra monad \mathcal{D} indeed defines a suitable relative monad $\mathcal{F}^{\mathcal{D}}$.

Conversely, given a lifting \mathcal{F} of W along $(\text{Id}_{\text{Set}}, \text{cod})$, we reconstruct a Dijkstra monad $\mathcal{D}^{\mathcal{F}}$ over W by the formula $\mathcal{D}^{\mathcal{F}} A (w : W A) = f^{-1}(w)$ where $\mathcal{F}(A) = (E, B, f : E \rightarrow B)$ and $B = W A$ by the lifting condition. The return and bind operations on $\mathcal{D}^{\mathcal{F}}$ are provided by the relative monad structure on \mathcal{F} , using the lifting condition to show that they have the correct type, while the weaken operation is derived from the fact that f above is a fibration. \square

5.3.1 Graded monads

In this section, we present graded monads that were studied in ([Fujii et al., 2016](#)). Dijkstra monads and graded monads share a reasonable amount of similarities:

- ▷ both structure are indexed by an object equipped with a monoid structure, specification monads for the former, monoidal categories for the latter;
- ▷ effect observations were originally introduced in ([Katsumata, 2013](#)) in order to build graded monad, and [section 5.2](#) shows that they can be used as well for Dijkstra monads.

In order to compare the two structures, we restrict ourselves to the simple case of a monad on sets graded by a monoid.

Definition 5.3.1. A graded monad on *Set* graded by a monoid $(\mathcal{M}, *, e)$ is a lax-monoidal functor $\mathcal{G} : \mathcal{M} \rightarrow [\text{Set}, \text{Set}]$, that is:

- ▷ For each $m : \mathcal{M}$ and set A , a set $\mathcal{G} A m$
- ▷ For each $m : \mathcal{M}$, sets A, B and function $f : A \rightarrow B$ a functorial action $\mathcal{G} A m \rightarrow \mathcal{G} B m$
- ▷ For each set A , a unit $A \rightarrow \mathcal{G} A e$ natural in A
- ▷ For each $m_1, m_2 : \mathcal{M}$ and set A , a multiplication $\mathcal{G}(\mathcal{G} A m_1) m_2 \rightarrow \mathcal{G} A (m_1 * m_2)$ natural in A
- ▷ satisfying laws analogous to the monoid laws

We wish to understand better the connection between the two indexed algebraic structure and to that end, we reformulate graded monads in terms of indexed families. As usual, the functorial action and multiplication can alternatively be reformulated in terms of Kleisli extension: for each $m : \mathcal{M}$, set A, B and function $A \rightarrow \mathcal{G} B m$, an extension $\mathcal{G} A m' \rightarrow \mathcal{G} B (m * m')$. We obtain the following presentation of a graded monad \mathcal{G} graded by a monoid \mathcal{M} .

- ▷ For each set A , an indexed family $\tau_A : \mathcal{G} A \rightarrow \mathcal{M}$
- ▷ For each set A , a map of indexed families $(e, \text{ret}^{\mathcal{G}}) : !_A \rightarrow \tau_A$

$$\begin{array}{ccc} A & \xrightarrow{\text{ret}^{\mathcal{G}}} & \mathcal{G} A \\ !_A \downarrow & & \downarrow \tau_A \\ \mathbb{1} & \xrightarrow{e} & \mathcal{M} \end{array}$$

- ▷ For each sets A, B and map of families $(m, f) : !_A \rightarrow \tau_B$, an extension $(m * -, \text{bind}^{\mathcal{G}} f) : \tau_A \rightarrow \tau_B$

$$\begin{array}{ccc} A & \xrightarrow{f} & \mathcal{G} B \\ !_A \downarrow & & \downarrow \tau_B \\ \mathbb{1} & \xrightarrow{m} & \mathcal{M} \end{array} \quad \mapsto \quad \begin{array}{ccc} \mathcal{G} A & \xrightarrow{\text{bind}^{\mathcal{G}} f} & \mathcal{G} B \\ \tau_A \downarrow & & \downarrow \tau_B \\ \mathcal{M} & \xrightarrow{m * -} & \mathcal{M} \end{array}$$

This reformulation makes it clear that we can see a monad \mathcal{G} graded by a monoid \mathcal{M} as a functor $\mathcal{F}^{\mathcal{G}} : \text{Set} \rightarrow \text{Set}^{\rightarrow}$ equipped with a monad structure relative to the functor $\mathcal{J}^!$:

$$\mathcal{J}^! : \left\{ \begin{array}{ccc} \text{Set} & \longrightarrow & \text{Set}^{\rightarrow} \\ A & \longmapsto & (!_A : A \rightarrow \mathbb{1}) \end{array} \right. \quad \mathcal{F}^{\mathcal{G}} : \left\{ \begin{array}{ccc} \text{Set} & \longrightarrow & \text{Set}^{\rightarrow} \\ A & \longmapsto & (\pi_1 : (m : \mathcal{M}) \times \mathcal{G} A m \rightarrow \mathcal{M}) \end{array} \right.$$

Moreover, this $\mathcal{J}^!$ -relative monad structure on $\mathcal{F}^{\mathcal{G}}$ lifts the monoid structure on \mathcal{M} (that can alternatively seen as a monad relative to $\mathbb{1} : \mathbb{1} \rightarrow \text{Set}$) through the morphism of base functors $(!_{\text{Set}}, \text{cod}) : \mathcal{J}^! \rightarrow \mathbb{1}$, where we write $!_{\text{Set}} : \text{Set} \rightarrow \mathbb{1}$ for the unique functor to the terminal category and $\mathbb{1} : \mathbb{1} \rightarrow \text{Set}$ for the functor picking a terminal object in *Set*.

$$\begin{array}{ccc} \text{Set} & \xrightarrow{\mathcal{J}^!} & \text{Set}^{\rightarrow} \\ !_{\text{Set}} \downarrow & & \downarrow \text{cod} \\ \mathbb{1} & \xrightarrow{\mathbb{1}} & \text{Set} \end{array} \quad \begin{array}{ccc} \text{Set} & \xrightarrow{\mathcal{F}^{\mathcal{G}}} & \text{Set}^{\rightarrow} \\ !_{\text{Set}} \downarrow & & \downarrow \text{cod} \\ \mathbb{1} & \xrightarrow{\mathcal{M}} & \text{Set} \end{array} \quad (5.6)$$

This analysis of graded monads shows that they admit a similar formal presentation in terms of lifting of relative monad structure as Dijkstra monads do (see diagrams 5.5 and 5.6). Since both kinds of algebraic structure are useful for verification purposes, and they do not seem to reduce to each other, it would be interesting as future work to investigate further the common structure provided by relative monad liftings.

5.4 Conclusion & Related work

The work presented in this section directly builds on prior work on Dijkstra monads in F^* (Swamy et al., 2013, 2016), in particular the *DM4Free* approach (Ahman et al., 2017). Our investigation of Dijkstra monad was primarily motivated by syntactic conditions required in the *DM4Free* approach that were at the time quite obscure, and prevented us from obtaining Dijkstra monads for some effects, e.g., IO. The construction of [section 5.2](#) together with those of [chapter 4](#) shed some light on these awkward restrictions: the Dijkstra monads derivable with *DM4Free* are those obtained from effect observations with shape

$$\mathcal{T}(\text{ret}^{W^{\text{Pure}}}) : \mathcal{T}(\text{Id}) \longrightarrow \mathcal{T}(W^{\text{Pure}})$$

where \mathcal{T} is a monad transformer and $\text{ret}^{W^{\text{Pure}}} : \text{Id} \rightarrow W^{\text{Pure}}$ the canonical monad morphism from the identity monad.

Katsumata (2014) uses graded monads to give semantics to type-and-effect systems, introduces effect observations as monad morphisms, and constructs graded monads out of effect observations by restricting the specification monads to their value at $\mathbb{1}$. We extend his construction to Dijkstra monads, showing that they are equivalent to effect observations, and unify Katsumata’s two notions of algebraic operation. We sketched a common framework for graded monads and Dijkstra monad but further investigation of such indexed structure obtained by lifting a relative monad remains to be done. In particular, generic lifting methods for monads such as the monadic $\top\top$ -lifting (Katsumata, 2013) or the codensity lifting (Katsumata et al., 2018) may also extend to the case of relative monads.

Dijkstra monads & monotonicity A long line of work in F^* uses *monotonicity* to alleviate the proof effort in protocols and state intensive programs. The soundness of these reasoning principles rely on some abstraction properties: intuitively state can only be used linearly, so restricting the state updates to be monotonic with respect to a chosen order implies that any monotonic predicate witnessed at some program point will necessarily hold at any later point, independently of the actual state. Ahman et al. (2018) explain how it combines swiftly with the abstract approach to computations provided by Dijkstra monads. A general account of these monotonicity arguments for arbitrary effects, for instance for IO, is left for future work.

Dijkstra monads as displayed algebras Kaposi and Kovács (2019) propose a framework to define expressive signatures and to associate to each signature Σ a category with families consisting of Σ -algebras, Σ -algebra homomorphisms and displayed Σ -algebras. These signature can be used to capture the notion of Dijkstra monad (but for the order) in a concise way: Dijkstra monads arise as display algebras of a signature Σ^{mon} . Concretely, Kovács proposed (in private communication) the following signature Σ^{mon} to capture Dijkstra monads:

$$\begin{aligned} \text{M} & : \text{Set} \Rightarrow \mathbb{U}, \\ \text{ret} & : (A : \text{Set}) \Rightarrow A \Rightarrow \mathbb{E}(\text{M } A), \\ (-)^\dagger & : (AB : \text{Set}) \Rightarrow (\Pi_A \text{M } B) \Rightarrow \text{M } A \Rightarrow \mathbb{E}(\text{M } A), \\ \text{bind-ret} & : (A : \text{Set}) \Rightarrow (m : \text{M } A) \Rightarrow \text{Id } (\text{M } A) (\text{ret}^\dagger m) m, \\ \text{ret-bind} & : (A B : \text{Set})(x : A) \Rightarrow (f : \Pi_A \text{M } B) \Rightarrow \text{Id } (\text{M } B) (f^\dagger (\text{ret } x)) (f x), \\ \text{bind-assoc} & : (A B C : \text{Set}) \Rightarrow (m : \text{M } A)(f : \Pi_A \text{M } B)(g : \Pi_B \text{M } C) \\ & \quad \Rightarrow \text{Id } (\text{M } C) (g^\dagger (f^\dagger m)) ((\lambda x. g^\dagger (f x))^\dagger m) \end{aligned}$$

Here Π is the constructor for infinitary (A -indexed for any $\text{Set } A$) products. Taking models of this signature in the CwF of sets and families gives monads on Set , and unary logical predicate gives the notion of Dijkstra monad without weakening. In this simplified setting, the equivalence between Dijkstra monads and monad morphisms is then a consequence of the CwF structure. An extension of this framework to the ordered setting might provide a simpler and abstract variant of the proof in [section 5.2](#).

Chapter 6

Relational reasoning

«- Ça n'est pas une question scientifique...
- Moi, je vais te questionner Einstein, et si tu ne peux
répondre, tout s'éteindra !!
[...] Quel est mon nom ?
Et comme il ne pouvait répondre, tout bascula dans le néant.»

F'murr, Le Génie des alpages n°5, les intondables

This chapter is dedicated to relational reasoning, i.e., proving relational properties between multiple runs of one or more programs, such as noninterference or program equivalence (see [section 1.3](#)). Our goal is to distill the generic relational reasoning principles that work for *arbitrary* monadic effects so that we can reconstruct relational program logics in a generic fashion. Reusing our knowledge from the unary setting, we devise *relational* variants of specification monads and effect observations providing the semantics of relational judgements.

6.1 The logic of relational rules

In this section, we make the simple but useful observation that rules of relational programs logics can be organised in 3 categories depending on their relationship to effects and then provide a high-level idea of how it leads to our generic relational framework. Exceptional control flows are surprisingly challenging in the relational setting, and we need the full power of our framework to account for them (see [section 6.3](#)). However a much simpler system already account for many effects and we use it to provide a smoother approach in [section 6.2](#).

6.1.1 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 (section 6.2), the judgment of the relational program logics of our framework has the shape: $\vdash c_1 \sim c_2 \{ w \}$, where $c_1 : M_1 A_1$ 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 $W_{\text{rel}}(A_1, A_2)$. Here M_1 and M_2 are two arbitrary and *potentially distinct* computation monads (e.g., the state monad $\text{St } A = S \rightarrow A \times S$ and the exception monad $\text{Exc } A = A + E$), while w could, for instance, be a pair of a relational precondition and a relational postcondition, or a relational predicate transformer—below we will use relational weakest preconditions. For instance, for relating two stateful monads on states S_1 and S_2 , we often use relational specifications drawn from

$$W_{\text{rel}}^{\text{St}}(A_1, A_2) = ((A_1 \times S_1) \times (A_2 \times S_2) \rightarrow \mathbb{P}) \rightarrow S_1 \times S_2 \rightarrow \mathbb{P}$$

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 ambient dependent type theory). As an example of the judgment above, consider the programs $c_1 = \text{bind}^{\text{St}} (\text{get } ()) (\lambda x. \text{put } (x + k))$, which increments the content of a memory cell, and $c_2 = \text{ret}^{\text{St}} ()$, which does nothing. These two programs are related by the specification $w = \lambda \varphi (s_1, s_2). \varphi (((), s_1 + k), ((), s_2)) : W_{\text{rel}}^{\text{St}}(\mathbb{1}, \mathbb{1})$ 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 save 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 section 6.2) provides the following rule for the if-then-else construct:

$$\frac{\text{if } b \text{ then } \vdash c_1 \sim c_2 \{ w^\top \} \text{ else } \vdash c_1 \sim c_2 \{ w^\perp \}}{\vdash c_1 \sim c_2 \{ \text{if } b \text{ then } w^\top \text{ else } w^\perp \}}$$

In order to prove that c_1 and c_2 satisfy the relational specification $\text{if } b \text{ then } w^\top \text{ else } w^\perp$, 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 $\text{if } b \text{ then } c_1^\top \text{ else } c_1^\perp$ 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:

$$\frac{\vdash c_1^\top \sim c_2 \{ w^\top \} \quad \vdash c_1^\perp \sim c_2 \{ w^\perp \}}{\vdash \text{if } b \text{ then } c_1^\top \text{ else } c_1^\perp \sim c_2 \{ \text{if } b \text{ then } w^\top \text{ else } w^\perp \}} \quad (6.1)$$

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

$$\frac{\vdash c_1^\top \sim c_2^\top \{ w^\top \} \quad \vdash c_1^\perp \sim c_2^\perp \{ w^\perp \}}{\vdash \text{if } b_1 \text{ then } c_1^\top \text{ else } c_1^\perp \sim \text{if } b_2 \text{ then } c_2^\top \text{ else } c_2^\perp \{ w^\bullet \}} \quad (6.2)$$

where the relational specification $w^\bullet = \lambda \varphi s_{12}. (b \Leftrightarrow b_1) \wedge (b \Leftrightarrow b_2) \wedge \text{if } b \text{ then } w^\top \varphi s_{12} \text{ else } w^\perp \varphi s_{12}$ 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 $\text{bind}^{M_1} m_1 f_1$ to $\text{bind}^{M_2} m_2 f_2$ 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:

$$\frac{\vdash m_1 \sim m_2 \{ w^m \} \quad \forall a_1, a_2 \vdash f_1 a_1 \sim f_2 a_2 \{ w^f (a_1, a_2) \}}{\vdash \text{bind}^{M_1} m_1 f_1 \sim \text{bind}^{M_2} m_2 f_2 \{ \text{bind}^{W_{\text{rel}}} w^m w^f \}} \quad (6.3)$$

