CHALMERS
UNIVERSITY OF TECHNOLOGY

Extensional equality preservation and verified generic programming

Citation for the original published paper (version of record):
Botta, N., Brede, N., Jansson, P. et al (2021). Extensional equality preservation and verified generic programming. Journal of Functional Programming, 31.
http://dx.doi.org/10.1017/S0956796821000204

JFP **31**, e24, 24 pages, 2021.

doi:10.1017/S0956796821000204

Extensional equality preservation and verified generic programming

NICOLA BOTTA
Potsdam Institute for Climate Impact Research, Potsdam, Germany, Chalmers University of Technology, Göteborg, Sweden
(*e-mail:* botta@pik-potsdam.de)

NURIA BREDE
Potsdam Institute for Climate Impact Research, Potsdam, Germany
(*e-mail:* nubrede@pik-potsdam.de)

PATRIK JANSSON
Chalmers University of Technology and University of Gothenburg, Göteborg, Sweden
(*e-mail:* patrikj@chalmers.se)

TIM RICHTER
Potsdam University, Potsdam, Germany
(*e-mail:* tim.richter@uni-potsdam.de)

Abstract

In verified generic programming, one cannot exploit the structure of concrete data types but has to rely on *well chosen* sets of specifications or abstract data types (ADTs). Functors and monads are at the core of many applications of functional programming. This raises the question of what useful ADTs for verified functors and monads could look like. The functorial map of many important monads preserves extensional equality. For instance, if $f, g : A \to B$ are extensionally equal, that is, $\forall x \in A,\ f\,x = g\,x$, then $map\,f : List\,A \to List\,B$ and $map\,g$ are also extensionally equal. This suggests that *preservation of extensional equality* could be a useful principle in verified generic programming. We explore this possibility with a minimalist approach: we deal with (the lack of) extensional equality in Martin-Löf's intensional type theories without extending the theories or using full-fledged setoids. Perhaps surprisingly, this minimal approach turns out to be extremely useful. It allows one to derive simple generic proofs of monadic laws but also verified, generic results in dynamical systems and control theory. In turn, these results avoid tedious code duplication and ad-hoc proofs. Thus, our work is a contribution toward pragmatic, verified generic programming.

1 Introduction

This paper is about *extensional equality preservation* in dependently typed languages like Idris (Brady, 2017), Agda (Norell, 2007) and Coq (The Coq Development Team, 2021) that implement Martin-Löf's intensional type theory (Martin-Löf & Sambin, 1984). We discuss Idris code, but the results could be translated to other languages. Extensional equality is a property of functions, stating that they are "pointwise equal":

$$(\doteq) : \{A, B : Type\} \to (A \to B) \to (A \to B) \to Type$$
$$(\doteq)\,\{A\}\,f\,g = (x : A) \to f\,x = g\,x$$

Note that the definition of extensional equality ($\doteq$) depends on another equality ($=$).

Different flavours of equality. *"All animals are equal, but some animals are more equal than others" [Animal Farm, Orwell (1946)]*

There are several kinds of "equality" relevant for programming. Programming languages usually offer a Boolean equality check operator and in Idris it is written (==), has type $\{A : Type\} \to Eq\,A \Rightarrow A \to A \to Bool$ and is packaged in the interface *Eq*. This is an "ad-hoc" equality, computing whatever the programmer supplies as an implementation. This paper is not about value level Boolean equality.

On the type level, the dependently typed languages we consider in this paper provide a notion of *intensional equality*, also referred to as an "equality type", which is an inductively defined family of types, usually written infix: $(a = b) : Type$ for $a : A$ and $b : B$. It has just one constructor $Refl : a = a$. The resulting notion is not as boring as it may look at first. We have $Refl : a = b$ not only if a and b are identical but also if they *reduce* to identical expressions. Builtin reduction rules normally include alpha-conversion (capture-free renaming of bound variables), beta-reduction (using substitution) and eta-reduction: $f = \lambda x \Rightarrow f\,x$. So, for example, we have $Refl : id\,x = x$. Furthermore, user-defined equations are also used for reduction. A typical example is addition of natural numbers: with $+$ defined by pattern matching on the first argument, we have, e.g., $Refl : 1 + 1 = 2$. However, while for a variable $n : \mathbb{N}$, we have $Refl : 0 + n = n$, we do not have $Refl : n + 0 = n$.

One very useful property of intensional equality is that it is a congruence with respect to any function. In other words, all functions preserve intensional equality. The proof uses pattern matching, which is straightforward here because *Refl* is the only constructor:

$$cong : \{A, B : Type\} \to \{f : A \to B\} \to \{a, a' : A\} \to a = a' \to f\,a = f\,a'$$
$$cong\ Refl = Refl$$

In a similar way, one can prove that $(=)$ is an equivalence relation: reflexivity is directly implemented by *Refl*, while symmetry and transitivity can be proven by pattern matching.

Extensional equality. As one would expect, extensional equality is an equivalence relation

$$reflEE : \{A, B : Type\} \to \{f : A \to B\} \to f \doteq f$$
$$symEE : \{A, B : Type\} \to \{f, g : A \to B\} \to f \doteq g \to g \doteq f$$
$$transEE : \{A, B : Type\} \to \{f, g, h : A \to B\} \to f \doteq g \to g \doteq h \to f \doteq h$$
$$reflEE = \lambda x \Rightarrow Refl$$
$$symEE\,p = \lambda x \Rightarrow sym\,(p\,x)$$
$$transEE\,p\,q = \lambda x \Rightarrow trans\,(p\,x)\,(q\,x)$$

In general, we can lift any (type-valued) binary relation on a type B to a binary relation on function types with co-domain B.

$$extify : \{A, B : Type\} \to (B \to B \to Type) \to ((A \to B) \to (A \to B) \to Type)$$
$$extify\,\{A\}\,relB\,f\,g = (a : A) \to relB\,(f\,a)\,(g\,a)$$

The *extify* combinator maps equivalence relations to equivalence relations. Using it we can redefine $(\doteq) = extify\,(=)$ and we can easily continue to quantify over more arguments:

$(\ddot{=}) = extify\,(\doteq)$, etc. In this paper, our main focus is equality on functions, and we will explore in some detail the relationship between $f = g$ and $f \doteq g$.

In Martin-Löf's intensional type theory, and thus in Idris, extensional equality is strictly weaker than intensional equality. More concretely, we can implement

$$IEqImplEE : \{A, B : Type\} \to (f, g : A \to B) \to f = g \to f \doteq g$$
$$IEqImplEE\ f\ f\ Refl = \lambda x \Rightarrow Refl$$

but not the converse, normally referred to as *function extensionality*:

$$EEqImplIE : \{A, B : Type\} \to (f, g : A \to B) \to f \doteq g \to f = g \quad \text{-- not implementable}$$

When working with functions, extensional equality is often the notion of interest and libraries of formalized mathematics typically provide definitions like $\doteq$ and basic results like *IEqImplEE*. See, for example, *homot* and *eqtohomot* in Part A of the UniMath library (Voevodsky *et al.*, 2021) or *eqfun* in Coq (The Coq Development Team, 2021).

In reasoning about generic programs in the style of the Algebra of Programming (Bird & de Moor, 1997; Mu *et al.*, 2009) and, more generally, in pen and paper proofs, the principle of function extensionality is often taken for granted.

EE preservation. Preservation of extensional equality is a property of higher order functions: we say that, for fixed, non-function types A, B, C and D, a function $h : (A \to B) \to (C \to D)$ preserves extensional equality (in one argument) if $f \doteq g$ implies $h\,f \doteq h\,g$.

Higher order functions are a distinguished trait of functional programming languages (Bird, 2014), and many well-known function combinators can be shown to preserve extensional equality. In particular, the arrow-function *map* for *Identity*, *List*, *Maybe* and for many other polymorphic data types preserves extensional equality. The standard libraries of Agda and Coq provide several instances[1] [2].

Similarly, if h takes two function arguments, it preserves extensional equality (in two arguments) if $f_1 \doteq g_1$ and $f_2 \doteq g_2$ implies $h\,f_1\,f_2 \doteq h\,g_1\,g_2$, etc. To illustrate the Idris notation for equational reasoning, we show the lemma *compPresEE* proving that function composition satisfies the two-argument version of extensional equality preservation:

$$\begin{aligned}
&compPresEE : \{A, B, C : Type\} \to \{g, g' : B \to C\} \to \{f, f' : A \to B\} \to \\
&\qquad g \doteq g' \to f \doteq f' \to g \circ f \doteq g' \circ f' \\
&compPresEE\ \{g\}\ \{g'\}\ \{f\}\ \{f'\}\ gExtEq\ fExtEq\ x = \\
&\quad ((g \circ f)\ x\ ) =\{\ Refl\ \}= \\
&\quad (g\ (f\ x)\ \ ) =\{\ cong\ (fExtEq\ x)\ \}= \\
&\quad (g\ (f'\ x)\ ) =\{\ gExtEq\ (f'\ x)\ \}= \\
&\quad (g'\ (f'\ x)\ ) =\{\ Refl\ \}= \\
&\quad ((g' \circ f')\ x)\ QED
\end{aligned}$$

The right-hand side is a chain of equal expressions connected by the $=\{\ \text{proofs}\ \}=$ of the individual steps within special braces and ending in *QED*, see "Preorder reasoning" in the documentation by The Idris Community (2020). The steps with *Refl* are just for human readability, and they could be omitted as far as Idris is concerned.