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

$$\text{bind}^{W_{\text{rel}}} : W_{\text{rel}}(A_1, A_2) \rightarrow (A_1 \times A_2 \rightarrow W_{\text{rel}}(B_1, B_2)) \rightarrow W_{\text{rel}}(B_1, B_2) \quad (6.4)$$

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

$$\text{bind}^{W_{\text{rel}}^{\text{St}}} w^m w^f = \lambda \varphi (s_1, s_2). w^m (\lambda ((a_1, s'_1), (a_2, s'_2)). w^f (a_1, a_2) (s'_1, s'_2) \varphi) (s_1, s_2).$$

This construct is written in continuation passing style: the specification of the continuation w^f maps a postcondition $\varphi : (B_1 \times S_1) \times (B_2 \times S_2) \rightarrow \mathbb{P}$, to an intermediate postcondition $(A_1 \times S_1) \times (A_2 \times S_2) \rightarrow \mathbb{P}$, 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 $\text{ret}^{M_1}()$ or f_1 to be ret^{M_1} above and using the monadic laws of M_1 (and symmetrically for M_2):

$$\frac{\vdash \text{ret}^{M_1}() \sim m_2 \{ w^m \} \quad \forall a_2 \vdash c_1 \sim f_2 a_2 \{ w^f a_2 \}}{\vdash c_1 \sim \text{bind}^{M_2} m_2 f_2 \{ \text{bind}^{W_{\text{rel}}} w^m (\lambda((), a_2). w^f a_2) \}} \quad (6.5)$$

$$\frac{\vdash c_1 \sim m_2 \{ w^m \} \quad \forall a_1, a_2 \vdash \text{ret}^{M_1} a_1 \sim f_2 a_2 \{ w^f (a_1, a_2) \}}{\vdash c_1 \sim \text{bind}^{M_2} m_2 f_2 \{ \text{bind}^{W_{\text{rel}}} w^m w^f \}} \quad (6.6)$$

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 :

$$\overline{\vdash \text{get}() \sim \text{ret } a_2 \{ w_{\text{get}^l} \}} \quad \overline{\vdash \text{ret } a_1 \sim \text{get}() \{ w_{\text{get}^r} \}} \quad (6.7)$$

$$\overline{\vdash \text{put } s \sim \text{ret } a_2 \{ w_{\text{put}^l} \}} \quad \overline{\vdash \text{ret } a_1 \sim \text{put } s \{ w_{\text{put}^r} \}} \quad (6.8)$$

where $w_{\text{get}^l} = \lambda \varphi (s_1, s_2). \varphi ((s_1, s_1), (a_2, s_2))$, $w_{\text{get}^r} = \lambda \varphi (s_1, s_2). \varphi ((a_1, s_1), (s_2, s_2))$, $w_{\text{put}^l} = \lambda \varphi (s_1, s_2). \varphi ((((), s), (a_2, s_2)))$ and $w_{\text{put}^r} = \lambda \varphi (s_1, s_2). \varphi ((a_1, s_1), (((), s)))$. 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:

$$\overline{\vdash \text{get}() \sim \text{get}() \{ w_{\text{get}} \}} \quad \overline{\vdash \text{put } s \sim \text{put } s' \{ w_{\text{put}} \}}$$

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

$$\frac{\vdash \text{ret}() \sim \text{get}() \{ w_{\text{get}^l} \} \quad \forall u : \mathbb{1}, s_2 : S_2 \vdash \text{get } u \sim \text{ret } s_2 \{ w_{\text{get}^r} \}}{\vdash \text{bind}^{\text{St}_{S_1}} (\text{ret}()) \text{get} \sim \text{bind}^{\text{St}_{S_2}} (\text{get}()) \text{ret}^{\text{St}_{S_2}} \{ \text{bind}^{W_{\text{rel}}^{\text{St}}} w_{\text{get}^l} (\lambda(u, s_2). w_{\text{get}^r}) \}} \quad (6.9)$$

where the last specification reduces to w_{get} using the definition of $\text{bind}^{W_{\text{rel}}^{\text{St}}}$.

6.1.2 Simple semantics

To define a semantics for the \vdash judgment above, we make the important observation that $W_{\text{rel}}(A_1, A_2)$ is a relative monad (see [chapter 3](#)) over the product $(A_1, A_2) \mapsto A_1 \times A_2$, as illustrated by the type of $\text{bind}^{W_{\text{rel}}}$ above (6.4), 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 $\theta_{\text{rel}} : M_1 A_1 \times M_2 A_2 \rightarrow W_{\text{rel}}(A_1, A_2)$ can immediately give us a semantics for the judgment above:

$$\models_{\theta_{\text{rel}}} c_1 \sim c_2 \{ w \} = \theta_{\text{rel}}(c_1, c_2) \leq w,$$

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

$$w' \leq^{W_{\text{rel}}^{\text{St}}} w = \forall \varphi s_1 s_2. w \varphi (s_1, s_2) \Rightarrow w' \varphi (s_1, s_2), \quad (6.9)$$

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

$$\models_{\theta_{\text{rel}}^{\text{St}}} c_1 \sim c_2 \{ w \} = \forall \varphi s_1 s_2. w \varphi (s_1, s_2) \Rightarrow \varphi (c_1 s_1, c_2 s_2)$$

Another important point is that the relational effect observation can help us in deriving simple effect-specific rules, such as the ones for `get` (6.7) and `put` (6.8) above. For deriving such rules, one first has to choose c_1 and c_2 (and we hope that the product programs of [section 6.4](#) can provide guidance on this in the future) and then one can simply compute the specification using θ . For instance, $w_{\text{get}^!} = \lambda \varphi (s_1, s_2). \varphi ((s_1, s_1), (a_2, s_2))$ in the first `get` rule (6.7) really is just $\theta(\text{get } (), \text{ret } a_2)$. This idea is further discussed in [subsection 6.2.5](#).

6.1.3 Exceptions, and why the simple semantics is not enough

While the simple construction we described so far works well for state, it does not work for exceptions. 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:

$$W_{\text{rel}}^{\text{Exc}}(A_1, A_2) = ((A_1 + E_1) \times (A_2 + E_2) \rightarrow \mathbb{P}) \rightarrow \mathbb{P}.$$

A predicate transformer $w : W_{\text{rel}}^{\text{Exc}}(A_1, A_2)$ maps an exception-aware postcondition $\varphi : (A_1 + E_1) \times (A_2 + E_2) \rightarrow \mathbb{P}$ 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 $\forall a_1, a_2, \vdash f_1 a_1 \sim f_2 a_2 \{ w^f(a_1, a_2) \}$ with specifications w^m, w^f drawn from $W_{\text{rel}}^{\text{Exc}}$. In order to build a composite proof relating $c_1 = \text{bind}^{\text{Exc}} m_1 f_1$ and $c_2 = \text{bind}^{\text{Exc}} m_2 f_2$ we need to be able to compose w^m and w^f in some way. If w^m ensures that m_1 and m_2 terminate both normally returning values, or throw an exception at the same time, we can compose with w^f or pass the exception 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 $W_{\text{rel}}^{\text{Exc}}$, this failure is an obstruction to complete the following tentative definition of $\text{bind}^{W_{\text{rel}}^{\text{Exc}}}$:

```

let bindWrelExc wm (wf : A1 × A2 → (((B1 + E1) × (B2 + E2)) → ℙ) → ℙ) (φ : (B1 + E1) × (B2 + E2) → ℙ) =
  wm (λx : (A1 + E1) × (A2 + E2).
    match x with
    | Inl a1, Inl a2 → wf a1 a2 φ
    | Inr e1, Inr e2 → φ (Inr e1, Inr e2)
    | _ → ??? )

```

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`:

```

let bindWrelExc  $w_m (w_{f_1} : A_1 \rightarrow ((B_1 + E_1) \rightarrow \mathbb{P}) \rightarrow \mathbb{P}) (w_{f_2} : A_2 \rightarrow ((B_2 + E_2) \rightarrow \mathbb{P}) \rightarrow \mathbb{P}) w_f \varphi =$ 
   $w_m (\lambda x : (A_1 + E_1) \times (A_2 + E_2).$ 
    match  $x$  with
    ...
    | Inl  $a_1, \text{Inr } e_2 \rightarrow w_{f_1} a_1 (\lambda be. \varphi be (\text{Inr } e_2))$ 
    | Inr  $e_1, \text{Inl } a_2 \rightarrow w_{f_2} a_2 (\lambda be. \varphi (\text{Inr } e_1) be)$ 
  )

```

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. It turns out that this `bindWrelExc` operation can still be used to define a relative monad, but in a more complex relational setting that we introduce in [section 6.3](#). 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). In practice we can still implement this relational dependent type theory inside our ambient type theory, in our case Coq, and continue using the same tools for verification.

6.2 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.

6.2.1 Specifications as (relative) monads

We extend the important idea from [section 2.3](#) of giving the same algebraic footing to both computations and specifications.

Moving to the relational setting, a relational specification for a pair of stateful computations $c_1 : \text{St}_{S_1} A_1$ and $c_2 : \text{St}_{S_2} A_2$ consist of a predicate transformer w mapping postconditions relating 2 pairs of a result value and a final state to a precondition relating 2 initial states, i.e.,

$$W_{\text{rel}}^{\text{St}}(A_1, A_2) = ((A_1 \times S_1) \times (A_2 \times S_2) \rightarrow \mathbb{P}) \rightarrow S_1 \times S_2 \rightarrow \mathbb{P}. \quad (6.10)$$

$W_{\text{rel}}^{\text{St}}$ does not posse the monad structure present on its unary variant. To begin with it is not even an endofunctor: it takes *two* types as input and produces one. However, the monadic operations of the unary variant do extend to the relational setting

```

let retWrelSt  $(a_1, a_2) : A_1 \times A_2 : W_{\text{rel}}^{\text{St}}(A_1, A_2) = \lambda \varphi (s_1, s_2). \varphi ((a_1, s_1), (a_2, s_2))$ 

```

```

let bindWrelSt  $(wm : W_{\text{rel}}^{\text{St}}(A_1, A_2)) (wf : A_1 \times A_2 \rightarrow W_{\text{rel}}^{\text{St}}(B_1, B_2)) : W_{\text{rel}}^{\text{St}}(B_1, B_2) =$ 
   $\lambda \varphi (s_1, s_2). wm (\lambda ((a_1, s_1'), (a_2, s_2')). wf(a_1, a_2) \varphi (s_1', s_2'))$ 

```

These operations satisfy equations analogs to the monadic ones and are part of a relative monad structure in the sense of Altenkirch et al. (2015) (see also [chapter 3](#)). The relational specifications for state $W_{\text{rel}}^{\text{St}}$ are also naturally ordered by $\leq^{W_{\text{rel}}^{\text{St}}}$ (see (6.9)) 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 transformer. We call such monad-like structure equipped with a compatible ordering a *simple relational specification monad*.

Definition 6.2.1. A simple relational specification monad consist of

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

$$bind^{W_{rel}}(ret^{W_{rel}}(a_1, a_2)) w^f = w^f(a_1, a_2) \quad bind^{W_{rel}} w^m ret^{W_{rel}} = w^m$$

$$bind^{W_{rel}}(bind^{W_{rel}} w^m w^f) w^g = bind^{W_{rel}} w^m (\lambda x. bind^{W_{rel}}(w^f x) w^g)$$

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

A simple way to produce various examples of simple relational specification monads besides W_{rel}^{St} is to start from a (non-relational) specification monad W , 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 $W_{rel}(A_1, A_2) = W(A_1 \times A_2)$ is a simple relational specification monad. In the following paragraphs, we illustrate this construction with a few concrete instances showing the flexibility of this notion. Depending on the property we want to verify, we can pick simpler or more sophisticated relational specification monads among these. For instance, relational specification monads based on pre-/postconditions make the connection to relational program logics in the literature more evident.

Backward predicate transformer A stateless version of W_{rel}^{St} is the predicate transformer

$$W_{rel}^{Pure}(A_1, A_2) = (A_1 \times A_2 \rightarrow \mathbb{P}) \rightarrow \mathbb{P}$$

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 [subsection 6.2.5](#).

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 from a relational specification monads. The type constructor

$$PP_{rel}^{Pure}(A_1, A_2) = \mathbb{P} \times (A_1 \times A_2 \rightarrow \mathbb{P})$$

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

$$(pre_1, post_1) \leq^{PP_{rel}^{Pure}} (pre_2, post_2) \iff pre_2 \Rightarrow pre_1 \wedge \forall(a_1 : A_1)(a_2 : A_2). post_1(a_1, a_2) \Rightarrow post_2(a_1, a_2).$$

The monadic structure is given by

$$\text{let } ret^{PP_{rel}^{Pure}}(a_1, a_2) = (\top, \lambda(a_1', a_2'). a_1 = a_1' \wedge a_2 = a_2')$$

$$\begin{aligned} \text{let } bind^{PP_{rel}^{Pure}}(pre, post) f = \\ \text{let } pre' = pre \wedge \forall a_1, a_2. post(a_1, a_2) \implies \pi_1(f(a_1, a_2)) \text{ in} \\ \text{let } post'(b_1, b_2) = \exists a_1, a_2. post(a_1, a_2) \wedge \pi_2(f(a_1, a_2))(b_1, b_2) \text{ in} \\ (pre', post') \end{aligned}$$

The return operation results in a trivial precondition and a postcondition holding exactly for the given arguments, whereas $bind^{PP_{rel}^{Pure}}$ 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 $\text{PP}_{\text{rel}}^{\text{Pure}}$:

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

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 $\text{PP}_{\text{rel}}^{\text{Pure}}$, threading in the state where necessary, and specifying that the initial state does not change for return:

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

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

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

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

$$\text{W}_{\text{rel}}^{\text{Err}}(A_1, A_2) = ((A_1 \times A_2 + \mathbb{1}) \rightarrow \mathbb{P}) \rightarrow \mathbb{P}. \quad (6.11)$$

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, any simple relational specification monad accounting for exceptions cannot 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 $\text{W}_{\text{rel}}^{\text{Exc}}$ and solve this problem in [section 6.3](#), while previous relational program logics have generally been stuck with weak specification monads in the style of $\text{W}_{\text{rel}}^{\text{Err}}(A_1, A_2)$ above ([Barthe et al., 2016](#)).

6.2.2 Relational semantics from effect observations

The relational judgment $\vdash c_1 \sim c_2 \{ w \}$ should assert that monadic computations $c_1 : M_1 A_1$ and $c_2 : M_2 A_2$ satisfy a relational specification $w : \text{W}_{\text{rel}}(A_1, A_2)$ drawn from a simple relational specification monad. What does satisfaction mean in our monadic framework? Certainly it requires a specific connection between the computational monads M_1, M_2 and the simple relational specification monad W_{rel} . Following the idea of the unary effect observations ([section 2.4](#)), we introduce *relational effect observations*, families of functions respecting the monadic structure, defined here from first principles, but that can be easily seen to be an instance of a relative monad morphism ([section 3.2](#)).

Definition 6.2.2. A simple relational effect observation θ_{rel} from computational monads M_1, M_2 to a simple relational specification monad W_{rel} is given by

- ▷ for each pair of types A_1, A_2 a function $\theta_{\text{rel}} : M_1 A_1 \times M_2 A_2 \rightarrow \text{W}_{\text{rel}}(A_1, A_2)$
- ▷ such that

$$\begin{aligned} & \theta_{\text{rel}}(\text{ret}^{M_1} a_1, \text{ret}^{M_2} a_2) = \text{ret}^{\text{W}_{\text{rel}}} (a_1, a_2) \\ & \theta_{\text{rel}}(\text{bind}^{M_1} m_1 f_1, \text{bind}^{M_2} m_2 f_2) = \text{bind}^{\text{W}_{\text{rel}}} (\theta_{\text{rel}}(m_1, m_2)) (\theta_{\text{rel}} \circ (f_1, f_2)) \end{aligned}$$

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

$$\theta_{\text{rel}}^{\text{St}}(c_1, c_2) = \lambda\varphi (s_1, s_2). \varphi(c_1 s_1, c_2 s_2). \quad (6.12)$$

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

$$\theta_{\text{rel}}^{\forall}(c_1, c_2) = \lambda\varphi. \forall a_1 \in c_1, a_2 \in c_2. \varphi(a_1, a_2), \quad (6.13)$$

$$\theta_{\text{rel}}^{\exists}(c_1, c_2) = \lambda\varphi. \exists a_1 \in c_1, a_2 \in c_2. \varphi(a_1, a_2). \quad (6.14)$$

The first one $\theta_{\text{rel}}^{\forall}$ 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 $\theta_{\text{rel}}^{\exists}$ 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 relational effect observations. To state it, we first recall that two computations $c_1 : M A_1$ and $c_2 : M A_2$ commute (Bowler et al., 2013; Fühmann, 2002) when

$$\text{bind}^M c_1 (\lambda a_1. \text{bind}^M c_2 (\lambda a_2. \text{ret}^M(a_1, a_2))) = \text{bind}^M c_2 (\lambda a_2. \text{bind}^M c_1 (\lambda a_1. \text{ret}^M(a_1, a_2))).$$

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

Theorem 6.2.1. *Let $\theta_1 : M_1 \rightarrow W$ and $\theta_2 : M_2 \rightarrow W$ be unary effect observations, where M_1 and M_2 are computational monads and W is a (unary) specification monad. We denote with $W_{\text{rel}}(A_1, A_2) = W(A_1 \times A_2)$ the simple relational specification monad derived from W (see [subsection 6.2.1](#)). If for all $c_1 : M_1 A_1$ and $c_2 : M_2 A_2$, we have that $\theta_1(c_1)$ and $\theta_2(c_2)$ commute, then the following function $\theta_{\text{rel}} : M_1 A_1 \times M_2 A_2 \rightarrow W_{\text{rel}}(A_1, A_2)$ is a simple relational effect observation*

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

In general, given a simple relational effect observation $\theta_{\text{rel}} : M_1, M_2 \rightarrow W_{\text{rel}}$, we define the semantic relational judgment by

$$\models_{\theta_{\text{rel}}} c_1 \sim c_2 \{ w \} \quad = \quad \theta_{\text{rel}}(c_1, c_2) \leq_{W_{\text{rel}}} w, \quad (6.15)$$

where we make use of the preorder given by W_{rel} . The following 3 subsections explain how to derive rules for a relational logic parameterized by the computational monads M_1, M_2 , the simple relational specification monad W_{rel} , and the simple relational effect observation θ_{rel} .

6.2.3 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 basic pure rules introduced in [Figure 6.1](#) 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.

These rules can then be tailored as explained in the introduction to derive asynchronous (6.1) or synchronous (6.2) 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.

Most of the examples of specification monads we work with actually provide enough structure to strengthen preconditions. An adequate extension of specification monads to provide such strengthening operations and solve this shortcoming also relevant in the unary setting is left as future work.

$$\begin{array}{c}
\mathbb{B}\text{-ELIM} \frac{\text{if } b \text{ then } \vdash c_1 \sim c_2 \{ w^\top \} \quad \text{else } \vdash c_1 \sim c_2 \{ w^\perp \}}{\vdash c_1 \sim c_2 \{ \text{if } b \text{ then } w^\top \text{ else } w^\perp \}} \quad \mathbb{0}\text{-ELIM}^1 \frac{w \leq \perp}{\vdash c_1 \sim c_2 \{ w \}} \\
\\
\mathbb{N}\text{-ELIM} \frac{\begin{array}{c} n : \mathbb{N} \quad w = \text{elim}^{\mathbb{N}} w_0 w_{suc} \quad \vdash c_1[0/n] \sim c_2[0/n] \{ w_0 \} \\ \forall n : \mathbb{N}, \vdash c_1 \sim c_2 \{ w n \} \Rightarrow \vdash c_1[\text{S } n/n] \sim c_2[\text{S } n/n] \{ w_{suc}(w n) \} \end{array}}{\vdash c_1 \sim c_2 \{ w n \}}
\end{array}$$

Figure 6.1: Pure relational rules

$$\begin{array}{c}
\text{RET} \frac{a_1 : A_1 \quad a_2 : A_2}{\vdash \text{ret}^{M_1} a_1 \sim \text{ret}^{M_2} a_2 \{ \text{ret}^W(a_1, a_2) \}} \quad \text{WEAKEN} \frac{\vdash c_1 \sim c_2 \{ w \} \quad w \leq w'}{\vdash c_1 \sim c_2 \{ w' \}} \\
\\
\text{BIND} \frac{\vdash m_1 \sim m_2 \{ w^m \} \quad \forall a_1, a_2 \vdash f_1 a_1 \sim f_2 a_2 \{ w^f(a_1, a_2) \}}{\vdash \text{bind}^{M_1} m_1 f_1 \sim \text{bind}^{M_2} m_2 f_2 \{ \text{bind}^{W_{\text{rel}}} w^m w^f \}}
\end{array}$$

Figure 6.2: Generic monadic rules in the simple framework

6.2.4 Generic monadic rules

Given any computational monads M_1, M_2 and a simple relational specification monad W_{rel} , we introduce three rules governing the monadic part of a relational program logic (Figure 6.2). Each of these rules straightforwardly corresponds to a specific aspect of the simple relational specification monad and are all synchronous. As explained in the introduction (6.5), it is then possible to derive asynchronous variants using the monadic laws of the computational monads.

Theorem 6.2.2 (Soundness of generic monadic rules). *The relational rules in Figure 6.2 are sound with respect to any relational effect observation θ_{rel} , that is $\vdash c_1 \sim c_2 \{ w \} \Rightarrow \forall \theta_{\text{rel}}, \models_{\theta_{\text{rel}}} c_1 \sim c_2 \{ w \}$.*

Proof. For rules RET and BIND, we need to prove that $\theta_{\text{rel}}(\text{ret}^{M_1} a_1, \text{ret}^{M_2} a_2) \leq \text{ret}^W(a_1, a_2)$ and $\theta_{\text{rel}}(\text{bind}^{M_1} m_1 f_1, \text{bind}^{M_2} m_2 f_2) \leq \text{bind}^W(\theta_{\text{rel}}(m_1, m_2))(\theta_{\text{rel}} \circ (f_1, f_2))$, which both hold by the relational effect observation laws and reflexivity. 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. \square

We note that the soundness proof would still be valid if we were to weaken the relational effect observation laws to inequalities. A few examples for such lax relational effect observation appears naturally, for instance in order to deal with variants of relational partial correctness, but we will not consider these in this paper. We further discuss this in the future work section (section 6.5).

6.2.5 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

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

enough to handle full programs written in the computational monads M_1 and M_2 , as we 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 $\theta_{\text{rel}} : M_1, M_2 \rightarrow W_{\text{rel}}$. Consequently, we make essential use of θ_{rel} to introduce 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 [section 6.4](#)) from the problem of choosing sensible specifications, which is captured in the choice of θ_{rel} .

Non-deterministic computations The two relational effect observations $\theta_{\text{rel}}^{\forall}$ and $\theta_{\text{rel}}^{\exists}$ 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 non-deterministic computations that works on the left program, and which is sound with respect to $\theta_{\text{rel}}^{\forall}$. This means that the conclusion will be of the form $\vdash \text{pick} \sim \text{ret } a_2 \{ w \}$ for some $w : \text{PP}_{\text{rel}}^{\text{Pure}}$. To obtain w , we apply the effect observation to the computations involved in the rule

$$w = \theta_{\text{rel}}^{\forall}(\text{pick}, \text{ret } a_2) = \lambda\varphi. \forall b \in \{\text{true}, \text{false}\}, a \in \{a_2\}. \varphi(b, a),$$

obtaining thus a rule which is trivially sound:

$$\text{DEMONICLEFT} \frac{}{\vdash \text{pick} \sim \text{ret } a_2 \{ \lambda\varphi. \varphi(\text{true}, a_2) \wedge \varphi(\text{false}, a_2) \}}.$$

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 $\theta_{\text{rel}}^{\exists}$:

$$\text{ANGELIC} \frac{}{\vdash \text{pick} \sim \text{pick} \left\{ \lambda\varphi. \begin{array}{l} \varphi(\text{true}, \text{true}) \vee \varphi(\text{true}, \text{false}) \vee \\ \varphi(\text{false}, \text{true}) \vee \varphi(\text{false}, \text{false}) \end{array} \right\}}.$$

Exceptions using $W_{\text{rel}}^{\text{Err}}$ Taking M_1 and M_2 to be exception monads on exception sets E_1 and E_2 , and the relational specification monad $W_{\text{rel}}^{\text{Err}}$ ([Equation 6.11](#) on page 97), we have an effect observation interpreting any thrown exception as a unique erroneous termination situation, that is

`let $\theta \text{ Err } ((c_1, c_2) : \text{Exc } A_1 \times \text{Exc } A_2) : W_{\text{rel}}^{\text{Err}}(A_1, A_2) =$`
 `$\lambda\varphi. \text{match } c_1, c_2 \text{ with } | \text{Inl } a_1, \text{Inl } a_2 \rightarrow \varphi(\text{Inl } (a_1, a_2)) | _, _ \rightarrow \varphi(\text{Inr } ())$`

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

$$\begin{array}{c} \text{THROWL} \frac{}{\vdash \text{throw } e_1 \sim \text{ret } a_2 \{ \lambda\varphi. \varphi(\text{inr } ()) \}} \\ \text{THROWR} \frac{}{\vdash \text{ret } a_1 \sim \text{throw } e_2 \{ \lambda\varphi. \varphi(\text{inr } ()) \}} \\ \text{CATCH} \frac{\begin{array}{c} \vdash c_1 \sim c_2 \{ w \} \quad \forall e_1 e_2 \vdash c_1^{\star} e_1 \sim c_2^{\star} e_2 \{ w^{\star} \} \\ \forall e_1 a_2 \vdash c_1^{\star} e_1 \sim \text{ret } a_2 \{ w^{\star} \} \quad \forall a_1 e_2 \vdash \text{ret } a_1 \sim c_2^{\star} e_2 \{ w^{\star} \} \end{array}}{\vdash \text{catch } c_1 c_1^{\star} \sim \text{catch } c_2 c_2^{\star} \{ \lambda\varphi. w(\lambda a_0. \text{match } a_0 \text{ with } \text{Inl } a \rightarrow \varphi a | \text{Inr } () \rightarrow w^{\star} \varphi) \}} \end{array}$$

The rules `THROWL` and `THROWR` can be derived using the recipe above, but the exceptions have to be conflated to the same exceptional result `inr ()`, a situation that is forced by the choice of relational effect observation and a weak specification monad. As a consequence, the `CATCH` rule has to consider three exceptional cases. The specification for `CATCH` does not follow mechanically from $\theta_{\text{rel}}^{\text{Err}}$ using our recipe since it is a handler and not an algebraic operation.

6.3 Generic Framework

While the simple framework works well for a variety of effects, it falls short of providing a convincing treatment of effects with control such as exceptions or non-termination. 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 notions 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.

6.3.1 Exceptional control flow in relational reasoning

We explained in [subsection 6.2.5](#) 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 post-condition on pairs of either a value or an exceptional final state to a proposition:

$$W_{\text{rel}}^{\text{Exc}}(A_1, A_2) = ((A_1 + E_1) \times (A_2 + E_2) \rightarrow \mathbb{P}) \rightarrow \mathbb{P}.$$

For instance, the specification above can be stated as $\lambda\varphi. \forall ae_1 ae_2. (\text{Inr}?ae_1 \Rightarrow \text{Inr}?ae_2) \Rightarrow \varphi(ae_1, ae_2) : W_{\text{rel}}^{\text{Exc}}(A_1, A_2)$, where $\text{Inr}?ae = \text{match } ae \text{ with } \text{Inl } _ \rightarrow \top \mid _ \rightarrow \perp$.

As explained in [section 6.1](#), this type does not admit a monadic operation $\text{bind } w^m w^f$ using only a continuation of type $w^f : A_1 \times A_2 \rightarrow W_{\text{rel}}^{\text{Exc}}(B_1, B_2)$ 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 $\text{bind}_{\text{rel}}^{W_{\text{rel}}^{\text{Exc}}}$ the missing information it needs in such cases. To that purpose, we use the unary specification monads $W_1^{\text{Exc}} A_1 = (A_1 + E_1 \rightarrow \mathbb{P}) \rightarrow \mathbb{P}$ and $W_2^{\text{Exc}} A_2 = (A_2 + E_2 \rightarrow \mathbb{P}) \rightarrow \mathbb{P}$ to provide independent specifications of each program. With the addition of these, we can write a function that relies on the unary specifications when the results of the first computations differ (one raise an exception and the other returns).

```

val bindWrelExc : WrelExc(A1, A2) → (A1 → W1Exc B1) → (A2 → W2Exc B2) →
    (A1 × A2 → WrelExc(B1, B2)) → WrelExc(B1, B2)
let bindWrelExc wm (f1 : A1 → ((B1 + E1) → ℙ) → ℙ) (f2 : A2 → ((B2 + E2) → ℙ) → ℙ) f =
  λ(φ : (B1 + E1) → ℙ).
    wm (λ ae : (A1 + E1) × (A2 + E2).
      match ae with
      | Inl a1, Inl a2 → f a1 a2 φ
      | Inl a1, Inr e2 → f1 a1 (λ be → φ be (Inr e2))
      | Inr e1, Inl a2 → φ (Inr e1, Inl a2)
      | Inr e1, Inr e2 → φ (Inr e1, Inr e2)

```

6.3.2 A problem of context

In order to keep track of these unary specifications drawn from W_1^{Exc} and W_2^{Exc} in the relational proofs, we extend the relational judgment to

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

Here, $w_1 : W_1^{\text{Exc}} A_1$ is a unary specification for $c_1 : \text{Exc}_1 A_1$, symmetrically $w_2 : W_2^{\text{Exc}} A_2$ is a unary specification for $c_2 : \text{Exc}_2 A_2$, and $w_{\text{rel}} : W_{\text{rel}}^{\text{Exc}}(A_1, A_2)$ 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 w stands for the triple $(w_1, w_2, w_{\text{rel}})$:

$$\frac{\vdash m_1 \{w_1^m\} \sim m_2 \{w_2^m\} \mid w_{\text{rel}}^m \quad \forall a_1, a_2 \vdash f_1 a_1 \{w_1^f a_1\} \sim f_2 a_2 \{w_2^f a_2\} \mid w_{\text{rel}}^f a_1 a_2}{\vdash \text{bind}^{\text{Exc}_1} m_1 f_1 \{\text{bind}^{W_1^{\text{Exc}}} w_1^m w_1^f\} \sim \text{bind}^{\text{Exc}_2} m_2 f_2 \{\text{bind}^{W_2^{\text{Exc}}} w_2^m w_2^f\} \mid \text{bind}^{W_{\text{rel}}^{\text{Exc}}} w^m w^f}$$

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:

$$\models c_1 \{w_1\} \sim c_2 \{w_2\} \mid w_{\text{rel}} = \theta_1^{\text{Exc}} c_1 \leq w_1 \wedge \theta_2^{\text{Exc}} c_2 \leq w_2 \wedge \theta_{\text{rel}}^{\text{Exc}}(c_1, c_2) \leq w_{\text{rel}}$$

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 $\theta_1^{\text{Exc}}(\text{bind}^{\text{Exc}_1} m_1 f_1) \leq \text{bind}^{W_1^{\text{Exc}}} w_1^m w_1^f$ under the hypothesis $\theta_1^{\text{Exc}} m_1 \leq w_1^m \wedge \dots$ and $\forall a_1, a_2, \theta^{W_1^{\text{Exc}}}(f_1 a_1) \leq w_1^f a_1 \wedge \dots$, in particular the second hypothesis requires an element $a_2 : A_2$ that prevents² us from concluding by monotonicity of $\text{bind}^{W_1^{\text{Exc}}}$.