Note that the proof steps are at the level of intensional equality which all functions preserve as witnessed by *cong*. So one can often use *cong* in steps where an outer context

[1] e.g., Agda for *Maybe*: https://agda.github.io/agda-stdlib/Data.Maybe.Properties.html
[2] e.g., Coq for *List*: https://coq.inria.fr/distrib/current/stdlib/Coq.Lists.List.html

is unchanged (like g in this example). A special case of a two-argument version of *cong* shows that composition (like all functions) preserves intensional equality:

$$\begin{aligned} &compPresIE : \{A, B, C : Type\} \to \{g, g' : B \to C\} \to \{f, f' : A \to B\} \to \\ &\qquad g = g' \to f = f' \to g \circ f = g' \circ f' \\ &compPresIE\ Refl\ Refl = Refl \end{aligned}$$

Note that the "strengths" of the two equality preservation lemmas are not comparable: *compPresIE* proves a stronger conclusion, but from stronger assumptions.

ADTs and equality preservation. Abstract data types are often specified (e.g., via Idris *interfaces* or Agda *records*) in terms of higher order functions. Typical examples are, beside the already mentioned *map* for functors, bind and Kleisli composition (see Section 2) for monads. This paper is also about ADTs and generic programming. More specifically, we show how to exploit the notion of extensional equality preservation to inform the design of ADTs for generic programming and embedded domain-specific languages (DSLs). This is exemplified in Sections 3 and 4 with ADTs for functors and monads, but we conjecture that other abstract data types, e.g., for applicatives and arrows, could also profit from a design informed by the notion of preservation of extensional equality.

Thus, our work can also be seen as a contribution to the discussion on verified ADTs initiated by Nicholas Drozd on idris-lang. A caveat is perhaps in place: the discussion on ADTs for functors and monads in Sections 3 and 4 is not meant to answer the question of "what verified interfaces should look like". Our aim is to demonstrate that, like preservation of identity functions or preservation of composition, preservation of extensional equality is a useful principle for ADT design.

What this paper is not about. Before turning to a first example, let us spend a few words on what this paper is *not* about. It is not intended as a contribution to the theoretical study of the equality type in intensional type theory or the algorithmic content of the function extensionality principle.

The equality type in intensional type theory and the question of how to deal with extensional concepts in this context has been the subject of important research for the last thirty years. Since Hofmann's seminal work (Hofmann, 1995), setoids have been the established, but also often dreaded (who coined the expression *"setoid hell"*?) means to deal with extensional concepts in intensional type theory (see also Section 6). Eventually, the study of Martin-Löf's equality type has lead to the development of Homotopy Type Theory and Voevodsky's Univalent Foundations program (Streicher, 1991; Hofmann & Streicher, 1994; The Univalent Foundations Program, 2013). Univalence and recent developments in *Cubical Type Theory* (Cohen *et al.*, 2018) promise to finally provide developers with a computational version of function extensionality.

This paper is a contribution toward *pragmatic* verified generic programming. From this perspective, it might become obsolete when fully computational notions of function extensionality will become available in mainstream programming. From a more mathematical perspective, there are good reasons not to rely on axioms that are stronger than necessary: there are interesting models of type theory that refute function extensionality (Streicher, 1993; von Glehn, 2015; Boulier *et al.*, 2017), and our results can be interpreted in these models.

The paper has been generated from literate Idris files. These can be type checked with Idris 1.3.2 and are available at https://gitlab.pik-potsdam.de/botta/papers.

2 Equality examples from dynamical systems theory

In dynamical systems theory (Kuznetsov, 1998; Thomas & Arnol'd, 2012), a prominent notion is that of the flow (or iteration) of a system. We start by discussing deterministic systems and then generalize to the monadic case. A *deterministic* dynamical system on a set X is an endofunction on X. The set X is often called the *state space* of the system.

Given a deterministic system $f : X \to X$, its nth iterate or flow, is typically denoted by $f^n : X \to X$ and is defined by induction on n: the base case $f^0 = id$ is the identity function and f^{n+1} is defined to be either $f \circ f^n$ or $f^n \circ f$. The two definitions are mathematically equivalent because of associativity of function composition but what can one prove about $f \circ f^n$ and $f^n \circ f$ in intensional type theory? We define the two variants as *flowL* and *flowR*:

```
flowL : {X : Type} -> (X -> X) -> Nat -> (X -> X)
flowL f Z     = id
flowL f (S n) = flowL f n . f
flowR : {X : Type} -> (X -> X) -> Nat -> (X -> X)
flowR f Z     = id
flowR f (S n) = f . flowR f n
```

The flows *flowL f n* and *flowR f n* are intensionally equal:

```
flowLemma : {X : Type} -> (f : X -> X) -> (n : Nat) -> flowL f n = flowR f n
```

With *compPresIE* from Section 1, one can implement *flowLemma* by induction on the number of iterations n. The base case is trivial

```
flowLemma f Z = Refl
```

For readability, we spell out the proof sketch for the induction step in full:

```
flowLemma f (S n) = (flowL f (S n))  ={ Refl }=
                    (flowL f n . f)  ={ compPresIE (flowLemma f n) Refl }=
                    (flowR f n . f)  ={ flowRLemma f n }=
                    (f . flowR f n)  ={ Refl }=
                    (flowR f (S n)) QED
```

First, we apply the definition of *flowL* to deduce *flowL f (S n) = flowL f n . f*. Next, we apply *compPresIE* with the induction hypothesis *flowLemma f n* and deduce *flowL f n . f = flowR f n . f*. The (almost) final step is to show that *flowR f n . f = f . flowR f n*. This is obtained via the auxiliary *flowRLemma* where we use associativity and preservation of intensional equality again.

```
flowRLemma : {X : Type} -> (f : X -> X) -> (n : Nat) -> flowR f n . f = f . flowR f n
flowRLemma f Z     = Refl
flowRLemma f (S n) = (flowR f (S n) . f)   ={ Refl }=
                     ((f . flowR f n) . f) ={ compAssociative f (flowR f n) f }=
                     (f . (flowR f n . f)) ={ compPresIE Refl (flowRLemma f n) }=
                     (f . (f . flowR f n)) ={ Refl }=
                     (f . flowR f (S n))   QED
```

Let us summarize: we have considered the special case of deterministic dynamical systems, defined the flow (a higher order function) in two different ways and shown that the two definitions are equivalent in the sense that $e_1 = \mathit{flowR}\, f\ n$ and $e_2 = \mathit{flowL}\, f\ n$ are intensionally equal for all f and n. Before we move on to a more general setting, where intensional equality does not hold, let us expand a bit on the different levels of equality relevant for these two functions. The two expressions e_1 and e_2 denote functions of type $X \to X$ and thus for any $x : X$ we also have $e_1\, x = e_2\, x$. On the other hand, the quantification over all n can be absorbed into the definition of extensional equality so that we have $\mathit{flowR}\, f \doteq \mathit{flowL}\, f$ for all f. And with the two-argument version of extensional equality we get $\mathit{flowR} \doteqdot \mathit{flowL}$.

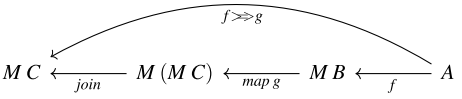
Monadic systems. What about nondeterministic systems, stochastic systems, or perhaps fuzzy systems? Can we extend our results to the general case of *monadic* dynamical systems? Monadic dynamical systems (Ionescu, 2009) on a set X are functions of type $X \to M\, X$ where M is a monad. When M is the identity monad, one recovers the deterministic case. When M is *List* one has nondeterministic systems, and finite probability monads capture the notion of stochastic systems.

One can extend the flow (and, as we will see in Section 5, other elementary operations) of deterministic systems to the general, monadic case by replacing *id* with *pure* and function composition with Kleisli composition ($\gg\!=\!>$):

$$
\begin{aligned}
&\mathit{flowMonL} : \{X : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}\, M \Rightarrow \\
&\qquad (X \to M\, X) \to \mathbb{N} \to (X \to M\, X) \\
&\mathit{flowMonL}\ f\ Z \quad = \mathit{pure} \\
&\mathit{flowMonL}\ f\ (S\, n) = \mathit{flowMonL}\, f\ n \gg\!=\!> f \\
&\mathit{flowMonR} : \{X : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}\, M \Rightarrow \\
&\qquad (X \to M\, X) \to \mathbb{N} \to (X \to M\, X) \\
&\mathit{flowMonR}\ f\ Z \quad = \mathit{pure} \\
&\mathit{flowMonR}\ f\ (S\, n) = f \gg\!=\!> \mathit{flowMonR}\, f\ n
\end{aligned}
$$

Notice, however, that now the implementations of *flowMonL* and *flowMonR* depend on ($\gg\!=\!>$), which is a monad-specific operation. This means that, in proving properties of the flow of monadic systems, we can no longer rely on a specific *definition* of ($\gg\!=\!>$): we have to derive our proofs on the basis of properties that we know (or require) ($\gg\!=\!>$) to fulfil – that is, on its *specification*.