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 $\Gamma \vdash c_1 \{w_1\} \sim c_2 \{w_2\} \mid w_{\text{rel}}$ now presupposes that $\Gamma_i \vdash c_i : M_i A_i$, $\Gamma_i \vdash w_i : W_i$ ($i = 1, 2$) and that $\Gamma_1, \Gamma_2 \vdash w_{\text{rel}} : W_{\text{rel}}(A_1, A_2)$. The semantics of this judgment is given by

$$\Gamma \models c_1 \{w_1\} \sim c_2 \{w_2\} \mid w_{\text{rel}} = \left(\begin{array}{c} \forall \gamma_1 : \Gamma_1, \theta_1(c_1 \gamma_1) \leq w_1 \gamma_1, \\ \forall \gamma_2 : \Gamma_2, \theta_2(c_2 \gamma_2) \leq w_2 \gamma_2, \\ \forall (\gamma_1, \gamma_2) : \Gamma_1 \times \Gamma_2, \theta_{\text{rel}}(c_1 \gamma_1, c_2 \gamma_2) \leq w_{\text{rel}}(\gamma_1, \gamma_2) \end{array} \right) \quad (6.16)$$

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

$$\begin{aligned} \Theta_1(\gamma_1 : \Gamma_1) &= \theta_1(c_1 \gamma_1) \leq w_1 \gamma_1, & \Theta_2(\gamma_2 : \Gamma_2) &= \theta_2(c_2 \gamma_2) \leq w_2 \gamma_2, \\ \Theta_{\text{rel}}(\gamma : \Gamma)(w_1 : \Theta_1 \gamma_1, w_2 : \Theta_2 \gamma_2) &= \theta_{\text{rel}}(c_1 \gamma_1, c_2 \gamma_2) \leq w_{\text{rel}} \gamma. \end{aligned}$$

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 $(\gamma : \Gamma^r) \rightarrow \Theta^r \gamma$ in an appropriate relational dependent type theory.

6.3.3 A relational dependent type theory

Adding unary specifications in the relational judgment enables a full treatment of exceptions, however the pure rules of section [subsection 6.2.3](#) do not deal with a structured context $\Gamma^r = (\Gamma_1, \Gamma_2, \Gamma_{\text{rel}})$. 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 is 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 6.3](#),

²Instead of insisting that $\vdash c_1 \{w_1\} \sim c_2 \{w_2\} \mid w_{\text{rel}}$ 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 $\theta_1^{\text{Exc}}(\text{bind}^{\text{Exc}_1} m_1 f_1) \leq \text{bind}^{W_1^{\text{Exc}}} w_1^m w_1^f \Rightarrow \theta_1^{\text{Exc}} m_1 \leq w_1^m$ that has no reason to hold in our examples.

$$\begin{aligned}
A^r, B^r, \Gamma^r &::= \mathbb{0}^r \mid \mathbb{1}^r \mid \mathbb{B}^r \mid \mathbb{N}^r \mid A^r + B^r \mid (a : A^r) \times B^r \mid (a : A^r) \rightarrow B^r \mid a \\
\llbracket - \rrbracket &\text{ maps a relational type } A^r \text{ to its underlying representation } \llbracket A^r \rrbracket = (A_0, A_1, A_r) \\
\llbracket \mathbb{0}^r \rrbracket &= (\mathbb{0}, \mathbb{0}, =) \quad \llbracket \mathbb{1}^r \rrbracket = (\mathbb{1}, \mathbb{1}, =) \quad \llbracket \mathbb{B}^r \rrbracket = (\mathbb{B}, \mathbb{B}, =) \quad \llbracket \mathbb{N}^r \rrbracket = (\mathbb{N}, \mathbb{N}, =) \\
\llbracket A^r + B^r \rrbracket &= \left(\begin{array}{l} ab_1 : A_1 + B_1 \\ ab_2 : A_2 + B_2 \\ \text{case } (ab_1, ab_2) \left[\begin{array}{l} (\text{inl } a_1, \text{inl } a_2). A_{\text{rel}} a_1 a_2 \\ (\text{inr } b_1, \text{inr } b_2). B_{\text{rel}} b_1 b_2 \\ (\quad , \quad). \quad \quad \quad \mathbb{0} \end{array} \right] \end{array} \right) \\
\llbracket (a : A^r) \times B^r \mid a \rrbracket &= \left(\begin{array}{l} (a_1, b_1) : (a_1 : A_1) \times B_1 a_1, \\ (a_2, b_2) : (a_2 : A_2) \times B_2 a_2, \\ (a_r : A_r a_1 a_2) \times B_r a_1 a_2 a_r b_1 b_2 \end{array} \right) \\
\llbracket (a : A^r) \rightarrow B^r \mid a \rrbracket &= \left(\begin{array}{l} f_1 : (a_1 : A_1) \rightarrow B_1 a_1, \\ f_2 : (a_2 : A_2) \rightarrow B_2 a_2, \\ (a_1 : A_1)(a_2 : A_2)(a_r : A_r a_1 a_2) \rightarrow B_r a_1 a_2 a_r (f_1 a_1) (f_2 a_2) \end{array} \right)
\end{aligned}$$

Figure 6.3: Syntax of RDTT and translation to base type theory

$$\begin{aligned}
\llbracket \text{elim_sum} \rrbracket &: (P_1 : A_1 + B_1 \rightarrow \text{Type}) \rightarrow (P_2 : A_2 + B_2 \rightarrow \text{Type}) \rightarrow \\
&(P_{\text{rel}} : \forall(ab_1 : A_1 + B_1)(ab_2 : A_2 + B_2), (A^r + B^r)_{\text{rel}} ab_1 ab_2 \rightarrow \text{Type}) \rightarrow \\
&(\forall(a_1 : A_1), P_1(\text{inl } a_1)) \rightarrow (\forall(a_2 : A_2), P_2(\text{inl } a_2)) \rightarrow \\
&(\forall a_1 a_2 (a_{\text{rel}} : A^r a_1 a_2), P_{\text{rel}}(\text{inl } a_1)(\text{inl } a_2) a_{\text{rel}}) \rightarrow \\
&(\forall(b_1 : B_1), P_1(\text{inr } b_1)) \rightarrow (\forall(b_2 : B_2), P_2(\text{inr } b_2)) \rightarrow \\
&(\forall b_1 b_2 (b_{\text{rel}} : B^r b_1 b_2), P_{\text{rel}}(\text{inr } b_1)(\text{inr } b_2) b_{\text{rel}}) \rightarrow \\
&\forall ab_1 ab_2 (ab_{\text{rel}} : (A^r + B^r)_{\text{rel}} ab_1 ab_2), P_{\text{rel}} ab_1 ab_2 ab_{\text{rel}}
\end{aligned}$$

Figure 6.4: Translation of the eliminator for sums in RDTT

providing the definition of types and omitting the terms and typing judgements. A systematic construction of RDTT at the semantic level is obtained by considering 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 $P(ab : A^r + B^r) = \Theta^r(ab) : \text{Type}^r$ on the dependent eliminator for sum type

$$\text{elim_sum} : (P : (A^r + B^r) \rightarrow \text{Type}^r) \rightarrow (a : A^r \rightarrow P a) \rightarrow (b : B^r \rightarrow P b) \rightarrow (x : A^r + B^r) \rightarrow P x$$

to obtain a rule for case splitting. This eliminator translates to a large term described in [Figure 6.4](#) that induces the following relational rule using $\mathbf{w}^l = (w_1^l, w_2^l, w_{\text{rel}}^l)$, $\mathbf{w}^r = (w_1^r, w_2^r, w_{\text{rel}}^r)$ and the relational specifications of the conclusion – where we abbreviate pattern matching with a case

construction – as arguments to the eliminator

$$\frac{\begin{array}{l} \Gamma, \mathbf{a} : A^r \vdash c_1[\text{inl } a_1/ab_1] \{w_1^l\} \sim c_2[\text{inl } a_2/ab_2] \{w_2^l\} \mid w_{\text{rel}}^l[a_{\text{rel}}/ab_{\text{rel}}] \\ \Gamma, \mathbf{b} : B^r \vdash c_1[\text{inr } b_1/ab_1] \{w_1^r\} \sim c_2[\text{inr } b_2/ab_2] \{w_2^r\} \mid w_{\text{rel}}^r[b_{\text{rel}}/ab_{\text{rel}}] \end{array}}{\Gamma^r, \mathbf{ab} : A^r + B^r \vdash \begin{array}{l} c_1 \quad \{\text{case } ab_1 [\text{inl } a_1.w_1^l \mid \text{inr } b_1.w_1^r]\} \\ \sim \\ c_2 \quad \{\text{case } ab_2 [\text{inl } a_2.w_2^l \mid \text{inr } b_2.w_2^r]\} \end{array} \mid \text{case } ab_1, ab_2 \left[\begin{array}{l} \text{inl } a_1, \text{inl } a_2.w_{\text{rel}}^l \\ \text{inr } b_1, \text{inr } b_2.w_{\text{rel}}^r \end{array} \right]}$$

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

6.3.4 Relational specification monads

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 from [chapter 3](#) to our relational dependent type theory, ensuring that we obtain a theory uniform with the simple setting, and crucially that we can use the same methodology to introduce relational rules.

What we ought to call a relational specification monad should assign to any pair of types (A_1, A_2) , a triple of ordered types $(W_1 A_1, W_2 A_2, W_{\text{rel}}(A_1, A_2))$ corresponding respectively to the type of unary specifications for the left and right programs, together with the type of relational specifications.

This description would invite us to consider relational specification monad as Pos -enriched functors $\underline{Set}^2 \rightarrow \underline{Pos}^3$ with a relative monad structure with respect to the base functor

$$\mathcal{J} : \left\{ \begin{array}{ll} \underline{Set}^2 & \longrightarrow \\ (A_1, A_2) & \longmapsto (\text{Disc } A_1, \text{Disc } A_2, \text{Disc } A_1 \times \text{Disc } A_2) \end{array} \right. \underline{Pos}^3$$

There is however a small discrepancy: if W is such a \mathcal{J} -relative monad, its value at a pair of types (A_1, A_2) is a triple $(W_1(A_1, A_2), W_2(A_1, A_2), W_{\text{rel}}(A_1, A_2))$ where the first and second component can respectively depend on A_2 and A_1 , a feature that we do not expect from a relational specification monad.

A first way to solve this discrepancy would be by enforcing that the first and second projections come from unary specification monads. If W_1, W_2 are two (unary) specification monads, we can pair these together to define a monad relative to $\text{Disc} \times \text{Disc} : \underline{Set}^2 \rightarrow \underline{Pos}^2$

$$W_1 \times W_2 : \underline{Set}^2 \longrightarrow \underline{Pos}^2$$

Since we have a commuting triangle of (Pos -enriched) functors

$$\begin{array}{ccc} \underline{Set}^2 & \xrightarrow{\mathcal{J}} & \underline{Pos}^3 \\ & \searrow \text{Disc} \times \text{Disc} & \downarrow \pi_{12} \\ & & \underline{Pos}^2 \end{array}$$

we could define a relational specification monad W to be \mathcal{J} -relative monad lifting $W_1 \times W_2$ along the functor $\pi_{12} : \underline{Pos}^3 \rightarrow \underline{Pos}^2$.

We choose to use a second, slightly more convoluted solution, that has the benefit of making clearer the connection with the relational dependent type theory of the previous section. Moreover, it provides a definition that does not involve any lifting condition, presumably simpler to implement inside an intensional type theory such as Coq.

Recall the following categorical presentation of relations. If \mathcal{C} is a category, the category $\text{Span}(\mathcal{C})$ consists of spans in \mathcal{C} , that is diagrams $C_1 \leftarrow C_{\text{rel}} \rightarrow C_2$ in \mathcal{C} , and morphisms of spans, that

is morphisms $(f_1, f_2, f_{\text{rel}}) : (C_1 \leftarrow C_{\text{rel}} \rightarrow C_2) \rightarrow (D_1 \leftarrow D_{\text{rel}} \rightarrow D_2)$ in \mathcal{C} such that the following diagram commute

$$\begin{array}{ccccc} C_1 & \longleftarrow & C_{\text{rel}} & \longrightarrow & C_2 \\ f_1 \downarrow & & \downarrow f_{\text{rel}} & & \downarrow f_2 \\ D_1 & \longleftarrow & D_{\text{rel}} & \longrightarrow & D_2 \end{array}$$

We are concerned with the case $\mathcal{C} = \text{Pos}$. Since $\text{Span}(\text{Pos})$ is cartesian closed, with internal hom between $\mathbf{A} = A_1 \leftarrow A_{\text{rel}} \rightarrow A_2$ and $\mathbf{B} = B_1 \leftarrow B_{\text{rel}} \rightarrow B_2$ given by the span

$$\mathbf{A} \Rightarrow \mathbf{B} = \text{Pos}(A_1, B_1) \leftarrow (\mathbf{A} \Rightarrow \mathbf{B})_{\text{rel}} \rightarrow \text{Pos}(A_2, B_2)$$

$$\begin{aligned} (\mathbf{A} \Rightarrow \mathbf{B})_{\text{rel}} &= (f_1 : \text{Pos}(A_1, B_1)) \times (f_2 : \text{Pos}(A_2, B_2)) \times (f_{\text{rel}} : \text{Pos}(A_{\text{rel}}, B_{\text{rel}})) \\ &\times \{\forall(a_{\text{rel}} : A_{\text{rel}}), p_1(f_{\text{rel}} a_{\text{rel}}) = f_1(p_1 a_{\text{rel}}) \wedge p_2(f_{\text{rel}} a_{\text{rel}}) = f_2(p_2 a_{\text{rel}})\} \end{aligned}$$

we can enrich it over itself, yielding the enrich category $\underline{\text{Span}}(\text{Pos})$. Similarly to the discussion above, we define an enriched base functor $\underline{\mathcal{J}}_{\times}$ whose definition on a pair of types (A_1, A_2) is given by the span

$$\underline{\mathcal{J}}_{\times}(A_1, A_2) = \text{Disc } A_1 \xleftarrow{\pi_1} \text{Disc } A_1 \times \text{Disc } A_2 \xrightarrow{\pi_2} \text{Disc } A_2.$$

Now, a $\underline{\mathcal{J}}_{\times}$ -relative monad W is almost what we need to interpret relational specifications: it consists of a mapping from pairs of types (A_1, A_2) to a spans

$$W_1(A_1, A_2) \leftarrow W_{\text{rel}}(A_1, A_2) \rightarrow W_2(A_1, A_2),$$

together with return and bind operations satisfying monotonicity conditions. We tame the potential dependency of W_1 in A_2 (respectively W_2 in A_1) thanks to the following theorem.

Theorem 6.3.1. *The mapping $\widetilde{(\cdot)}$ from $\underline{\mathcal{J}}_{\times}$ -relative monad to $\underline{\mathcal{J}}_{\times}$ -relative monad sending*

$$W(A_1, A_2) = W_1(A_1, A_2) \leftarrow W_{\text{rel}}(A_1, A_2) \rightarrow W_2(A_1, A_2)$$

to

$$\widetilde{W}(A_1, A_2) = W_1(A_1, \mathbb{1}) \leftarrow W_{\text{rel}}(A_1, A_2) \rightarrow W_2(\mathbb{1}, A_2)$$

extends to an idempotent monad on the category of $\underline{\mathcal{J}}_{\times}$ -relative monads.

In particular any $\underline{\mathcal{J}}_{\times}$ -relative monad can be canonically completed so that W_1 and W_2 respectively depend only on A_1 or A_2 when applied to the pair (A_1, A_2) .

Proof. Let W be $\underline{\mathcal{J}}_{\times}$ -relative monad and $W_1 \xleftarrow{p_1} W_{\text{rel}} \xrightarrow{p_2} W_2$ its components. The main idea of the proof is that $W_1(A_1, A_2)$ cannot depend essentially on A_2 because of the constraints of the $\underline{\mathcal{J}}_{\times}$ -relative monad structure. This is made more formal by the following observation: let A_1, X_2, Y_2 be sets, then the map

$$\varphi_{A_1, X_2, Y_2}^1 = \text{bind}_1^W{}_{A_1, X_2, A_1, Y_2}(\text{ret}_1^W{}_{A_1, Y_2}) \in \text{Pos}(W_1(A_1, X_2), W_1(A_1, Y_2)),$$

where we wrote explicitly as subscript the sets at which we instantiate the monadic operations of W , is an isomorphism. Its inverse is $\varphi_{A_1, Y_2, X_2}^1$ as shown by the following elementary computation

$$\begin{aligned} \varphi_{A_1, X_2, Y_2}^1 \circ \varphi_{A_1, Y_2, X_2}^1 &= \text{bind}_1^W{}_{A_1, X_2, A_1, Y_2}(\text{ret}_1^W{}_{A_1, Y_2}) \circ \text{bind}_1^W{}_{A_1, Y_2, A_1, X_2}(\text{ret}_1^W{}_{A_1, X_2}) \\ &= \text{bind}_1^W{}_{A_1, Y_2, A_1, Y_2}(\text{bind}_1^W{}_{A_1, X_2, A_1, Y_2}(\text{ret}_1^W{}_{A_1, Y_2}) \circ (\text{ret}_1^W{}_{A_1, X_2})) \\ &= \text{bind}_1^W{}_{A_1, Y_2, A_1, Y_2}(\text{ret}_1^W{}_{A_1, Y_2}) \\ &= \text{id}_{W_1(A_1, Y_2)}. \end{aligned}$$

We note $\varphi_{A_2, X_1, Y_1}^2 \in \text{Pos}(W_2(X_1, A_2), W_2(Y_1, A_2))$ the corresponding isomorphism for W_2 .

With this observation in hand, we define a \mathcal{J}_\times -relative monad structure on

$$\widetilde{W}(A_1, A_2) = W_1(A_1, \mathbb{1}) \xleftarrow{\tilde{p}_1} W_{\text{rel}}(A_1, A_2) \xrightarrow{\tilde{p}_2} W_2(\mathbb{1}, A_2)$$

where $\tilde{p}_1 = \varphi_{A_1, A_2, \mathbb{1}}^1 \circ p_1$ and $\tilde{p}_2 = \varphi_{A_2, A_1, \mathbb{1}}^2 \circ p_2$. The return operation of \widetilde{W} is simply the adequate restriction of W where the triangles on both sides commute by naturality of ret^W

$$\begin{array}{ccccccc} & & A_1 & \xleftarrow{\pi_1} & A_1 \times A_2 & \xrightarrow{\pi_2} & A_2 \\ & \swarrow \text{ret}^{\widetilde{W}_1} & \downarrow \text{ret}^{W_1} & & \downarrow \text{ret}^{\widetilde{W}_{\text{rel}}} & & \downarrow \text{ret}^{W_2} \\ & \swarrow & & & & & \searrow \text{ret}^{\widetilde{W}_2} \\ W_1(A_1, \mathbb{1}) & \xleftarrow{\widetilde{W}_1(A_1, \mathbb{1})} & W_1(A_1, A_2) & \xleftarrow{p_1} & W_{\text{rel}}(A_1, A_2) & \xrightarrow{p_2} & W_2(A_1, A_2) \xrightarrow{\widetilde{W}_2(\mathbb{1}, A_2)} W_2(\mathbb{1}, A_2) \end{array}$$

The definition of bind is slightly more involved. We need to define a morphism of span $\text{bind}^{\widetilde{W}} = (\text{bind}_1^{\widetilde{W}}, \text{bind}_2^{\widetilde{W}}, \text{bind}_{\text{rel}}^{\widetilde{W}})$ for sets A_1, A_2, B_1, B_2

$$\text{bind}^{\widetilde{W}} : \underline{\text{Span}}(\text{Pos})(\mathcal{J}_\times(A_1, A_2), \widetilde{W}(B_1, B_2)) \rightarrow \underline{\text{Span}}(\text{Pos})(\widetilde{W}(A_1, A_2), \widetilde{W}(B_1, B_2)).$$

The components $\text{bind}_1^{\widetilde{W}}$ and $\text{bind}_2^{\widetilde{W}}$ are provided respectively by the adequate instantiations of

$$\begin{aligned} \text{bind}_1^W & : \text{Pos}(A_1, W_1(B_1, \mathbb{1})) \rightarrow \text{Pos}(W_1(A_1, \mathbb{1}), W_1(B_1, \mathbb{1})) \quad \text{and} \\ \text{bind}_2^W & : \text{Pos}(A_2, W_2(\mathbb{1}, B_2)) \rightarrow \text{Pos}(W_2(\mathbb{1}, A_2), W_2(\mathbb{1}, B_2)) \end{aligned}$$

In order to define the component $\text{bind}_{\text{rel}}^{\widetilde{W}}$ on $\mathbf{f} = (f_1, f_2, f_{\text{rel}}) \in \underline{\text{Span}}(\text{Pos})(\mathcal{J}_\times(A_1, A_2), \widetilde{W}(B_1, B_2))$, that is morphisms

$$f_1 \in \text{Pos}(A_1, W_1(B_1, \mathbb{1})), \quad f_{\text{rel}} \in \text{Pos}(A_1 \times A_2, W_{\text{rel}}(B_1, B_2)), \quad f_2 \in \text{Pos}(A_2, W_2(\mathbb{1}, B_2)),$$

we complete where needed with $\varphi^{(\cdot)}$ and define $\text{bind}_{\text{rel}}^{\widetilde{W}} \mathbf{f} = \text{bind}_{\text{rel}}^W (\varphi_{A_1, \mathbb{1}, B_2}^1 \circ f_1, \varphi_{A_2, \mathbb{1}, B_1}^2 \circ f_2, f_{\text{rel}})$. The following diagram shows that this indeed define a morphism of spans as required (we only show it for the first projection, the second projection being symmetric).

$$\begin{array}{ccccc} A_1 & & & & A_1 \times A_2 \\ & \searrow \text{bind}_1^W f_1 & & \searrow \text{bind}_1^W (\varphi_{A_1, \mathbb{1}, A_2}^1 \circ f_1) & \\ & & W_1(A_1, \mathbb{1}) & & \\ & \searrow \varphi_{A_1, \mathbb{1}, A_2}^1 & & \searrow \varphi_{A_1, A_2, \mathbb{1}}^1 & \\ W_1(A_1, \mathbb{1}) & \xleftarrow{\varphi_{A_1, A_2, \mathbb{1}}^1} & W_1(A_1, A_2) & \xleftarrow{p_1} & W_{\text{rel}}(A_1, A_2) \end{array}$$

The monadic laws for \widetilde{W} are straightforward consequences of the same laws for W . The functorial action of (\cdot) just restricts a \mathcal{J}_\times -relative monad morphism to the adequate components. The return operation of (\cdot) as a monad is the span morphism

$$(\varphi_{A_1, A_2, \mathbb{1}}^1, \varphi_{A_2, A_1, \mathbb{1}}^2, \text{id}_{W_{\text{rel}}(A_1, A_2)}) : W(A_1, A_2) \longrightarrow \widetilde{W}(A_1, A_2)$$

while multiplication is the identity, obviously making the monad idempotent. \square

This discussion lead us to the following elementary definition of a relational specification monad.

Definition 6.3.1. A relational specification monad consist of

- ▷ for each pair of types (A_1, A_2) , types $W_1 A_1, W_2 A_2$ and a relation $W_{rel}(A_1, A_2) : W_1 A_1 \rightarrow W_2 A_2 \rightarrow \text{Type}$ between them, each equipped with a preorder \leq^W ;
- ▷ operations

$$\text{ret}^{W_1} : A_1 \rightarrow W_1 A_1 \qquad \text{ret}^{W_2} : A_2 \rightarrow W_2 A_2$$

$$\text{ret}^{W_{rel}} : (a_1, a_2) : A_1 \times A_2 \rightarrow W_{rel}(A_1, A_2) (\text{ret}^{W_1} a_1) (\text{ret}^{W_2} a_2)$$

- ▷ operations

$$\text{bind}^{W_1} : W_1 A_1 \rightarrow (A_1 \rightarrow W_1 B_1) \rightarrow W_1 B_1$$

$$\text{bind}^{W_2} : W_2 A_2 \rightarrow (A_2 \rightarrow W_2 B_2) \rightarrow W_2 B_2$$

$$\begin{aligned} \text{bind}^{W_{rel}} : w_1^m : W_1 A_1 \rightarrow w_2^m : W_2 A_2 \rightarrow w_{rel}^m : W_{rel}(A_1, A_2) w_1^m w_2^m \rightarrow \\ w_1^f : (A_1 \rightarrow W_1 B_1) \rightarrow w_2^f : (A_2 \rightarrow W_2 B_1) \rightarrow \\ w_{rel}^f : (((a_1, a_2) : A_1 \times A_2) \rightarrow W_{rel}(B_1, B_2) (w_1^f a_1) (w_2^f a_2)) \rightarrow \\ W_{rel}(B_1, B_2) (\text{bind}^{W_1} w_1^m w_1^f) (\text{bind}^{W_2} w_2^m w_2^f) \end{aligned}$$

monotonic in all arguments

- ▷ satisfying equations analogous to the monadic laws
- ▷ as well as monotonic operations $\tau_1 : w_1 : W_1 A_1 \rightarrow W_{rel}(A_1, \mathbb{1}) w_1 (\text{ret}^{W_2} ())$ and $\tau_2 : w_2 : W_2 A_2 \rightarrow W_{rel}(\mathbb{1}, A_2) (\text{ret}^{W_1} ()) w_2$ satisfying certain compatibility with the monadic operations detailed in the discussion below.

If the presence of the operations τ_1 and τ_2 can seem surprising, they ensure that we can construct exception transformers (see [subsection 6.3.6](#)). In order to explain what these operations are, first note that from a \mathcal{J}_\times -relative monad $W = (W_1 \leftarrow W_{rel} \rightarrow W_2)$, we can derive four unary specification monads – two for each legs of the span – by combining restrictions of the domain and projections:

$$\begin{aligned} W_1^1 A &= W_1(A, \mathbb{1}) & W_1^\Sigma A &= (w : W_1(A, \mathbb{1})) \times W_{rel}(A, \mathbb{1}) w (\text{ret}^{W_2} ()) \\ W_2^1 A &= W_2(\mathbb{1}, A) & W_2^\Sigma A &= (w : W_2(\mathbb{1}, A)) \times W_{rel}(\mathbb{1}, A) (\text{ret}^{W_1} ()) w \end{aligned}$$

There are obvious projections $\pi^1 : W_1^\Sigma \rightarrow W_1^1$ and $\pi^2 : W_2^\Sigma \rightarrow W_2^1$ that preserves the monadic structure. We require τ_1 (resp. τ_2) to be induced by a section of π_1 , in particular it needs to be a monad morphism.

In most of our examples the relation part of the monad is actually constant, simplifying further the type of operations to:

$$\text{ret}^{W_{rel}} : A_1 \times A_2 \rightarrow W_{rel}(A_1, A_2)$$

$$\begin{aligned} \text{bind}^{W_{rel}} : W_1 A_1 \rightarrow W_2 A_2 \rightarrow W_{rel}(A_1, A_2) \rightarrow \\ (A_1 \rightarrow W_1 B_1) \rightarrow (A_2 \rightarrow W_2 B_1) \rightarrow (A_1 \times A_2 \rightarrow W_{rel}(B_1, B_2)) \rightarrow W_{rel}(B_1, B_2) \end{aligned}$$

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 W_{rel} the relational specification monad $W(A_1, A_2) = (W_{rel}(A_1, \mathbb{1}), W_{rel}(\mathbb{1}, A_2), \lambda w_1 w_2. W_{rel}(A_1, A_2))$. The monadic operations just discard the superfluous arguments and τ_1, τ_2 are just identities.

$$\begin{array}{c}
\text{WEAKEN} \frac{\Gamma^r \vdash c_1 \{w_1\} \sim c_2 \{w_2\} \mid w_{\text{rel}} \quad w_1 \leq^{W_1} w'_1 \quad w_2 \leq^{W_2} w'_2 \quad w_{\text{rel}} \leq^{W_{\text{rel}}} w'_{\text{rel}}}{\Gamma^r \vdash c_1 \{w'_1\} \sim c_2 \{w'_2\} \mid w'_{\text{rel}}} \\
\\
\text{RET} \frac{\Gamma_1 \vdash a_1 : A_1 \quad \Gamma_2 \vdash a_2 : A_2}{\Gamma^r \vdash \text{ret}^{M_1} a_1 \{ \text{ret}^{W_1} a_1 \} \sim \text{ret}^{M_2} a_2 \{ \text{ret}^{W_2} a_2 \} \mid \text{ret}^{W_{\text{rel}}} (a_1, a_2)} \\
\\
\text{BIND} \frac{\Gamma^r \vdash m_1 \{w_1^m\} \sim m_2 \{w_2^m\} \mid w^m \quad \Gamma^r, a : A^r \vdash f_1 a_1 \{w_1^f a_1\} \sim f_2 a_2 \{w_2^m a_2\} \mid w^f a}{\Gamma^r \vdash \begin{array}{c} \text{bind}^{M_1} m_1 f_1 \{ \text{bind}^{W_1} w_1^m w_1^f \} \\ \text{bind}^{M_2} m_2 f_2 \{ \text{bind}^{W_2} w_2^m w_2^f \} \end{array} \mid \text{bind}^{W_{\text{rel}}} w^m w^f}
\end{array}$$

Figure 6.5: Generic monadic rules in the full relational setting

6.3.5 Relational effect observations

The adequate notion of morphism between relational specification monad is given by relative monad morphisms over $(\text{Id}_{\text{Set}^2}, \underline{\mathcal{J}}_{\times}) : \text{Id}_{\text{Set}^2} \rightarrow \underline{\mathcal{J}}_{\times}$ (see [subsection 3.5.1](#)), that we unravel here for concreteness.