What do we know about Kleisli composition *in general*? We discuss this question in the next two sections but let us anticipate that, if we require functors to preserve the extensional equality of arrows (in addition to identity and composition) and Kleisli composition to fulfil the specification



$$
\begin{aligned}
&\mathit{kleisliSpec} : \{A, B, C : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}\, M \Rightarrow \\
&\qquad (f : A \to M\, B) \to (g : B \to M\, C) \to (f \gg\!=\!> g) \doteq \mathit{join} \circ \mathit{map}\, g \circ f
\end{aligned}
$$

then we can derive preservation of extensional equality

$$
\begin{aligned}
&\mathit{kleisliPresEE} : \{A, B, C : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}\, M \Rightarrow \\
&\qquad (f, f' : A \to M\, B) \to (g, g' : B \to M\, C) \to \\
&\qquad f \doteq f' \to g \doteq g' \to (f \gg\!=\!> g) \doteq (f' \gg\!=\!> g')
\end{aligned}
$$

and associativity of Kleisli composition generically.

$$\begin{aligned}&\mathit{kleisliAssoc} : \{A, B, C, D : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}\, M \Rightarrow \\ &\quad (f : A \to M\, B) \to (g : B \to M\, C) \to (h : C \to M\, D) \to \\ &\quad ((f \gg\!=\!> g) \gg\!=\!> h) \doteq (f \gg\!=\!> (g \gg\!=\!> h))\end{aligned}$$

From these premises, we can prove the *extensional* equality of *flowMonL* and *flowMonR* using a similar lemma as in the deterministic case:

$$\begin{aligned}&\mathit{flowMonRLem} : \{X : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}\, M \Rightarrow \\ &\quad (f : X \to M\, X) \to (n : \mathbb{N}) \to (\mathit{flowMonR}\, f\, n \gg\!=\!> f) \doteq (f \gg\!=\!> \mathit{flowMonR}\, f\, n)\end{aligned}$$

First, notice that the base case of the lemma requires computing evidence that $\mathit{pure} \gg\!=\!> f$ is extensionally equal to $f \gg\!=\!> \mathit{pure}$. This is a consequence of *pure* being a left and a right identity for Kleisli composition: for $f : A \to M\, B$ we have

$$\begin{aligned}&\mathit{pureLeftIdKleisli}\ \ f : (\mathit{pure} \gg\!=\!> f) \doteq f \\ &\mathit{pureRightIdKleisli}\, f : (f \gg\!=\!> \mathit{pure}) \doteq f\end{aligned}$$

$$\begin{aligned}&\mathit{flowMonRLem}\, f\, Z\, x = \\ &\quad ((\mathit{flowMonR}\, f\, Z \gg\!=\!> f)\, x) &&=\{\ \mathit{Refl}\ \}= \\ &\quad ((\mathit{pure} \gg\!=\!> f)\, x) &&=\{\ \mathit{pureLeftIdKleisli}\, f\, x\ \}= \\ &\quad (f\, x) &&=\{\ \mathit{sym}\, (\mathit{pureRightIdKleisli}\, f\, x)\ \}= \\ &\quad ((f \gg\!=\!> \mathit{pure})\, x) &&=\{\ \mathit{Refl}\ \}= \\ &\quad ((f \gg\!=\!> \mathit{flowMonR}\, f\, Z)\, x)\ \mathit{QED}\end{aligned}$$

As we will see in Subsection 4.1, *pureLeftIdKleisli* and *pureRightIdKleisli* are either ADT axioms or theorems, depending of the formulation of the monad ADT. The induction step of *flowMonRLem* relies on preservation of extensional equality and on associativity of Kleisli composition:

$$\begin{aligned}&\mathit{flowMonRLem}\, f\, (S\, n)\, x = \\ &\quad \mathbf{let}\ \mathit{rest} = \mathit{flowMonR}\, f\, n\ \mathbf{in} \\ &\qquad ((\mathit{flowMonR}\, f\, (S\, n) \gg\!=\!> f)\, x) &&=\{\ \mathit{Refl}\ \}= \\ &\qquad (((f \gg\!=\!> \mathit{rest}) \gg\!=\!> f)\, x) &&=\{\ \mathit{kleisliAssoc}\, f\, \mathit{rest}\, f\, x\ \}= \\ &\qquad ((f \gg\!=\!> (\mathit{rest} \gg\!=\!> f))\, x) &&=\{\ \mathit{kleisliPresEE}\, f\, f \qquad (\mathit{rest} \gg\!=\!> f)\, (f \gg\!=\!> \mathit{rest}) \\ &\qquad\qquad &&\qquad\qquad (\lambda v \Rightarrow \mathit{Refl})\, (\mathit{flowMonRLem}\, f\, n)\, x\ \}= \\ &\qquad ((f \gg\!=\!> (f \gg\!=\!> \mathit{rest}))\, x) &&=\{\ \mathit{Refl}\ \}= \\ &\qquad ((f \gg\!=\!> \mathit{flowMonR}\, f\, (S\, n))\, x)\ \mathit{QED}\end{aligned}$$

Finally, the extensional equality of *flowMonL* and *flowMonR*

$$\begin{aligned}&\mathit{flowMonLemma} : \{X : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}\, M \Rightarrow \\ &\quad (f : X \to M\, X) \to (n : \mathbb{N}) \to \mathit{flowMonL}\, f\, n \doteq \mathit{flowMonR}\, f\, n\end{aligned}$$

$$\mathit{flowMonLemma}\, f\, Z\, x = \mathit{Refl}$$

$$\begin{aligned}&\mathit{flowMonLemma}\, f\, (S\, n)\, x = \\ &\quad \mathbf{let}\ \mathit{fLn} = \mathit{flowMonL}\, f\, n \\ &\qquad \mathit{fRn} = \mathit{flowMonR}\, f\, n\ \mathbf{in} \\ &\qquad (\mathit{flowMonL}\, f\, (S\, n)\, x) &&=\{\ \mathit{Refl}\ \}= \\ &\qquad ((\mathit{fLn} \gg\!=\!> f)\, x) &&=\{\ \mathit{kleisliPresEE}\, \mathit{fLn}\, \mathit{fRn} \qquad f\, f \\ &\qquad\qquad &&\qquad\qquad (\mathit{flowMonLemma}\, f\, n)\, (\lambda v \Rightarrow \mathit{Refl})\, x\ \}= \\ &\qquad ((\mathit{fRn} \gg\!=\!> f)\, x) &&=\{\ \mathit{flowMonRLem}\, f\, n\, x\ \}= \\ &\qquad ((f \gg\!=\!> \mathit{fRn})\, x) &&=\{\ \mathit{Refl}\ \}= \\ &\qquad (\mathit{flowMonR}\, f\, (S\, n)\, x)\ \mathit{QED}\end{aligned}$$

follows from *flowMonRLem* and, again, preservation of extensional equality.

Discussion. Before we turn to the next section, let us discuss one objection to what we have just done. Why have we not tried to prove that *flowMonL f n* and *flowMonR f n* are *intensionally* equal as we did for the deterministic flows? If we managed to show the intensional equality of the two flow computations, their extensional equality would follow.

The problem with that approach is that it would require much stronger assumptions: *pureLeftIdKleisli*, *pureRightIdKleisli*, *kleisliAssoc* and *kleisliSpec* would need to hold intensionally. For example, it would require $f >\!\!=\!\!> g$ to be intensionally equal to $join \circ map\ g \circ f$. In Section 4, we will see that, in some abstract data types for monads this is indeed the case, but to require all of these would make our monad interface impossible (or at least very hard) to implement. In general, we cannot rely on $f >\!\!=\!\!> g$ to be intensionally equal to $join \circ map\ g \circ f$.

In designing ADTs and formulating generic results, we have to be careful not to impose too strong proof obligations on implementors. Requiring the monad operations to fulfil intensional equalities would perhaps not be as bad as pretending that function extensionality holds in general, but would still imply unnecessary restrictions. By contrast, requiring proper functors to preserve the extensional equality of arrows is a natural, minimal invasive specification: it allows one to leverage on what *List*, *Maybe*, *Dist* (Erwig & Kollmansberger, 2006), *SimpleProb* (Botta *et al.*, 2017) and many other monads that are relevant for applications are known to fulfil, derive generic verified implementations, avoid boilerplate code, and improve the understandability of proofs.

3 Functors and extensional equality preservation

In category theory, a functor F is a structure-preserving mapping between two categories $\mathcal{C}$ and $\mathcal{D}$. A functor is both a total function from the objects of $\mathcal{C}$ to the objects of $\mathcal{D}$ and a total function from the arrows of $\mathcal{C}$ to the arrows of $\mathcal{D}$ (often both denoted by F) such that for each arrow $f : A \to B$ in $\mathcal{C}$ there is an arrow $F f : F A \to F B$ in $\mathcal{D}$. For an introduction to category theory, see Pierce (1991). The arrow map preserves identity arrows and arrow composition. In formulas:

$$\begin{aligned} F\, id_A &= id_{F A} \\ F\,(g \circ f) &= F\, g \circ F f \end{aligned}$$

Here A denotes an object of $\mathcal{C}$, $F A$ the corresponding object of $\mathcal{D}$ under F, id_A and $id_{F A}$ denote the identity arrows of A and $F A$ in $\mathcal{C}$ and $\mathcal{D}$, respectively and g and f denote arrows between suitable objects in $\mathcal{C}$. In $\mathcal{D}$, $F\, id_A$, $F\,(g \circ f)$, $F\, g$ and $F f$ denote the arrows corresponding to the $\mathcal{C}$-arrows id_A, $g \circ f$, g and f.

Level of formalization. When considering ADT specifications of functor (and of natural transformation, monad, etc.) in dependently typed languages, one has to distinguish between two related but different situations.

One in which the specification is put forward in the context of formalizations of category theory, see, for example (Voevodsky *et al.*, 2021; The Coq Development Team, 2021). In this situation, one has to expect the notion of category to be in place and that of functor to be predicated on that of its source and target categories. A functor ADT in this situation is

an answer to the question "What shall the notion of functor look like in dependently typed formalizations of category theory?"

A different situation is the one in which, in a dependently typed language, we consider the category whose objects are types (in Idris, values of type *Type*), arrows are functions, and functors are of type $Type \to Type$. In this case, a functor ADT is an answer to the question "What does it mean for a value of type $Type \to Type$ to be a functor?" and category theory plays the role of a meta-theory that we use to motivate the specification.

The latter situation is the one considered in this paper. More specifically, we consider the ADTs encoded in the Haskell type classes *Functor* and *Monad* and ask ourselves what are meaningful specifications for these ADTs in dependently typed languages.

Toward an ADT for functors. In Idris, the notion of a functor that preserves identity and composition can be specified as follows (but this is not our final version):

```
interface Functor (F : Type → Type) where
  map         : {A, B : Type}     → (A → B) → F A → F B
  mapPresId   : {A : Type}        → map id ≐ id {a = F A}
  mapPresComp : {A, B, C : Type} → (g : B → C) → (f : A → B) →
                                    map (g ∘ f) ≐ map g ∘ map f
```

In *mapPresId* we have to help the type checker a little bit and give the domain of the two functions that are posited to be extensionally equal explicitly.

Notice that the function *map* is required to preserve identity and composition *extensionally*. In other words, *Functor* does not require *map id* and *id* (*map* $(g \circ f)$ and *map* $g \circ$ *map* f) to be intensionally equal but only to be equal extensionally. This is for very good reasons! If functors were required to preserve identity and composition intensionally, the interface would be hardly implementable. By contrast, *Identity*, *List*, *Maybe*, *Vect n*, and many other important type constructors are functors in the sense specified by this *Functor* interface.

Does this *Functor* interface represent a suitable Idris implementation of the notion of functor in dependently typed languages? We argue that this is not the case and that beside requiring from *map* preservation of identity and of composition, one should additionally require preservation of extensional equality. In other words, we argue that the above specification of *Functor* is incomplete. A more complete specification could look like

```
interface Functor (F : Type → Type) where
  map         : {A, B : Type}     → (A → B) → F A → F B
  mapPresEE   : {A, B : Type}     → (f, g : A → B) → f ≐ g → map f ≐ map g   -- New!
  mapPresId   : {A : Type}        → map id ≐ id {a = F A}
  mapPresComp : {A, B, C : Type} → (g : B → C) → (f : A → B) →
                                    map (g ∘ f) ≐ map g ∘ map f
```

The *Identity* functor, *List*, *Maybe*, *Vect n*, and, more generally, container-like functors built from algebraic datatypes fulfil the complete specification and the proofs for *mapPresEE* do not add significant work. But other prominent functors such as *Reader* do not fulfil the above specification as we will explain below.

Note that it is quite possible to continue on the road toward full generality (supporting a larger class of functors) by parameterizing over the equalities used, but this leads to quite

a bit of book-keeping (basically a setoid-based framework). We instead stop at this point and show that it is a pragmatic compromise between generality and convenient usage.

Equality preservation examples. Let us first have a look at *map* and a proof of *mapPresEE* for *List*, one of the functors that fulfil the above specification:

```
mapList : {A, B : Type} → (A → B) → (List A → List B)
mapList f []        = []
mapList f (a :: as) = f a :: mapList f as
```

Written out in equational reasoning style, the preservation of EE proof looks as follows:

```
mapListPresEE : {A, B : Type} → (f, g : A → B) → f ≐ g → mapList f ≐ mapList g
mapListPresEE f g fEEg []        =  Refl
mapListPresEE f g fEEg (a :: as) =
  (mapList f (a :: as))          ={ Refl }=
  (f a :: mapList f as)          ={ cong {f = (:: mapList f as)} (fEEg a) }=
  (g a :: mapList f as)          ={ cong (mapListPresEE f g fEEg as) }=
  (g a :: mapList g as)          ={ Refl }=
  (mapList g (a :: as))          QED
```

In general the proofs have a very simple structure: they use the $f \doteq g$ arguments at the "leaves," and otherwise only use the induction hypotheses. With a suitable universe of codes for types, or a library for parametricity proofs, these proofs can be automated using datatype-generic programming.

Let's now turn to a type constructor that is not an instance of our *Functor*, namely *Reader E* for some environment E : *Type*.

```
Reader : Type → Type → Type
Reader E A = E → A

mapR : {A, B, E : Type} → (A → B) → (Reader E A → Reader E B)
mapR f r = f ∘ r
```

If we try to implement preservation of extensional equality, we end up with

```
mapRPresEE : {A, B : Type} → (f, g : A → B) → f ≐ g → mapR f ≐ mapR g
mapRPresEE f g fEEg r =
  (mapR f r)  ={ Refl }=
  (f ∘ r)     ={ ?whatnow }=    -- here we need f = g to proceed
  (g ∘ r)     ={ Refl }=
  (mapR g r) QED
```

Notice the question mark in front of **whatnow**. This introduces an unresolved proof step and allows us to ask Idris to help us implementing this step, see "Elaborator Reflection – Holes" in (The Idris Community, 2020). Among other things, we can ask about the type of **whatnow**. Perhaps not surprisingly, this turns out to be $f \circ r = g \circ r$.

The problem is that, although we know that $(f \circ r)\, e = (g \circ r)\, e$ for all $e : E$, we cannot deduce $f \circ r = g \circ r$ without extensionality. Thus, *Reader E* does not implement the *Functor* interface, but it is "very close". Using the 2-argument version of function extensionality $(\doteqdot) = \mathit{extify}\,(\doteq)$, it is easy to show

```
mapRPresEE2 : {E, A, B : Type} → (f, g : A → B) → f ≐ g → mapR f ≐≐ mapR g
mapRPresEE2 f g fEEg r x = fEEg (r x)
```

Thus, *Reader E* is an example of a functor that does not preserve, but rather *transforms* the notion of equality. By a similar argument, *mapPresEE* does not hold for the continuation monad.

As we mentioned earlier, it is tempting to start adding equalities to the interface (toward a setoid-based framework), but this is not a path we take here. As a small hint of the problems that the setoid path leads to, consider that we already have four different objects (A, B, $F\,A$, $F\,B$) and two arrow types ($A \to B$, $F\,A \to F\,B$), all of which could be allowed "their own" notion of equality.

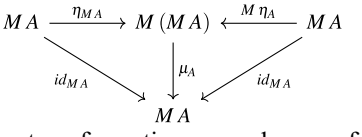
Wrapping up. As stated in Section 1, we argue that, for verified generic programming, it is useful to distinguish between type constructors whose *map* can be shown to preserve extensional equality and type constructors for which this is not the case. A discussion of what are appropriate names for the respective ADTs is beyond the scope of this paper. In the next section, we explore how functors with *mapPresEE* affect the monad ADT design.

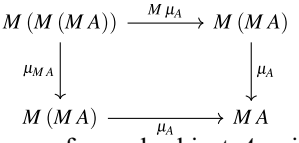
4 Verified monad interfaces

In this section, we review two standard notions of monads. We discuss their mathematical equivalence and consider "thin" and "fat" monad ADT formulations. We discuss the role of extensional equality preservation for deriving monad laws and for verifying the equivalence between the different ADT formulations. There are many possible ways of formulating Monad axioms. Here, we do not argue for or against a specific formulation but rather discuss the most popular ones and their trade-offs.

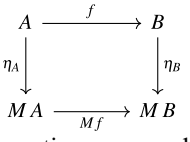
4.1 The traditional view

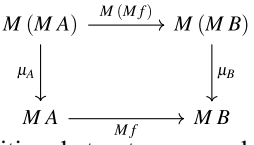
In category theory, a monad is an *endofunctor* M on a category $\mathcal{C}$ together with two *natural transformations* $\eta : Id \dot{\to} M$ (the *unit*) and $\mu : M \circ M \dot{\to} M$ (the *multiplication*) such that, for any object A of $\mathcal{C}$, the following diagrams commute:





The transformations η and μ are families of arrows, one for each object A, with types $\eta_A : A \to M\,A$ and $\mu_A : M\,(M\,A) \to M\,A$. That they are *natural transformations* means that the following diagrams commute for any arrow $f : A \to B$ in $\mathcal{C}$:





From this perspective, a monad is a functor with additional structure, namely families of maps η and μ, satisfying, for any arrow $f : A \to B$, the five properties:

T1. Triangle left: $\mu_A \circ \eta_{M\,A} = id_{M\,A}$
T2. Triangle right: $\mu_A \circ M\,\eta_A = id_{M\,A}$
T3. Square: $\mu_A \circ \mu_{M\,A} = \mu_A \circ M\,\mu_A$
T4. Naturality of η: $M\,f \circ \eta_A = \eta_B \circ f$
T5. Naturality of μ: $M\,f \circ \mu_A = \mu_B \circ M\,(M\,f)$

In functional programming, η is traditionally denoted by *return* or by *pure* and μ is traditionally called *join*. Idris provides language support for interface *refinement*. Thus, we can leverage on the functor ADT *Functor* from Section 3 and define a monad to be a functor with additional methods *pure* and *join* that satisfy the requirements T1–T5:

```
interface Functor M ⇒ Monad₁ (M : Type → Type) where
  pure          : {A : Type}     → A → M A
  join          : {A : Type}     → M (M A) → M A

  triangleLeft  : {A : Type}     → join ∘ pure      ≐ id {a = M A}
  triangleRight : {A : Type}     → join ∘ map pure  ≐ id {a = M A}
  square        : {A : Type}     → join ∘ join      ≐ join ∘ map {A = M (M A)} join

  pureNatTrans  : {A, B : Type} → (f : A → B) → map f ∘ pure ≐ pure ∘ f
  joinNatTrans  : {A, B : Type} → (f : A → B) → map f ∘ join ≐ join ∘ map (map f)
```