Definition 6.3.2. A relational effect observation consists of a triple $\theta = (\theta_1, \theta_2, \theta_{\text{rel}}) : M_1 \otimes M_2 \rightarrow W$ where $\theta_1 : M_1 \rightarrow W_1, \theta_2 : M_2 \rightarrow W_2$ are (plain) monad morphisms, and

$$\theta_{\text{rel}} : ((m_1, m_2) : M_1 A_1 \times M_2 A_2) \rightarrow W_{\text{rel}}(A_1, A_2) (\theta_1 m_1) (\theta_2 m_2)$$

verify the two equations with respect to the monadic operations

$$\begin{aligned}
\theta_{\text{rel}}(\text{ret}^{M_1} a_1, \text{ret}^{M_2} a_2) &= \text{ret}^{W_{\text{rel}}} (a_1, a_2) : W_{\text{rel}}(A_1, A_2) (\theta_1 (\text{ret}^{M_1} a_1)) (\theta_2 (\text{ret}^{M_2} a_2)) \\
\theta_{\text{rel}}(\text{bind}^{M_1} m_1 f_1, \text{bind}^{M_2} m_2 f_2) &= \text{bind}^{W_{\text{rel}}} (\theta_1 m_1) (\theta_2 m_2) (\theta_{\text{rel}} m_{\text{rel}}) \theta_1 \circ f_1 \theta_2 \circ f_2 \theta_{\text{rel}} \circ (f_1 \times f_2)
\end{aligned}$$

Given a relational effect observation $\theta : M_1 \otimes M_2 \rightarrow W$, we can define in full generality the semantics of the relational judgment by the [Equation 6.16](#). We introduce the generic monadic rules in [Figure 6.5](#), and similarly to the simple setting obtain the following soundness theorem.

Theorem 6.3.2 (Soundness of monadic rules). *The relational rules in [Figure 6.5](#) are sound with respect to any relational effect observation θ , that is*

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

6.3.6 Relational specification monad transformers

Having a category of relational specification monads, we define a *relational specification monad transformer* to be a pointed endofunctor on this category ([Lüth and Ghani, 2002](#)). We show that 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 $W(A_1, A_2)$. 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 $\text{StT}(M)$ $A = S \rightarrow M(A \times S)$. The left relational state monad transformer StT_{rel} maps a relational specification monad $W(A_1, A_2) = (W_1 A_1, W_2 A_2, \lambda w_1 w_2. W_{\text{rel}}(A_1, A_2) w_1 w_2)$ to the relational specification monad with carrier

$$\text{StT}_{\text{rel}}(W)(A_1, A_2) = (\text{StT}(W_1) A_1, W_2 A_2, \lambda w_1 w_2. (s_1 : S_1) \rightarrow W_{\text{rel}}(A_1 \times S_1, A_2) (w_1 s_1) w_2)$$

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

`let` $\text{ret}_{\text{rel}}^{\text{StT}(W)}(a_1, a_2) : (s_1 : S_1) \rightarrow W_{\text{rel}}(A_1 \times S_1, A_2) (\text{ret}^{\text{StT}(W)_1}(a_1, s_1)) (\text{ret}^{W_2} a_2) = \lambda s_1. \text{ret}^{W_{\text{rel}}}((a_1, s_1), a_2)$

`let` $\text{bind}_{\text{rel}}^{\text{StT}(W)}(m_1 : \text{StT}(W)_1 A_1) (m_2 : W_2 A_2) (m_{\text{rel}} : \text{StT}(W)_{\text{rel}}(A_1, A_2) m_1 m_2)$
 $(f_1 : A_1 \rightarrow \text{StT}(W)_1 B_1) (f_2 : A_2 \rightarrow W_2 B_2)$
 $(f_{\text{rel}} : (a_1, a_2) : A_1 \times A_2 \rightarrow \text{StT}(W)_{\text{rel}}(B_1, B_2) (f_1 a_1) (f_2 a_2))$
 $: \text{StT}(W)_{\text{rel}}(B_1, B_2) (\text{bind}^{\text{StT}(W)_1} m_1 f_1) (\text{bind}^{W_2} m_2 f_2) =$
 $\lambda s_1. \text{bind}^{W_{\text{rel}}}(m_1 s_1) m_2 (m_{\text{rel}} s_1) (\lambda (a_1, s_1'). f_1 a_1 s_1') f_2 (\lambda ((a_1, s_1'), a_2). f_{\text{rel}}(a_1, a_2) s_1')$

The operation $\tau_1 : w_1 : \text{StT}(W)_1 A_1 \rightarrow (s_1 : S_1) \rightarrow W_{\text{rel}}(A_1 \times S_1, A_2) (w_1 s_1) (\text{ret}^{W_2} ())$ is given by

`let` $\tau_1^{\text{StT}(W)_{\text{rel}}}(w_1 : \text{StT}(W)_1 A_1) = \lambda s_1. \tau_1^W(w_1 s_1)$

Adding exceptions In a similar flavor, the exception monad transformer ExcT mapping a monad M to $\text{ExcT}(M)A = M(A + E_1)$ gives rise to its relational specification monad counterpart $\text{ExcT}_{\text{rel}}(W)(A_1, A_2) = (\text{ExcT}(W)_1 A_1, W_2 A_2, W_{\text{rel}}(A_1 + E_1, A_2))$. The `bind` operation is more involved here, and makes full use of the presence of the unary specifications.

`let` $\text{ret}^{\text{ExcT}(W)_{\text{rel}}}(a_1, a_2) : W_{\text{rel}}(A_1 + E_1, A_2) (\text{ret}^{\text{ExcT}(W)_1} a_1) (\text{ret}^{W_2} a_2) = \text{ret}^{W_{\text{rel}}}(\text{Inl } a_1, a_2)$

`let` $\text{bind}^{\text{ExcT}(W)_{\text{rel}}}(m_1 : \text{ExcT}(W)_1 A_1) (m_2 : W_2 A_2) (m_{\text{rel}} : \text{ExcT}(W)_{\text{rel}}(A_1, A_2) m_1 m_2)$
 $(f_1 : A_1 \rightarrow \text{ExcT}(W)_1 B_1) (f_2 : A_2 \rightarrow W_2 B_2)$
 $(f_{\text{rel}} : (a_1, a_2) : A_1 \times A_2 \rightarrow \text{ExcT}(W)_{\text{rel}}(B_1, B_2) (f_1 a_1) (f_2 a_2))$
 $: \text{ExcT}(W)_{\text{rel}}(B_1, B_2) (\text{bind}^{\text{ExcT}(W)_1} m_1 f_1) (\text{bind}^{W_2} m_2 f_2) =$
 $\text{bind}^{W_{\text{rel}}} m_1 m_2 m_{\text{rel}} (\lambda ae_1. \text{match } ae_1 \text{ with } | \text{Inl } a_1 \rightarrow f_1 a_1 | \text{Inr } e_1 \rightarrow \text{ret}^{W_1}(\text{Inr } e_1)) f_2$
 $(\lambda ae_1 a_2. \text{match } ae_1 \text{ with}$
 $| \text{Inl } a_1 \rightarrow f_{\text{rel}} a_1 a_2$
 $| \text{Inr } e_1 \rightarrow \text{bind}^{W_{\text{rel}}}(\tau_2(f_2 a_2)) (\lambda (), b_2). \text{ret}^{W_{\text{rel}}}(\text{Inr } e_1, b_2)))$

Note the crucial use of the $\tau_2 : w_2 : W_2 A_2 \rightarrow W_{\text{rel}}(\perp, A_2) (\text{ret}^{W_1} ()) w_2$ in the last error branch.

Putting these monad transformer to practice, we can finally define the full relational specification monad for exceptions validating the rules in [Figure 6.6](#) by first lifting the simple relational $W_{\text{rel}}^{\text{Pure}}$ and applying the exception transformers on both left and right sides. Further, applications would involve specifications relating state and exceptions with rollback state.

6.4 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

$$\begin{array}{c}
\frac{}{\Gamma^r \vdash \text{throw } e_1 \{ \lambda \varphi_1. \varphi_1 (\text{inr } e_1) \} \sim \text{ret}^{\text{Exc}} a_2 \{ \text{ret}^{W_2^{\text{Exc}}} a_2 \} \mid \lambda \varphi. \varphi (\text{inl } e_1, \text{inl } a_2)} \\
\frac{}{\Gamma^r \vdash \text{ret}^{\text{Exc}} a_1 \{ \text{ret}^{W_1^{\text{Exc}}} a_1 \} \sim \text{throw } e_2 \{ \lambda \varphi_2. \varphi_2 (\text{inr } e_2) \} \mid \lambda \varphi. \varphi (\text{inl } a_1, \text{inr } e_2)} \\
\frac{\Gamma^r \vdash c_1 \{ w_1 \} \sim c_2 \{ w_2 \} \mid w_{\text{rel}} \quad \Gamma^r \vdash c_1^{\text{err}} \{ w_1^{\text{err}} \} \sim c_2^{\text{err}} \{ w_2^{\text{err}} \} \mid w_{\text{rel}}^{\text{err}}}{\Gamma^r \vdash \text{catch } c_1 c_1^{\text{err}} \{ w^{\text{catch}} w_1 w_1^{\text{err}} \} \sim \text{catch } c_2 c_2^{\text{err}} \{ w^{\text{catch}} w_2 w_2^{\text{err}} \} \mid w_{\text{rel}}^{\text{catch}} w_{\text{rel}} w^{\text{err}}} \\
\text{let } w^{\text{catch}} (w : W^{\text{Exc}} A) (werr : E \rightarrow W^{\text{Exc}} A) : W A = \\
\lambda \varphi. w (\lambda ae. \text{match } ae \text{ with } \mid \text{Inl } a \rightarrow \text{ret}^{W^{\text{Exc}}} a \varphi \mid \text{Inr } e \rightarrow werr e \varphi) \\
\text{let } w_{\text{rel}}^{\text{catch}} (w : W_{\text{rel}}^{\text{Exc}} (A_1, A_2)) (werr_1 : E_1 \rightarrow W_{\text{rel}}^{\text{Exc}} A_1) (werr_2 : E_2 \rightarrow W_{\text{rel}}^{\text{Exc}} A_2) \\
(werr_{\text{rel}} : E_1 \times E_2 \rightarrow W_{\text{rel}}^{\text{Exc}} (A_1, A_2)) : W_{\text{rel}}^{\text{Exc}} (A_1, A_2) = \\
\lambda \varphi. w (\lambda (ae_1, ae_2). \text{match } ae_1, ae_2 \text{ with} \\
\mid \text{Inl } a_1, \text{Inl } a_2 \rightarrow \text{ret}^{W_{\text{rel}}^{\text{Exc}}} (a_1, a_2) \varphi \\
\mid \text{Inr } e_1, \text{Inl } a_2 \rightarrow werr_1 e_1 (\lambda ae_1 \rightarrow \varphi (ae_1, \text{Inl } a_2)) \\
\mid \text{Inl } a_1, \text{Inr } e_2 \rightarrow werr_2 e_2 (\lambda ae_2 \rightarrow \varphi (\text{Inl } a_1, ae_2)) \\
\mid \text{Inr } e_1, \text{Inr } e_2 \rightarrow werr_{\text{rel}} (e_1, e_2) \varphi)
\end{array}$$

Figure 6.6: Rules for exceptions

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 $\models_{\theta} c_1 \sim c_2 \{ w \}$.

First of all, we need a notion of product program. In the setting of monadic programs, we capture a product program of $c_1 : M_1 A_1$ and $c_2 : M_2 A_2$ as a program $c : P(A_1, A_2)$, where P is a relative monad over $(A_1, A_2) \mapsto A_1 \times A_2$ (see [section 3.2](#)). We can think of $c : P(A_1, A_2)$ 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 – $M_1 A_1 = \text{St}_{S_1} A_1$ and $M_2 A_2 = \text{St}_{S_2} A_2$ – inhabiting the relative monad $P^{\text{St}}(A_1, A_2) = \text{St}_{S_1 \times S_2}(A_1 \times A_2)$. To complete the definition of product programs, we also need to explain when a concrete product program $c : P(A_1, A_2)$ is capturing the behavior of $c_1 : M_1 A_1$ and $c_2 : M_2 A_2$. 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

$$\frac{a_1 : A_1 \quad a_2 : A_2}{\text{ret}^{M_1} a_1 \times \text{ret}^{M_2} a_2 \rightsquigarrow \text{ret}^P (a_1, a_2)} \quad \frac{m_1 \times m_2 \rightsquigarrow m_{\text{rel}} \quad \forall a_1 a_2, f_1 a_1 \times f_2 a_2 \rightsquigarrow f_{\text{rel}} (a_1, a_2)}{\text{bind}^{M_1} m_1 f_1 \times \text{bind}^{M_2} m_2 f_2 \rightsquigarrow \text{bind}^P m_{\text{rel}} f_{\text{rel}}}$$

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 ([subsection 6.2.1](#)) 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 $P(A_1, A_2) = \text{St}_{S_1 \times S_2}(A_1 \times A_2)$ with specifications provided by the simple relational specification monad $W_{\text{rel}}^{\text{St}}$, and the effect observation $\zeta : P \rightarrow W_{\text{rel}}^{\text{St}}$ obtained by lifting the unary effect observation $\theta^{\text{St}} : \text{St} \rightarrow W^{\text{St}}$ from [section 2.4](#),

resulting in

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

where $\sigma : (A_1 \times A_2) \times (S_1 \times S_2) \rightarrow (A_1 \times S_1) \times (A_2 \times S_2)$ 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 as stated below.

Theorem 6.4.1 (Soundness of product programs). *If $c_1 \times c_2 \rightsquigarrow c$ and $\zeta(c) \leq w$, then $\models_{\theta_{\text{rel}}} c_1 \sim c_2 \{w\}$.*

For state, this theorem is proved by analyzing the relation $c_1 \times c_2 \rightsquigarrow c$ and showing in each case that our choice of θ_{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.

6.5 Related work

Many different relational verification tools have been proposed, making different trade-offs, 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.

Relators Gavazzo (2018) recently proposed a type system for differential privacy that is parameterized by a signature of algebraic effects. The type system is given a relational interpretation in terms of *relators*, which lift relations on values to relations on monadic computations:

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

Lochbihler (2018) also used relators in a recent library for effect polymorphic definitions and proofs in Isabelle/HOL, based on value-monomorphic monads. There seems to be a strong connection between such relators and the effect observations going into one of the simplest relational specification monads we consider: $(A_1 \times A_2 \rightarrow \mathbb{P}) \rightarrow \mathbb{P}$. Such an effect observation has type

$$MA_1 \times MA_2 \rightarrow (A_1 \times A_2 \rightarrow \mathbb{P}) \rightarrow \mathbb{P},$$

which is isomorphic to the type of the relator Γ above (this is obvious to see by just swapping the two arguments). While further investigating this connection is very interesting, since relators are inherently lax this requires first working out the theory of lax effect observations, for which the relative monad morphism laws hold with \leq instead of $=$ (see the end of subsection 6.2.2). While we expect such an extension to our framework to be possible and generally useful, the technical development is involved even for the simple setting of section 6.2, so we leave it for future work.

Relational models of type theory The relational dependent type theory RDTT we employ in section 6.3 and the translation from the ambient type theory to RDTT is inspired by the parametricity translations for dependent type theory of Bernardy and Lasson (2011). Relations on types can also be internalized inside dependent type theory, making them first class citizens, as in the work of Nuyts and Devriese (2018) where it is used to characterize various properties such as continuity or parametricity. The work of Cavallo and Harper (2019) on cubical models of type theory introduce an interesting property on these internalizations called *relativity*, playing a similar role as univalence for equivalences.

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., 2015; 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 Ștefan Ciobăcă 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).

6.6 Conclusion

We introduced in this chapter semantics tools to analyse relational program logics for arbitrary monadic effects by extending the notions of specification monads and effect observations to this relational setting. We can then reconstruct relational program logics for specific effects in a principled way using the general building blocks provided in subsection 6.2.4 and combining them with effect specific rules along the lines of subsection 6.2.5. An 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), but also investigating at the categorical level notions of presentations of relative monads, in connection with the theory of monads with arities (Berger et al., 2012). Finally, it also remains to be seen whether our notion of relational effect observations can be generalized to turn the laws from equalities to inequalities. The proof of Thm. 6.2.2 from subsection 6.2.4 would be easy to extend, and this extension would allow for more examples, including the ones previously done using relators such as simulations for nondeterminism (Dal Lago et al., 2017), and would also make certain examples such as relational partial correctness easier. Yet a technical development following the ideas of chapter 3 seems more involved, even for the simple setting of section 6.2.

Bibliography

- C. Abate, R. Blanco, D. Garg, C. Hrițcu, M. Patrignani, and J. Thibault. [Journey beyond full abstraction: Exploring robust property preservation for secure compilation](#). *CSF*, 2019. To Appear. [5](#)
- J. Adámek, S. Milius, N. Bowler, and P. B. Levy. [Coproducts of monads on set](#). *LICS*. 2012. [4](#)
- A. Aguirre, G. Barthe, M. Gaboardi, D. Garg, and P.-Y. Strub. [A relational logic for higher-order programs](#). *ICFP*, 2017. [6](#), [111](#)
- D. Ahman and T. Uustalu. [Update monads: Cointerpreting directed containers](#). In *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France*, 2013. [16](#), [17](#), [23](#), [24](#)
- D. Ahman, C. Hrițcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. [Dijkstra monads for free](#). *POPL*. 2017. [4](#), [7](#), [17](#), [63](#), [89](#)
- D. Ahman, C. Fournet, C. Hrițcu, K. Maillard, A. Rastogi, and N. Swamy. [Recalling a witness: Foundations and applications of monotonic state](#). *PACMPL*, 2(POPL):65:1–65:30, 2018. [7](#), [17](#), [79](#), [89](#)
- A. Ahmed, D. Dreyer, and A. Rossberg. [State-dependent representation independence](#). *POPL*. 2009. [113](#)
- T. Altenkirch and A. Kaposi. [Type theory in type theory using quotient inductive types](#). In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016. [15](#)
- T. Altenkirch, J. Chapman, and T. Uustalu. [Monads need not be endofunctors](#). *LMCS*, 11(1), 2015. [29](#), [33](#), [34](#), [42](#), [46](#), [49](#), [50](#), [95](#), [96](#)
- T. Altenkirch, N. A. Danielsson, and N. Kraus. [Partiality, revisited - the partiality monad as a quotient inductive-inductive type](#). *FOSSACS*, 2017. [13](#)
- R. M. Amadio and P. Curien. *Domains and lambda-calculi*, volume 46 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1998. [12](#)
- J. Andrianopoulos. [Remarks on units of skew monoidal categories](#). *Applied Categorical Structures*, 25(5):863–873, 2017. [50](#)
- 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. [5](#), [112](#)
- R. Atkey. [Syntax for free: Representing syntax with binding using parametricity](#). In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, 2009. [65](#)

- R. Atkey, S. Lindley, and J. Yallop. [Unembedding domain-specific languages](#). In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, 2009. [64](#)
- P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in coq. In T. Uustalu, editor, *Mathematics of Program Construction*. 2006. [20](#)
- A. Banerjee, D. A. Naumann, and M. Nikouei. [Relational logic with framing and hypotheses](#). *FSTTCS*. 2016. [5](#), [6](#), [111](#)
- G. Barthe, B. Grégoire, and S. Zanella-Béguelin. [Formal certification of code-based cryptographic proofs](#). *POPL*, 2009. [5](#), [111](#)
- G. Barthe, P. R. D’Argenio, and T. Rezk. [Secure information flow by self-composition](#). *MSCS*, 21(6):1207–1252, 2011. [109](#), [112](#)
- 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. [5](#), [111](#)
- G. Barthe, B. Köpf, F. Olmedo, and S. Zanella-Béguelin. [Probabilistic relational reasoning for differential privacy](#). *TOPLAS*, 35(3):9:1–9:49, 2013b. [5](#), [6](#), [111](#)
- G. Barthe, C. Fournet, B. Grégoire, P. Strub, N. Swamy, and S. Zanella-Béguelin. [Probabilistic relational verification for cryptographic implementations](#). *POPL*. 2014. [5](#), [6](#), [111](#)
- 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*. 2015. [5](#), [6](#), [112](#)
- G. Barthe, J. M. Crespo, and C. Kunz. [Product programs and relational program logics](#). *JLAMP*, 85(5):847–859, 2016. [6](#), [97](#), [109](#), [111](#), [112](#)
- G. Barthe, B. Grégoire, J. Hsu, and P. Strub. [Coupling proofs are probabilistic product programs](#). *POPL*. 2017. [111](#)
- G. Barthe, R. Eilers, P. Georgiou, B. Gleiss, L. Kovács, and M. Maffei. [Verifying relational properties using trace logic](#). Draft, 2019. [5](#)
- D. A. Basin, A. Lochbihler, and S. R. Sefidgar. [CryptHOL: Game-based proofs in higher-order logic](#). *IACR Cryptology ePrint Archive*, 2017:753, 2017. [111](#)
- J. Beck. [Distributive laws](#). In *Seminar on Triples and Categorical Homology Theory*. 1969. [15](#)
- J. Bénabou. Introduction to bicategories. In *Reports of the Midwest Category Seminar*. 1967. [29](#), [31](#), [49](#)
- J. Benabou. [Distributors at work](#), 2000. [36](#)
- N. Benton. [Simple relational correctness proofs for static analyses and program transformations](#). *POPL*. 2004. [5](#), [6](#), [111](#)
- N. Benton, J. Hughes, and E. Moggi. [Monads and effects](#). *APPSEM*. 2000. [12](#), [25](#)
- N. Benton, A. Kennedy, L. Beringer, and M. Hofmann. [Relational semantics for effect-based program transformations: higher-order store](#). *POPL*. 2009. [113](#)
- N. Benton, M. Hofmann, and V. Nigam. [Proof-relevant logical relations for name generation](#). *TLCA*. 2013. [113](#)

- N. Benton, M. Hofmann, and V. Nigam. [Abstract effects and proof-relevant logical relations](#). *POPL*. 2014. [113](#)
- 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. [113](#)
- C. Berger, P.-A. Mellies, 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 redispached with Section 2; Subsections on symmetric operads (3.14) and symmetric simplicial sets (4.17) added; Bibliography completed. [113](#)
- J. Bernardy and M. Lasson. [Realizability and parametricity in pure type systems](#). *FOSSACS*, 2011. [112](#)
- J. Bernardy and G. Moulin. [Type-theory in color](#). In *ACM SIGPLAN International Conference on Functional Programming, ICFP’13, Boston, MA, USA - September 25 - 27, 2013*, 2013. [65](#)
- J. Bernardy, T. Coquand, and G. Moulin. [A presheaf model of parametric type theory](#). *Electr. Notes Theor. Comput. Sci.*, 319:67–82, 2015. [65](#)
- K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hritcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. Lorch, K. Maillard, J. Pang, B. Parno, J. Protzenko, T. Ramanandaro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Zanella-Béguelin, and J.-K. Zinzindohoué. [Everest: Towards a verified, drop-in replacement of HTTPS](#). *SNAPL*, 2017. [7](#)
- B. Blanchet, M. Abadi, and C. Fournet. [Automated verification of selected equivalences for security protocols](#). *J. Log. Algebr. Program.*, 75(1):3–51, 2008. [5](#), [113](#)
- A. Blass. [Words, free algebras, and coequalizers](#). *Fundamenta Mathematicae*, 117(2):117–160, 1983. [15](#)
- S. Boulier, P. Pédrot, and N. Tabareau. [The next 700 syntactical models of type theory](#). *CPP*, 2017. [25](#), [27](#), [102](#)
- J. Bourke and S. Lack. [Free skew monoidal categories](#). *Journal of Pure and Applied Algebra* 222(10):3255–3281, 2018a. [50](#)
- J. Bourke and S. Lack. [Skew monoidal categories and skew multicategories](#). *Journal of Algebra*, 506:237–266, 2018b. [50](#)
- 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. [4](#), [98](#)
- M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. [Proving acceptability properties of relaxed nondeterministic approximate programs](#). *PLDI*. 2012. [5](#), [6](#), [111](#)
- 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. [27](#)
- E. Cavallo and R. Harper. [Parametric cubical type theory](#). To appear at ICFP, 2019. [112](#)
- R. Chadha, V. Cheval, Ștefan Ciobăcă, and S. Kremer. [Automated verification of equivalence properties of cryptographic protocols](#). *ACM Trans. Comput. Log.*, 17(4):23:1–23:32, 2016. [5](#), [113](#)

- A. Chlipala. [Parametric higher-order abstract syntax for mechanized semantics](#). *ICFP*. 2008. [64](#), [65](#)
- E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann. [Relational cost analysis](#). *POPL*, 2017. [5](#), [112](#)
- M. R. Clarkson and F. B. Schneider. [Hyperproperties](#). *J. Comput. Secur.*, 18(6):1157–1210, 2010. [5](#), [112](#)
- Ştefan Ciobăcă, D. Lucanu, V. Rusu, and G. Rosu. [A language-independent proof system for full program equivalence](#). *Formal Asp. Comput.*, 28(3):469–497, 2016. [5](#), [113](#)
- P. Curien, R. Garner, and M. Hofmann. [Revisiting the categorical interpretation of dependent type theory](#). *Theor. Comput. Sci.*, 546:99–119, 2014. [30](#)
- U. Dal Lago, F. Gavazzo, and P. B. Levy. [Effectful applicative bisimilarity: Monads, relators, and Howe’s method](#). *LICS*. 2017. [113](#)
- G. A. Delbianco and A. Nanevski. [Hoare-style reasoning with \(algebraic\) continuations](#). *ICFP*. 2013. [3](#), [13](#)
- I. Di Liberti and F. Loregian. [On the unicity of formal category theories](#), 2019. [50](#)
- E. W. Dijkstra. [Guarded commands, nondeterminacy and formal derivation of programs](#). *CACM*, 18(8):453–457, 1975. [3](#), [4](#)
- D. Dreyer, G. Neis, A. Rossberg, and L. Birkedal. [A relational modal logic for higher-order stateful ADTs](#). *POPL*. 2010. [113](#)
- D. Dreyer, A. Ahmed, and L. Birkedal. [Logical step-indexed logical relations](#). *Logical Methods in Computer Science*, 7(2), 2011. [113](#)
- 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. [113](#)
- J. Egger, R. E. Møgelberg, and A. Simpson. [The enriched effect calculus: syntax and semantics](#). *LogCom*, 24(3):615–654, 2014. [60](#)
- 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. [112](#)
- 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. [112](#)
- M. Fiore, N. Gambino, M. Hyland, and G. Winskel. [Relative pseudomonads, kleisli bicategories, and substitution monoidal structures](#). *Selecta Mathematica*, 24(3):2791–2830, 2018. [50](#)
- R. W. Floyd. [Nondeterministic algorithms](#). *J. ACM*, 14(4):636–644, 1967. [4](#)
- C. Führmann. [Varieties of effects](#). *FOSSACS*, 2002. [98](#)
- S. Fujii, S. Katsumata, and P. Mellies. [Towards a formal theory of graded monads](#). In *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, 2016. [9](#), [86](#), [87](#)

- M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. [Linear dependent types for differential privacy](#). *POPL*. 2013. [112](#)
- F. Gavazzo. [Quantitative behavioural reasoning for higher-order effectful programs: Applicative distances](#). *LICS*. 2018. [5](#), [112](#), [113](#)
- 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. [5](#)
- 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. [5](#)
- M. Giry. [A categorical approach to probability theory](#). *Categorical Aspects of Topology and Analysis*. 1982. [15](#)
- 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. [5](#), [113](#)
- N. Grimm, K. Maillard, C. Fournet, C. Hrițcu, 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. [7](#), [26](#)
- I. Hasuo. [Generic weakest precondition semantics from monads enriched with order](#). *Theor. Comput. Sci.*, 604:2–29, 2015. [25](#)
- 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. [5](#), [112](#)
- C. Hermida, U. S. Reddy, and E. P. Robinson. [Logical relations and parametricity - A reynolds programme for category theory and programming languages](#). *Electr. Notes Theor. Comput. Sci.*, 303:149–180, 2014. [58](#)
- C. A. R. Hoare. [An axiomatic basis for computer programming](#). *Commun. ACM*, 12(10):576–580, 1969. [2](#)
- C. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. [The marriage of bisimulations and kripke logical relations](#). *POPL*. 2012. [5](#), [113](#)
- C. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. [A logical step forward in parametric bisimulations](#). Technical Report MPI-SWS-2014-003, 2014. [5](#), [113](#)
- M. Hyland, G. D. Plotkin, and J. Power. [Combining effects: Sum and tensor](#). *Theor. Comput. Sci.*, 357(1-3):70–99, 2006. [15](#), [25](#)
- M. Hyland, P. B. Levy, G. D. Plotkin, and J. Power. [Combining algebraic effects with continuations](#). *Theor. Comput. Sci.*, 375(1-3):20–40, 2007. [4](#), [24](#), [25](#)
- G. Jaber, N. Tabareau, and M. Sozeau. [Extending type theory with forcing](#). In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, 2012. [25](#)
- G. Jaber, G. Lewertowski, P. Pédro, M. Sozeau, and N. Tabareau. [The definitional side of the forcing](#). In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, 2016. [25](#)

- B. Jacobs. [Dijkstra monads in monadic computation](#). *CMCS*, 2014. [4](#), [17](#), [25](#)
- B. Jacobs. [Dijkstra and Hoare monads in monadic computation](#). *Theor. Comput. Sci.*, 604:30–45, 2015. [4](#), [17](#), [25](#)
- B. Jacobs. [A recipe for state-and-effect triangles](#). *Logical Methods in Computer Science*, 13(2), 2017. [25](#)
- M. Jaskelioff and E. Moggi. [Monad transformers as monoid transformers](#). *Theor. Comput. Sci.*, 411(51-52):4441–4466, 2010. [63](#), [73](#)
- A. Joyal and R. Street. [The geometry of tensor calculus, i](#). *Advances in Mathematics*, 88(1):55 – 112, 1991. [31](#)
- R. Jung, R. Krebbers, J. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. [Iris from the ground up: A modular foundation for higher-order concurrent separation logic](#). *J. Funct. Program.*, 28:e20, 2018. [79](#)
- B. L. Kaminski, J. Katoen, C. Matheja, and F. Olmedo. [Weakest precondition reasoning for expected run-times of probabilistic programs](#). In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, 2016. [20](#)
- O. Kammar, P. B. Levy, S. K. Moss, and S. Staton. [A monad for full ground reference cells](#). In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, 2017. [25](#)
- A. Kaposi and A. Kovács. [Signatures and induction principles for higher inductive-inductive types](#). *arXiv:1902.00297*, 2019. [89](#)
- A. Kaposi, A. Kovács, and T. Altenkirch. [Constructing quotient inductive-inductive types](#). *PACMPL*, 3(POPL):2:1–2:24, 2019. [72](#)
- C. Kapulkin and P. L. Lumsdaine. [Homotopical inverse diagrams in categories with attributes](#), 2018. [103](#)
- S. Katsumata. [A semantic formulation of tt-lifting and logical predicates for computational metalanguage](#). In *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings*, 2005. [87](#)
- S. Katsumata. [Relating computational effects by \$\top\top\$ -lifting](#). *Inf. Comput.*, 222:228–246, 2013. [25](#), [87](#), [89](#)
- S. Katsumata. [Parametric effect monads and semantics of effect systems](#). *POPL*. 2014. [5](#), [20](#), [25](#), [75](#), [86](#), [89](#)
- S. Katsumata and T. Sato. [Preorders on monads and coalgebraic simulations](#). *FOSSACS*, 2013. [42](#)
- S. Katsumata, T. Sato, and T. Uustalu. [Codensity lifting of monads and its dual](#). *Logical Methods in Computer Science*, 14(4), 2018. [89](#)
- G. Kelly. *Basic Concepts of Enriched Category Theory*. Lecture note series / London mathematical society. Cambridge University Press, 1982. [33](#), [36](#)
- G. M. Kelly and R. Street. Review of the elements of 2-categories. In G. M. Kelly, editor, *Category Seminar*. 1974. [29](#), [49](#)