Kleisli composition in the traditional view. In Section 2, we have seen that monads are equipped with a (Kleisli) composition $(>=>)$ that we required to fulfil *kleisliSpec*:

```
(>=>)       : {A, B, C : Type} → {M : Type → Type} → Monad₁ M ⇒
              (A → M B) → (B → M C) → (A → M C)
kleisliSpec : {A, B, C : Type} → {M : Type → Type} → Monad₁ M ⇒
              (f : A → M B) → (g : B → M C) → (f >=> g) ≐ join ∘ map g ∘ f
```

One way of implementing $(>=>)$ that satisfies *kleisliSpec* is to *define*

```
f >=> g = join ∘ map g ∘ f
```

The extensional equality between $f >=> g$ and $join \circ map\,g \circ f$ then follows directly:

```
kleisliSpec f g = λx ⇒ Refl
```

The same approach can be followed for implementing bind, another monad combinator similar to Kleisli composition:

```
(>>=) : {A, B : Type} → {M : Type → Type} → Monad₁ M ⇒ M A → (A → M B) → M B
ma >>= f = join (map f ma)
```

Starting from a *thin* monad ADT as in the example above and adding monadic operators that fulfil a specification *by-construction* is a viable approach. It leads to a rich structure entailing monad laws that can be implemented generically. Thus, one can show that *pure* is a left and a right identity of Kleisli composition (we show only one side here)

```
pureLeftIdKleisli : {A, B : Type} → {M : Type → Type} → Monad₁ M ⇒
                    (f : A → M B) → (pure >=> f) ≐ f
pureLeftIdKleisli f a =
  ((pure >=> f) a)          ={ Refl }=
  (join (map f (pure a)))   ={ cong {f = join} (pureNatTrans f a) }=
  (join (pure (f a)))       ={ triangleLeft (f a) }=
  (f a)                     QED
```

and that Kleisli composition is associative as stated in Section 2, almost straightforwardly and without having to invoke the *mapPresEE* axiom of the underlying functor.

$$
\begin{aligned}
&\mathit{kleisliAssoc} : \{A, B, C, D : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}_1\, M \Rightarrow \\
&\qquad (f : A \to M\, B) \to (g : B \to M\, C) \to (h : C \to M\, D) \to \\
&\qquad ((f \gg\!=\!> g) \gg\!=\!> h) \doteq (f \gg\!=\!> (g \gg\!=\!> h)) \\
&\mathit{kleisliAssoc}\, \{M\}\, \{A\}\, \{C\}\, f\, g\, h\, a = \\
&\quad (((f \gg\!=\!> g) \gg\!=\!> h)\, a) && =\{\ \mathit{Refl}\ \}= \\
&\quad ((\mathit{join} \circ \mathit{map}\, h \circ \mathit{join} \circ \mathit{map}\, g \circ f)\, a) && =\{\ \mathit{cong}\, (\mathit{joinNatTrans}\, h\, (\mathit{map}\, g\, (f\ a)))\ \}= \\
&\quad ((\mathit{join} \circ \mathit{join} \circ \mathit{map}\, (\mathit{map}\, h) \circ \mathit{map}\, g \circ f)\, a) && =\{\ \mathit{square}\ \_\ \}= \\
&\quad ((\mathit{join} \circ \mathit{map}\, \mathit{join} \circ \mathit{map}\, (\mathit{map}\, h) \circ \mathit{map}\, g \circ f)\, a) && =\{\ \mathit{cong}\, \{f = \mathit{join} \circ \mathit{map}\, \mathit{join}\} \\
&\qquad\qquad && \qquad (\mathit{sym}\, (\mathit{mapPresComp}\ \_\ \_\ \_))\ \}= \\
&\quad ((\mathit{join} \circ \mathit{map}\, \mathit{join} \circ \mathit{map}\, (\mathit{map}\, h \circ g) \circ f)\, a) && =\{\ \mathit{cong}\, \{f = \mathit{join}\} \\
&\qquad\qquad && \qquad (\mathit{sym}\, (\mathit{mapPresComp}\ \_\ \_\ \_))\ \}= \\
&\quad ((\mathit{join} \circ \mathit{map}\, (\mathit{join} \circ \mathit{map}\, h \circ g) \circ f)\, a) && =\{\ \mathit{Refl}\ \}= \\
&\quad ((f \gg\!=\!> (g \gg\!=\!> h))\, a) && \mathit{QED}
\end{aligned}
$$

Notice that in order to show that Kleisli composition preserves extensional equality, one has to rely on the *mapPresEE* axiom of the underlying functor, as one would expect.

$$
\begin{aligned}
&\mathit{kleisliPresEE} : \{A, B, C : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}_1\, M \Rightarrow \\
&\qquad (f, f' : A \to M\, B) \to (g, g' : B \to M\, C) \to \\
&\qquad f \doteq f' \to g \doteq g' \to (f \gg\!=\!> g) \doteq (f' \gg\!=\!> g') \\
&\mathit{kleisliPresEE}\, f\, f'\, g\, g'\, fE\, gE\, a = \\
&\quad ((f \gg\!=\!> g)\ \ a\ ) && =\{\ \mathit{kleisliLeapfrog}\, f\, g\, a\ \}= \\
&\quad ((\mathit{id} \gg\!=\!> g)\ \ (f\ \ a)) && =\{\ \mathit{cong}\, (fE\, a)\ \}= \\
&\quad ((\mathit{id} \gg\!=\!> g)\ \ (f'\ a)) && =\{\ \mathit{Refl}\ \}= \\
&\quad ((\mathit{join} \circ \mathit{map}\, g)\ (f'\ a)) && =\{\ \mathit{cong}\, (\mathit{mapPresEE}\, g\, g'\, gE\, (f'\ a))\ \}= \\
&\quad ((\mathit{join} \circ \mathit{map}\, g')\, (f'\ a)) && =\{\ \mathit{Refl}\ \}= \\
&\quad ((f' \gg\!=\!> g')\ \ a\ ) && \mathit{QED}
\end{aligned}
$$

In the implementation of *kleisliPresEE*, we have applied the "leapfrogging" rule (compare (Bird, 2014), p. 250):

$$
\begin{aligned}
&\mathit{kleisliLeapfrog} : \{A, B, C : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}_1\, M \Rightarrow \\
&\qquad (f : A \to M\, B) \to (g : B \to M\, C) \to (f \gg\!=\!> g) \doteq (\mathit{id} \gg\!=\!> g) \circ f \\
&\mathit{kleisliLeapfrog}\, f\, g\, a = \mathit{Refl}
\end{aligned}
$$

Fat ADTs. The main advantages of the approach outlined above – a thin ADT and explicit definitions of the monadic combinators – are readability and straightforwardness of proofs: thanks to the intensional equality between $f \gg\!=\!> g$ and $\mathit{join} \circ \mathit{map}\, g \circ f$, we were able to implement many proof steps with just *Refl*.

The strength of thin ADT designs is also their weakness: in many practical cases, one would like to be able to define *join* in terms of bind and not the other way round. In other words, one would like to weaken the requirements on, e.g., *join*, *map* and Kleisli composition and just require that $f \gg\!=\!> g$ and $\mathit{join} \circ \mathit{map}\, g \circ f$ are extensionally equal. If they happen to be intensionally equal for a specific instance, the better.

This suggests that an alternative way of formalizing the traditional notion of monads from category theory could be through a *fat* ADT:

$$
\begin{array}{l}
\textbf{interface}\ \mathit{Functor}\ M \Rightarrow \mathit{Monad}_2\ (M\ :\ \mathit{Type} \to \mathit{Type})\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{pure} & :\ \{A\ :\ \mathit{Type}\} \to A \to M\,A \\
\mathit{join} & :\ \{A\ :\ \mathit{Type}\} \to M\,(M\,A) \to M\,A \\
(\gg\!=) & :\ \{A, B\ :\ \mathit{Type}\} \to M\,A \to (A \to M\,B) \to M\,B \\
(\gg\!>) & :\ \{A, B, C\ :\ \mathit{Type}\} \to (A \to M\,B) \to (B \to M\,C) \to (A \to M\,C) \\
\mathit{bindJoinMapSpec} & :\ \{A, B\ :\ \mathit{Type}\} \to (f\ :\ A \to M\,B) \to (\gg\!= f) \doteq \mathit{join} \circ \mathit{map}\,f \\
\mathit{kleisliJoinMapSpec} & :\ \{A, B, C\ :\ \mathit{Type}\} \to (f\ :\ A \to M\,B) \to \\
 & \qquad (g\ :\ B \to M\,C) \to (f \gg\!> g) \doteq \mathit{join} \circ \mathit{map}\,g \circ f \\
\mathit{triangleLeft} & :\ \{A\ :\ \mathit{Type}\} \to \mathit{join} \circ \mathit{pure} \doteq \mathit{id}\,\{a = M\,A\} \\
\mathit{triangleRight} & :\ \{A\ :\ \mathit{Type}\} \to \mathit{join} \circ \mathit{map}\,\mathit{pure} \doteq \mathit{id}\,\{a = M\,A\} \\
\mathit{square} & :\ \{A\ :\ \mathit{Type}\} \to \mathit{join} \circ \mathit{join} \doteq \mathit{join} \circ \mathit{map}\,\{A = M\,(M\,A)\}\,\mathit{join} \\
\mathit{pureNatTrans} & :\ \{A, B\ :\ \mathit{Type}\} \to (f\ :\ A \to B) \to \mathit{map}\,f \circ \mathit{pure} \doteq \mathit{pure} \circ f \\
\mathit{joinNatTrans} & :\ \{A, B\ :\ \mathit{Type}\} \to (f\ :\ A \to B) \to \mathit{map}\,f \circ \mathit{join} \doteq \mathit{join} \circ \mathit{map}\,(\mathit{map}\,f)
\end{array}
\end{array}
$$