- O. Kiselyov, A. Sabry, and C. Swords. [Extensible effects: an alternative to monad transformers](#). In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, 2013. 72
- V. Koutavas and M. Wand. [Small bisimulations for reasoning about higher-order imperative programs](#). *POPL*. 2006. 113
- S. Kundu, Z. Tatlock, and S. Lerner. [Proving optimizations correct using parameterized program equivalence](#). *PLDI*. 2009. 5, 113
- S. Lack. A 2-categories companion. *Institute for Mathematics and its Applications*, 2009. 29
- S. Lack and R. Street. [The formal theory of monads ii](#). *Journal of Pure and Applied Algebra*, 175 (1):243 – 265, 2002. Special Volume celebrating the 70th birthday of Professor Max Kelly. 29, 49
- S. Lack and R. Street. Skew-monoidal reflection and lifting theorems. *Theory and Applications of Categories*, 30:985–1000, 2015. 50
- S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. [SYMDIFF: A language-agnostic semantic diff tool for imperative programs](#). *CAV*. 2012. 5, 112
- K. R. M. Leino. [Efficient weakest preconditions](#). *Inf. Process. Lett.*, 93(6):281–288, 2005. 18
- T. Letan, Y. Régis-Gianas, P. Chifflier, and G. Hiet. [Modular verification of programs with effects and effect handlers in coq](#). *FM*. 2018. 26
- P. B. Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Semantics Structures in Computation*. Springer, 2004. 25
- S. Liang, P. Hudak, and M. P. Jones. [Monad transformers and modular interpreters](#). *POPL*. 1995. 51, 72
- S. Lindley and I. Stark. [Reducibility and \$\top\top\$ -lifting for computation types](#). *TLCA*. 2005. 25
- A. Lochbihler. [Effect polymorphism in higher-order logic \(proof pearl\)](#). *JAR*, 2018. 26, 112
- D. Lucanu and V. Rusu. [Program equivalence by circular reasoning](#). *Formal Asp. Comput.*, 27(4): 701–726, 2015. 113
- C. Lüth and N. Ghani. [Composing monads using coproducts](#). *ICFP*. 2002. 52, 108
- K. Maillard and P. Melliès. [A fibrational account of local states](#). In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, 2015. 25, 68
- K. Maillard, D. Ahman, R. Atkey, G. Martínez, C. Hrițcu, E. Rivas, and E. Tanter. [Dijkstra monads for all](#). To appear at ICFP, 2019a. 7
- K. Maillard, C. Hrițcu, E. Rivas, and A. V. Muyllder. [The next 700 relational program logics](#). arXiv:1907.05244, 2019b. 7
- G. Malecha, G. Morrisett, and R. Wisnesky. [Trace-based verification of imperative programs with I/O](#). *J. Symb. Comput.*, 46(2):95–118, 2011. 24
- C. Matache and S. Staton. [A sound and complete logic for algebraic effects](#). *FoSSaCS*. 2019. 26
- C. McBride. [Turing-completeness totally free](#). *MPC*. 2015. 13, 81

- P. Mellies. [Local states in string diagrams](#). In *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, 2014. [25](#)
- J. C. Mitchell. [Representation independence and data abstraction](#). In *POPL*. 1986. [113](#)
- E. Moggi. [Computational lambda-calculus and monads](#). *LICS*. 1989. [2](#), [12](#), [25](#), [26](#)
- E. Moggi. [A semantics for evaluation logic](#). *Fundam. Inform.*, 22(1/2):117–152, 1995. [23](#), [26](#)
- C. Morgan. *Programming from Specifications (2nd Ed.)*. Prentice Hall, Hertfordshire, UK, 1994. [26](#)
- G. Munch-Maccagnoni. [Syntax and Models of a non-Associative Composition of Programs and Proofs. \(Syntaxe et modèles d'une composition non-associative des programmes et des preuves\)](#). PhD thesis, Paris Diderot University, France, 2013. [60](#)
- D. Myers, Jaz. [String diagrams for double categories and equipments](#), 2016. [37](#)
- A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. [Ynot: dependent types for imperative programs](#). *ICFP*. 2008a. [2](#), [3](#)
- A. Nanevski, J. G. Morrisett, and L. Birkedal. [Hoare type theory, polymorphism and separation](#). *JFP*, 18(5-6):865–911, 2008b. [2](#)
- 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. [3](#), [5](#), [6](#), [111](#)
- D. A. Naumann. [From coupling relations to mated invariants for checking information flow](#). *ESORICS*. 2006. [112](#)
- A. Nuyts and D. Devriese. [Degrees of relatedness: A unified framework for parametricity, irrelevance, ad hoc polymorphism, intersections, unions and algebra in dependent type theory](#). In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, 2018. [112](#)
- S. S. Owicki and D. Gries. [Verifying properties of parallel programs: An axiomatic approach](#). *CACM*, 19(5):279–285, 1976. [23](#)
- P. Pédrot and N. Tabareau. [Failure is not an option - an exceptional type theory](#). *ESOP*, 2018. [27](#)
- A. Petcher and G. Morrisett. [The foundational cryptography framework](#). *POST*. 2015. [5](#), [6](#), [111](#)
- F. Pfenning and C. Elliott. [Higher-order abstract syntax](#). In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, 1988. [64](#)
- M. Piróg, T. Schrijvers, N. Wu, and M. Jaskelioff. [Syntax and semantics for operations with scopes](#). In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, 2018. [12](#)
- A. M. Pitts. [Evaluation logic](#). In *IV Higher Order Workshop, Banff 1990*. Springer, 1991. [23](#), [26](#)
- G. D. Plotkin and J. Power. [Notions of computation determine monads](#). *FOSSACS*, 2002. [12](#), [25](#)
- G. D. Plotkin and M. Pretnar. [A logic for algebraic effects](#). In *LICS*. 2008. [26](#)
- G. D. Plotkin and M. Pretnar. [Handlers of algebraic effects](#). *ESOP*. 2009. [12](#)

- A. Power. [A general coherence result](#). *Journal of Pure and Applied Algebra*, 57(2):165 – 173, 1989. [30](#)
- J. Power. [Semantics for local computational effects](#). *Electr. Notes Theor. Comput. Sci.*, 158:355–371, 2006. [25](#)
- J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hrițcu, K. Bhargavan, C. Fournet, and N. Swamy. [Verified low-level programming embedded in F*](#). *PACMPL*, 1(ICFP):17:1–17:29, 2017. [79](#)
- W. Qu, M. Gaboardi, and D. Garg. [Relational cost analysis for functional-imperative programs](#). To appear at ICFP, 2019. [5](#), [6](#), [112](#)
- 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. [5](#), [6](#), [111](#)
- C. Rauch, S. Goncharov, and L. Schröder. [Generic Hoare logic for order-enriched effects with exceptions](#). *WADT*, 2016. [26](#), [42](#)
- A. Sabelfeld and A. C. Myers. [Language-based information-flow security](#). *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003. [5](#), [112](#)
- 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. [113](#)
- 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. [5](#), [6](#), [111](#)
- T. Schrijvers, M. Piróg, N. Wu, and M. Jaskelioff. [Monad transformers and modular algebraic effects: what binds them together](#). In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, 2019. [72](#)
- M. Shulman. Framed bicategories and monoidal fibrations. *Theory and applications of categories*, 20(18):650–738, 2008. [29](#), [34](#), [38](#), [39](#), [49](#)
- M. Shulman. [Univalence for inverse diagrams and homotopy canonicity](#). *Mathematical Structures in Computer Science*, 25:1203–1277, 2014. [103](#)
- A. Simpson and N. F. W. Voorneveld. [Behavioural equivalence via modalities for algebraic effects](#). *ESOP*. 2018. [26](#)
- M. Sousa and I. Dillig. [Cartesian Hoare logic for verifying k-safety properties](#). *PLDI*. 2016. [5](#), [6](#), [111](#), [112](#)
- M. Sozeau and C. Mangin. [Equations reloaded: High-level dependently-typed functional programming and proving in coq](#). *Proc. ACM Program. Lang.*, 3(ICFP):86:1–86:29, 2019. [64](#)
- S. Staton. [Completeness for algebraic theories of local state](#). In *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, 2010. [25](#)
- S. Staton. [Algebraic effects, linearity, and quantum programming languages](#). In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, 2015. [25](#)
- R. Street. [The formal theory of monads](#). *Journal of Pure and Applied Algebra*, 2, 1972. [29](#), [49](#)

- R. Street and R. Walters. *Yoneda structures on 2-categories*. *Journal of Algebra*, 50(2):350 – 379, 1978. 50
- E. Sumii. *A complete characterization of observational equivalence in polymorphic lambda-calculus with general references*. *CSL*. 2009. 113
- N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. *Verifying higher-order programs with the Dijkstra monad*. *PLDI*, 2013. 3, 4, 17, 89
- N. Swamy, C. Hrițcu, 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. 3, 4, 17, 75, 89
- W. Swierstra and T. Baanen. *A predicate transformer semantics for effects*, 2019. 26
- K. Szlachányi. *Skew-monoidal categories and bialgebroids*. *Advances in Mathematics*, 231(3): 1694 – 1730, 2012. 50
- T. Terauchi and A. Aiken. *Secure information flow as a safety problem*. *SAS*. 2005. 112
- A. Timany and L. Birkedal. *Mechanized relational verification of concurrent programs with continuations*. To appear at ICFP, 2019. 5, 113
- A. Timany and B. Jacobs. *Category theory in Coq 8.5*. *FSCD*, 2016. 64
- A. Timany, L. Stefanescu, 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. 5
- 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. 95, 102
- T. Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013. 7, 76
- D. Unruh. *Quantum relational Hoare logic*. *PACMPL*, 3(POPL):33:1–33:31, 2019. 5, 6, 111
- T. Uustalu, N. Veltri, and N. Zeilberger. *The sequent calculus of skew monoidal categories*. *Electronic Notes in Theoretical Computer Science*, 341:345 – 370, 2018. Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIV). 50
- N. Voorneveld. *Quantitative logics for equivalence of effectful programs*. *MFPS*. 2019. To appear. 26
- P. Wadler. *Comprehending monads*. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. 1990. 25
- 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. 5, 113
- D. Winograd-Cort, A. Haeblerlen, A. Roth, and B. C. Pierce. *A framework for adaptive differential privacy*. *PACMPL*, 1(ICFP):10:1–10:29, 2017. 112
- R. J. Wood. *Abstract pro arrows i*. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 23(3):279–290, 1982. 50

- R. J. Wood. [Proarrows ii](#). *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 26(2): 135–168, 1985. [50](#)
- H. Yang. [Relational separation logic](#). *Theor. Comput. Sci.*, 375(1-3):308–334, 2007. [5](#), [6](#), [111](#)
- H. Yasuoka and T. Terauchi. [Quantitative information flow as safety and liveness hyperproperties](#). *Theor. Comput. Sci.*, 538:167–182, 2014. [112](#)
- A. Zaks and A. Pnueli. [CoVaC: Compiler validation by program analysis of the cross-product](#). *FM*. 2008. [112](#)
- N. Zeilberger. [The Logical Basis of Evaluation Order and Pattern-Matching](#). PhD thesis, Carnegie Mellon University, 2009. [113](#)
- D. Zhang and D. Kifer. [LightDP: towards automating differential privacy proofs](#). *POPL*. 2017. [5](#), [6](#), [112](#)
- 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. [112](#)

Principes de la vérification de programmes à effets monadiques arbitraires

Les effets de bord présent dans les langages de programmation tel que l'état mutable, la divergence ou le non déterminisme sont capturés de manière élégante par des monades. Plusieurs systèmes ont été proposés pour spécifier et prouver que des programmes manipulant une certaine combinaison d'effets respectent leur spécification. Par exemple, la logique de Hoare permet de vérifier la correction de programmes manipulant la mémoire en stipulant des prédicats sur les états initiaux et finaux.

Le but de cette thèse est de définir un cadre sémantique générique pour vérifier que des programmes avec des effets monadique arbitraire respectent de telles spécifications.

Le point de départ de ce travail sont les monades de Dijkstra, des structures monadiques classifiant un ensemble de calculs avec effets respectant une certaine spécification, elle-même appartenant à une monade. Ces monades de Dijkstra se sont révélées efficaces en pratique pour vérifier des programmes à effets, notamment dans le cadre du langage F^* où elles permettent de calculer des conditions de vérifications pour une certaine classe d'effets monadiques. Dans notre travail, nous étudions les structures algébriques sous-jacentes aux monades de Dijkstra, révélant un cadre riche pour la vérification de propriétés non relationnelles en présence d'effets arbitraires. Un point important est la correspondance entre monades de Dijkstra et observations d'effets, c'est à dire une application entre une monade de calcul et une monade de spécification respectant la structure monadique.

Ces observations d'effets permettent des interprétations diverses des effets tel que la correction totale ou partielle, ou encore le non-déterminisme angélique ou démoniaque. Elles sont aussi la notion clé pour étendre notre cadre à des propriétés relationnelles, c'est à dire des propriétés décrivant le comportement de plusieurs exécutions d'un programme ou des exécution de plusieurs programmes, comme la non-interférence ou l'équivalence observationnelle de programmes. Pour parvenir à cette généralisation nous développons les notions de monades de spécifications et d'observation d'effets dans le cadre des monades relatives.

Principles of Program Verification for Arbitrary Monadic Effects

Computational monads are a convenient algebraic gadget to uniformly represent side-effects in programming languages, such as mutable state, divergence, exceptions, or non-determinism. Various frameworks for verifying that programs meet their specification have been proposed, but are all specific to a particular combination of side-effects. For instance, one can use Hoare logic to verify the functional correctness of programs with mutable state with respect to pre/post-conditions specifications.

This thesis devises a principled semantic framework for verifying programs with arbitrary monadic effects in a generic way with respect to such expressive specifications. The starting point are Dijkstra monads, which are monad-like structures that classify effectful computations satisfying a specification drawn from a monad. Dijkstra monads have already proven valuable in practice for verifying effectful code, and in particular, they allow the F^* program verifier to compute verification conditions.

We provide the first semantic investigation of the algebraic structure underlying Dijkstra monads and unveil a close relationship between Dijkstra monads and effect observations, i.e., mappings between a computational and a specification monad that respect their monadic structure. Effect observations are flexible enough to provide various interpretations of effects, for instance total vs partial correctness, or angelic vs demonic nondeterminism. Our semantic investigation relies on a general theory of specification monads and effect observations, using an enriched notion of relative monads and relative monad morphisms. We moreover show that a large variety of specification monads can be obtained by applying monad transformers to various base specification monads, including predicate transformers and Hoare-style pre- and postconditions. For defining correct monad transformers, we design a language inspired by the categorical analysis of the relationship between monad transformers and algebras for a monad.

We also adapt our framework to relational verification, i.e., proving relational properties between multiple runs of one or more programs, such as noninterference or program equivalence. For this we extend specification monads and effect observations to the relational setting and use them to derive the semantics and core rules of a relational program logic generically for any monadic effect. Finally, we identify and overcome conceptual challenges that prevented previous relational program logics from properly dealing with effects such as exceptions, and are the first to provide a proper semantic foundation and a relational program logic for exceptions.