One could go even further and add more combinators (and their axioms) to the ADT, but for the purpose of this discussion the above example will do. (In any case, many of these methods could be filled in by defaults.) What are the implications of having replaced definitions with specifications? A direct implication is that now, in implementing generic proofs of the monad laws, we have to replace some *Refl* steps with suitable specifications. Thus, for instance *pureLeftIdKleisli* becomes

$$
\begin{array}{l}
\mathit{pureLeftIdKleisli}\ :\ \{A, B\ :\ \mathit{Type}\} \to \{M\ :\ \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}_2\,M \Rightarrow \\
\qquad (f\ :\ A \to M\,B) \to (\mathit{pure} \gg\!> f) \doteq f \\
\begin{array}{ll}
\mathit{pureLeftIdKleisli}\,f\,a = ((\mathit{pure} \gg\!> f)\,a) & =\{\ \mathit{kleisliJoinMapSpec}\,\mathit{pure}\,f\,a\ \}= \\
\quad (\mathit{join}\,(\mathit{map}\,f\,(\mathit{pure}\,a))) & =\{\ \mathit{cong}\,\{f = \mathit{join}\}\,(\mathit{pureNatTrans}\,f\,a)\ \}= \\
\quad (\mathit{join}\,(\mathit{pure}\,(f\,a))) & =\{\ \mathit{triangleLeft}\,(f\,a)\ \}= \\
\quad (f\,a) & \mathit{QED}
\end{array}
\end{array}
$$

where we have replaced the first proof step, *Refl*, with *kleisliJoinMapSpec pure f a*. Similar transformations have to be done for *pureRightIdKleisli*, *kleisliAssoc*, etc. However, completing the proof of the monad laws for the fat interface is not just a matter of replacing definitions with specifications. Consider associativity:

$$
\begin{array}{l}
\mathit{kleisliAssoc}\ :\ \{A, B, C, D\ :\ \mathit{Type}\} \to \{M\ :\ \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}_2\,M \Rightarrow \\
\qquad (f\ :\ A \to M\,B) \to (g\ :\ B \to M\,C) \to (h\ :\ C \to M\,D) \to \\
\qquad ((f \gg\!> g) \gg\!> h) \doteq (f \gg\!> (g \gg\!> h)) \\
\mathit{kleisliAssoc}\,\{M\}\,\{A\}\,\{C\}\,f\,g\,h\,a = \\
\begin{array}{ll}
\quad (((f \gg\!> g) \gg\!> h)\,a) & =\{\ \mathit{kleisliJoinMapSpec}\,(f \gg\!> g)\,h\,a\ \}= \\
\quad ((\mathit{join} \circ \mathit{map}\,h \circ (f \gg\!> g))\,a) & =\{\ \mathit{cong}\,\{f = \mathit{join} \circ \mathit{map}\,h\} \\
 & \qquad (\mathit{kleisliJoinMapSpec}\,f\,g\,a)\ \}= \\
\quad ((\mathit{join} \circ \mathit{map}\,h \circ \mathit{join} \circ \mathit{map}\,g \circ f)\,a) & =\{\ \mathit{cong}\,(\mathit{joinNatTrans}\,h\,(\mathit{map}\,g\,(f\,a)))\ \}= \\
\quad ((\mathit{join} \circ \mathit{join} \circ \mathit{map}\,(\mathit{map}\,h) \circ \mathit{map}\,g \circ f)\,a) & =\{\ \mathit{square}\ \_\ \}= \\
\quad ((\mathit{join} \circ \mathit{map}\,\mathit{join} \circ \mathit{map}\,(\mathit{map}\,h) \circ \mathit{map}\,g \circ f)\,a) & =\{\ \mathit{cong}\,\{f = \mathit{join} \circ \mathit{map}\,\mathit{join}\} \\
 & \qquad (\mathit{sym}\,(\mathit{mapPresComp}\ \_\ \_\ \_))\ \}= \\
\quad ((\mathit{join} \circ \mathit{map}\,\mathit{join} \circ \mathit{map}\,(\mathit{map}\,h \circ g) \circ f)\,a) & =\{\ \mathit{cong}\,\{f = \mathit{join}\} \\
 & \qquad (\mathit{sym}\,(\mathit{mapPresComp}\ \_\ \_\ \_))\ \}= \\
\quad ((\mathit{join} \circ \mathit{map}\,(\mathit{join} \circ \mathit{map}\,h \circ g) \circ f)\,a) & =\{\ \mathit{sym}\,(\mathit{kleisliJoinMapSpec}\,f\,(\mathit{join} \circ \mathit{map}\,h \circ g)\,a)\ \}= \\
\quad ((f \gg\!> (\mathit{join} \circ \mathit{map}\,h \circ g))\,a) & \\
\quad =\{\ \mathit{kleisliPresEE}\,f\,f\quad (\mathit{join} \circ \mathit{map}\,h \circ g)\,(g \gg\!> h) & \\
\qquad \mathit{reflEE}\,(\mathit{symEE}\,\{f = g \gg\!> h\}\,\{g = \mathit{join} \circ \mathit{map}\,h \circ g\} & \\
\qquad\quad (\mathit{kleisliJoinMapSpec}\,g\,h))\,a\ \}= & \\
\quad ((f \gg\!> (g \gg\!> h))\,a)\ \mathit{QED} &
\end{array}
\end{array}
$$

Here, in order to deduce $(f \gg\!= (g \gg\!= h))\, a$ from $(f \gg\!= (\mathit{join} \circ \mathit{map}\, h \circ g))\, a$ in the last step of the proof, we had to apply

$$\begin{aligned}&\mathit{kleisliPresEE} : \{A, B, C : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}_2\, M \Rightarrow \\ &\quad (f, f' : A \to M\, B) \to (g, g' : B \to M\, C) \to \\ &\quad f \doteq f' \to g \doteq g' \to (f \gg\!= g) \doteq (f' \gg\!= g')\end{aligned}$$

instead of just *Refl* as in the case of thin interfaces. In other words: the specification *kleisliJoinMapSpec* alone is not strong enough to grant the last step. It allows one to deduce that $\mathit{join} \circ \mathit{map}\, h \circ g$ and $g \gg\!= h$ are extensionally equal. But this is not enough: we need a proof that Kleisli composition preserves extensional equality. This relies on the functorial *map* of M preserving extensional equality, as in the case of thin ADTs.

The moral is that, when the relationships between the monadic operations *pure*, *join* and $(\gg\!=)$ are specified rather then defined, preservation of extensional equality plays a crucial role even in proofs of straightforward properties like associativity. The same situation occurs if we specify bind in terms of *pure* and *join*.

4.2 The Wadler view

A different perspective on monads goes back to Manes (1976) and has been popularized by Wadler (1992): a monad on a category $\mathcal{C}$ can be defined by giving an endofunction M on the objects of $\mathcal{C}$, a family of arrows $\eta_A : A \to M\, A$ (like η above, but not required to be natural), and a "lifting" operation that maps any arrow $f : A \to M\, B$ to an arrow $f^* : M\, A \to M\, B$. The lifting operation is required to satisfy W1–W3 for any objects A, B, C and arrows $f : A \to M\, B$ and $g : B \to M\, C$, see (Streicher, 2003):

W1. $f^* \circ \eta_A = f$
W2. $\eta_A^* = id_{M\,A}$
W3. $g^* \circ f^* = (g^* \circ f)^*$

From tradition to Wadler and back. The two monad definitions can be seen as two views on the same mathematical concept and we would like the corresponding ADT formulations to preserve this relationship.

It turns out that, if (M, η, μ) fulfil the properties T1–T5 of the traditional view, then the object part of M, η, and the lifting operation defined by $f^* = \mu_{M\,B} \circ \mathit{map}\, f$ satisfy W1–W3.

In turn, given M, η and $\cdot^*$ that satisfy W1–W3, one can define $\mathit{map}\, f = (\eta_B \circ f)^*$ and $\mu_A = id^*_{M\,A}$ and prove that (M, map) is a functor, and that T1–T5 are all satisfied.

This economical way to define a monad has become very popular in functional programming, where $\mathit{lift}\, f = f^*$ is usually given in infix form with flipped arguments and called bind: $ma \gg\!= f = f^*\, ma$. This suggests yet another ADT for monads:

$$\begin{aligned}&\textbf{interface}\ \mathit{Monad}_3\ (M : \mathit{Type} \to \mathit{Type})\ \textbf{where} \\ &\quad \mathit{pure} : \{A : \mathit{Type}\} \to A \to M\, A \\ &\quad (\gg\!=) : \{A, B : \mathit{Type}\} \to M\, A \to (A \to M\, B) \to M\, B \\ &\quad \mathit{pureLeftIdBind} : \{A, B : \mathit{Type}\} \to (f : A \to M\, B) \to (\lambda a \Rightarrow \mathit{pure}\, a \gg\!= f) \doteq f \\ &\quad \mathit{pureRightIdBind} : \{A : \mathit{Type}\} \to (\gg\!= \mathit{pure}) \doteq id\, \{a = M\, A\} \\ &\quad \mathit{bindAssoc} : \{A, B, C : \mathit{Type}\} \to (f : A \to M\, B) \to (g : B \to M\, C) \to \\ &\qquad (\lambda ma \Rightarrow (ma \gg\!= f) \gg\!= g) \doteq (\lambda ma \Rightarrow ma \gg\!= (\lambda a \Rightarrow f\, a \gg\!= g))\end{aligned}$$

The three axioms are formulations of the properties of *lift* W1–W3 in terms of bind. We can now *define map*, *join*, and Kleisli composition in terms of bind and *pure*:

$$
\begin{aligned}
&\mathit{map} : \{A, B : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}_3\, M \Rightarrow (A \to B) \to (M\, A \to M\, B)\\
&\mathit{map}\, f\, \mathit{ma} = \mathit{ma} \gg\!= (\mathit{pure} \circ f)\\
&\mathit{join} : \{A : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}_3\, M \Rightarrow M\,(M\, A) \to M\, A\\
&\mathit{join}\, \mathit{mma} = \mathit{mma} \gg\!= \mathit{id}\\
&(>\!=\!>) : \{A, B, C : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}_3\, M \Rightarrow\\
&\qquad (A \to M\, B) \to (B \to M\, C) \to (A \to M\, C)\\
&f >\!=\!> g = \lambda a \Rightarrow f\, a \gg\!= g
\end{aligned}
$$

The obligation is now to prove that *pure* and *join* fulfil the properties T1–T5, for instance, that *pure* is a natural transformation. In much the same way as for formalizations of the traditional view, some of these proofs can be implemented straightforwardly. But in some cases, one runs into trouble. Consider the proof of T2. Triangle right: $\mu_A \circ M\ \eta_A = id_{M\,A}$

$$
\begin{aligned}
&\mathit{triangleRightFromBind} : \{A : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}_3\, M \Rightarrow\\
&\qquad \mathit{join} \circ \mathit{map}\, \mathit{pure} \doteq \mathit{id}\, \{a = M\, A\}\\
&\mathit{triangleRightFromBind}\, \{A\}\, \{M\}\, \mathit{ma} =\\
&\quad (\mathit{join}\, (\mathit{map}\, \mathit{pure}\, \mathit{ma})) && =\{\ \mathit{Refl}\ \}=\\
&\quad ((\mathit{ma} \gg\!= (\mathit{pure} \circ \mathit{pure})) \gg\!= \mathit{id}) && =\{\ \mathit{bindAssoc}\, (\mathit{pure} \circ \mathit{pure})\, \mathit{id}\, \mathit{ma}\ \}=\\
&\quad (\mathit{ma} \gg\!= (\lambda a \Rightarrow \mathit{pure}\, (\mathit{pure}\, a) \gg\!= \mathit{id})) && =\{\ \mathit{Refl}\ \}=\\
&\quad (\mathit{ma} \gg\!= ((\lambda a \Rightarrow \mathit{pure}\, a \gg\!= \mathit{id}) \circ \mathit{pure})) && =\{\ ?\mathbf{whatnow}\ \}=\\
&\quad (\mathit{ma} \gg\!= \mathit{id} \circ \mathit{pure}) && =\{\ \mathit{Refl}\ \}=\\
&\quad (\mathit{ma} \gg\!= \mathit{pure}) && =\{\ \mathit{pureRightIdBind}\, \mathit{ma}\ \}=\\
&\quad (\mathit{ma}) && \mathit{QED}
\end{aligned}
$$

We know that $\lambda a \Rightarrow \mathit{pure}\, a \gg\!= \mathit{id}$ and *id* are *extensionally* equal by *pureLeftIdBind id*. If this equality would hold *intensionally*, we could fill the hole by congruence. But we cannot strengthen $(\doteq)$ to $(=)$ in *pureLeftIdBind*. Instead, we extend our ADT with

$$\mathit{liftPresEE} : \{A, B : \mathit{Type}\} \to (f, g : A \to M\, B) \to f \doteq g \to (\gg\!= f) \doteq (\gg\!= g)$$

and complete the proof by filling in ?**whatnow** by:

$$\mathit{liftPresEE}\, ((\lambda a \Rightarrow \mathit{pure}\, a \gg\!= \mathit{id}) \circ \mathit{pure})\, (\mathit{id} \circ \mathit{pure})\, (\lambda a \Rightarrow \mathit{pureLeftIdBind}\, \mathit{id}\, (\mathit{pure}\, a))\, \mathit{ma}$$

Notice that, in this approach, *map* is defined in terms of *pure* and bind. Thus, we do not have at our disposal the axioms of the functor ADT and thus we cannot leverage *mapPresEE* to *derive liftPresEE* as we have done in the traditional formulation for Kleisli composition.

The moral is that, even if we adopt the Wadler view on monads and a more economical specification, we have to require *lift* to preserve extensional equality if we want our specification to be consistent with the traditional one.

This completes the discussion on different notions of monads and on the role of extensional equality preservation in generic proofs of monad laws. For the rest of the paper, we apply the traditional view on monads and the fat monad interface.

4.3 More monad properties

As we have seen in Section 2 for *flowMonLemma*, extensional equality preservation is crucially needed in inductive proofs. We discuss more examples of applications of the

principle in the context of DSLs for dynamical systems theory in Section 5. In the rest of this section, we prepare by deriving two intermediate properties. The first is the extensional equality between $map\,f \circ join \circ map\,g$ and $join \circ map\,(map\,f \circ g)$:

```
mapJoinLemma : {M : Type → Type} → {A, B, C : Type} → Monad M ⇒
               (f : B → C) → (g : A → M B) →
               (≐) {A = M A} {B = M C} (map f ∘ join ∘ map g) (join ∘ map (map f ∘ g))
mapJoinLemma f g ma =
  (map f (join (map g ma)))              ={ joinNatTrans f (map g ma) }=
  (join (map (map f) (map g ma)))        ={ Refl }=
  (join ((map (map f) ∘ (map g)) ma))    ={ cong (sym (mapPresComp (map f) g ma)) }=
  (join (map (map f ∘ g) ma))            QED
```

The second,

```
mapKleisliLemma : {A, B, C, D : Type} → {M : Type → Type} → Monad M ⇒
                  (f : A → M B) → (g : B → M C) → (h : C → D) →
                  (map h ∘ (f >=> g)) ≐ (f >=> map h ∘ g)
mapKleisliLemma f g h ma =
  ((map h ∘ (f >=> g)) ma)          ={ cong (kleisliJoinMapSpec f g ma) }=
  (map h (join (map g (f ma))))     ={ mapJoinLemma h g (f ma) }=
  (join (map (map h ∘ g) (f ma)))   ={ sym (kleisliJoinMapSpec f (map h ∘ g) ma) }=
  ((f >=> map h ∘ g) ma)            QED
```

can be seen as an associativity law by rewriting it in terms of $(<=<) = flip\,(>=>)$:

```
mapKleisliLemma : {A, B, C, D : Type} → {M : Type → Type} → Monad M ⇒
                  (f : A → M B) → (g : B → M C) → (h : C → D) →
                  (map h ∘ (g <=< f)) ≐ ((map h ∘ g) <=< f)
```

5 Applications in dynamical systems and control theory

In this section, we discuss applications of the principle of preservation of extensional equality to dynamical systems and control theory. We have seen in Section 2 that time discrete deterministic dynamical systems on a set X are functions of type $X \to X$

```
DetSys : Type → Type
DetSys X = X → X
```

and that generalizing this notion to systems with uncertainties leads to monadic systems

```
MonSys : (Type → Type) → Type → Type
MonSys M X = X → M X
```

where M is an *uncertainty* monad: *List*, *Maybe*, *Dist* (Erwig & Kollmansberger, 2006), *SimpleProb* (Ionescu, 2009; Botta *et al.*, 2017), etc. For monadic systems, one can derive a number of general results. One is that every deterministic system can be embedded in a monadic system:

```
embed : {X : Type} → {M : Type → Type} → Monad M ⇒ DetSys X → MonSys M X
embed f = pure ∘ f
```

A more interesting result is that the flow of a monadic system is a monoid morphism from $(\mathbb{N}, (+), 0)$ to $(MonSys\,M\,X, (>=>), pure)$. As discussed in Section 2, $flowMonL \doteq flowMonR$ and here we write just *flow*. The two parts of the monoid morphism proof are

$$\begin{aligned}
&\mathit{flowLemma1} : \{X : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}\, M \Rightarrow \\
&\qquad (f : \mathit{MonSys}\, M\, X) \to \mathit{flow}\, f\, Z \doteq \mathit{pure} \\
&\mathit{flowLemma2} : \{X : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}\, M \Rightarrow \{m, n : \mathbb{N}\} \to \\
&\qquad (f : \mathit{MonSys}\, M\, X) \to \mathit{flow}\, f\, (m + n) \doteq (\mathit{flow}\, f\, m \gg\!= \mathit{flow}\, f\, n)
\end{aligned}$$

Proving *flowLemma1* is immediate (because $\mathit{flow}\, f\, Z = \mathit{pure}$):

$$\mathit{flowLemma1}\, f = \mathit{reflEE}$$

We prove *flowLemma2* by induction on m using the properties from Section 4: *pure* is a left and right identity of Kleisli composition and Kleisli composition is associative. The base case is straightforward

$$\begin{aligned}
&\mathit{flowLemma2}\, \{m = Z\}\, \{n\}\, f\, x = \\
&\quad (\mathit{flow}\, f\, (Z + n)\, x) && =\{\ \mathit{Refl}\ \}= \\
&\quad (\mathit{flow}\, f\, n\, x) && =\{\ \mathit{sym}\, (\mathit{pureLeftIdKleisli}\, (\mathit{flow}\, f\, n)\, x)\ \}= \\
&\quad ((\mathit{pure} \gg\!= \mathit{flow}\, f\, n)\, x) && =\{\ \mathit{Refl}\ \}= \\
&\quad ((\mathit{flow}\, f\, Z \gg\!= \mathit{flow}\, f\, n)\, x)\ \mathit{QED}
\end{aligned}$$

but the induction step again relies on Kleisli composition preserving extensional equality.

$$\begin{aligned}
&\mathit{flowLemma2}\, f\, \{m = S\, l\}\, \{n\}\, x = \\
&\quad (\mathit{flow}\, f\, (S\, l + n)\, x) && =\{\ \mathit{Refl}\ \}= \\
&\quad ((f \gg\!= \mathit{flow}\, f\, (l + n))\, x) && =\{\ \mathit{kleisliPresEE}\, f\, f \quad (\mathit{flow}\, f\, (l + n))\, (\mathit{flow}\, f\, l \gg\!= \mathit{flow}\, f\, n) \\
& && \qquad \mathit{reflEE}\, (\mathit{flowLemma2}\, f)\, x\ \}= \\
&\quad ((f \gg\!= (\mathit{flow}\, f\, l \gg\!= \mathit{flow}\, f\, n))\, x) && =\{\ \mathit{sym}\, (\mathit{kleisliAssoc}\, f\, (\mathit{flow}\, f\, l)\, (\mathit{flow}\, f\, n)\, x)\ \}= \\
&\quad (((f \gg\!= \mathit{flow}\, f\, l) \gg\!= \mathit{flow}\, f\, n)\, x) && =\{\ \mathit{Refl}\ \}= \\
&\quad ((\mathit{flow}\, f\, (S\, l) \gg\!= \mathit{flow}\, f\, n)\, x) && \mathit{QED}
\end{aligned}$$

As seen in Section 4, this follows directly from the monad ADT and from the preservation of extensional equality for functors.

A representation theorem. Another important result for monadic systems is a representation theorem: any monadic system $f : \mathit{MonSys}\, M\, X$ can be represented by a deterministic system on $M\, X$. With

$$\begin{aligned}
&\mathit{repr} : \{X : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}\, M \Rightarrow \mathit{MonSys}\, M\, X \to \mathit{DetSys}\, (M\, X) \\
&\mathit{repr}\, f = \mathit{id} \gg\!= f
\end{aligned}$$

and for an arbitrary monadic system f, $\mathit{repr}\, f$ is equivalent to f in the sense that

$$\begin{aligned}
&\mathit{reprLemma} : \{X : \mathit{Type}\} \to \{M : \mathit{Type} \to \mathit{Type}\} \to \mathit{Monad}\, M \Rightarrow \\
&\qquad (f : \mathit{MonSys}\, M\, X) \to (n : \mathbb{N}) \to \mathit{repr}\, (\mathit{flow}\, f\, n) \doteq \mathit{flowDet}\, (\mathit{repr}\, f)\, n
\end{aligned}$$

where *flowDet* is the flow of a deterministic system

$$\begin{aligned}
&\mathit{flowDet} : \{X : \mathit{Type}\} \to \mathit{DetSys}\, X \to \mathbb{N} \to \mathit{DetSys}\, X \\
&\mathit{flowDet}\, f\ \ Z\ \ = \mathit{id} \\
&\mathit{flowDet}\, f\, (S\, n) = \mathit{flowDet}\, f\, n \circ f
\end{aligned}$$

As for *flowLemma2*, proving the representation lemma is straightforward but crucially relies on associativity of Kleisli composition and thus, as seen in section 4, on preservation of extensional equality:

```
reprLemma f Z     mx = pureRightIdKleisli id mx
reprLemma f (S m) mx =
  (repr (flow f (S m)) mx)                 =={ Refl }==
  ((id >=> flow f (S m)) mx)               =={ Refl }==
  ((id >=> (f >=> flow f m)) mx)           =={ sym (kleisliAssoc id f (flow f m) mx) }==
  (((id >=> f) >=> flow f m) mx)           =={ kleisliLeapfrog (id >=> f) (flow f m) mx }==
  ((id >=> flow f m) ((id >=> f) mx))      =={ Refl }==
  (repr (flow f m) ((id >=> f) mx))        =={ reprLemma f m ((id >=> f) mx) }==
  (flowDet (repr f) m ((id >=> f) mx))     =={ Refl }==
  (flowDet (repr f) m (repr f mx))         =={ Refl }==
  (flowDet (repr f) (S m) mx)              QED
```

Notice also the application of *kleisliLeapfrog* to deduce $(id \gg\!= flow\, f\ m)\ ((id \gg\!= f)\ mx)$ from $((id \gg\!= f) \gg\!= flow\, f\ m)\ mx$. If we had formulated the theory in terms of bind instead of Kleisli composition, the two expressions would have been intensionally equal.

Flows and trajectories. Our last application of preservation of extensional equality in the context of dynamical systems theory is a result about flows and trajectories. For a monadic system f, the trajectories of length $n+1$ starting at state $x : X$ are

```
trj : {M : Type → Type} → {X : Type} → Monad M ⇒
      MonSys M X → (n : ℕ) → X → M (Vect (S n) X)
trj f Z     x = map (x::) (pure Nil)
trj f (S n) x = map (x::) ((f >=> trj f n) x)
```

In words, the trajectory obtained by making zero steps starting at x is an M-structure containing just $[x]$. To compute the trajectories for $S\,n$ steps, we first bind the outcome of a single step $f\,x : M\,X$ into $trj\,f\,n$. This results in an M-structure of vectors of length n. Finally, we prepend these possible trajectories with the initial state x.

Since $trj\,f\,n\,x$ is an M-structure of vectors of states, we can compute the last state of each trajectory. It turns out that this is point-wise equal to $flow\,f\,n$:

```
flowTrjLemma : {X : Type} → {M : Type → Type} → Monad M ⇒
               (f : MonSys M X) → (n : ℕ) →
               flow f n ≐ map {A = Vect (S n) X} last ∘ trj f n
```

To prove this result, we first derive the auxiliary lemma

```
mapLastLemma : {F : Type → Type} → {X : Type} → {n : ℕ} → Functor F ⇒
               (x : X) → (mvx : F (Vect (S n) X)) →
               (map last ∘ map (x::)) mvx = map last mvx
mapLastLemma {X} {n} x mvx =
  (map {A = Vect (S (S n)) X} last (map (x::) mvx))
    ={ sym (mapPresComp {A = Vect (S n) X} last (x::) mvx) }=
  (map (last ∘ (x::)) mvx)
    ={ mapPresEE (last ∘ (x::)) last (lastLemma x) mvx }=
  (map last mvx) QED
```

where $lastLemma\,x : last \circ (x::) \doteq last$.

In the implementation of *mapLastLemma*, we have applied both preservation of composition and preservation of extensional equality. With *mapLastLemma* in place, *flowTrjLemma* is readily implemented by induction on the number of steps

```
flowTrjLemma {X} f Z x =
  (flow f Z x) ={ Refl }= (pure x)          ={ Refl }=
  (pure (last (x :: Nil)))                  ={ sym (pureNatTrans last (x :: Nil)) }=
  (map last (pure (x :: Nil)))              ={ cong {f = map last}
                                               (sym (pureNatTrans {A = Vect Z X} (x::) Nil)) }=
  (map last (map (x::) (pure Nil)))         ={ Refl }=
  (map last (trj f Z x))                    QED
flowTrjLemma f (S m) x =
  (flow f (S m) x)                                    ={ Refl }=
  ((f >=> flow f m) x) ={ kleisliPresEE f f (flow f m) (map (last {len = m}) . trj f m)
                                        reflEE (flowTrjLemma f m) x }=
  ((f >=> map (last {len = m}) . trj f m) x) ={ sym (mapKleisliLemma f (trj f m) last x) }=
  (map last ((f >=> trj f m) x))             ={ sym (mapLastLemma x ((f >=> trj f m) x)) }=
  (map last (map (x::) ((f >=> trj f m) x))) ={ Refl }=
  (map last (trj f (S m) x)) QED
```

Again, preservation of extensional equality proves essential for the induction step.

Dynamic programming (DP). The relationship between the flow and the trajectory of a monadic dynamical system also plays a crucial role in the *semantic verification* of dynamic programming. DP (Bellman, 1957) is a method for solving sequential decision problems. These problems are at the core of many applications in economics, logistics, and computer science and are, in principle, well understood (Bellman, 1957; De Moor, 1995; Gnesi *et al.*, 1981; Botta *et al.*, 2017).

Proving that dynamic programming is semantically correct boils down to showing that the value function *val* that is at the core of the backward induction algorithm of DP is extensionally equal to a specification val'.

The *val* function of DP takes n policies or decision rules and is computed by iterating n times a monadic dynamical system similar to the function argument of *flow* but with an additional *control* argument. At each iteration, a *reward* function is mapped on the states and the result is reduced with a *measure* function. In this computation, the measure function is applied a number of times that is exponential in n.

By contrast, val' is computed by applying the measure function only once, but to a structure of a size exponential in n that is obtained by adding up the rewards along all the trajectories.

The equivalence between *val* and val' is established by structural induction. As in the *flowTrjLemma* discussed above, *map* preserving extensional equality turns out to be pivotal in applying the induction hypothesis, see Brede & Botta (2021) for details.

6 Related work

As already mentioned in Section 1, there is a large body of literature that relates in some form to (the treatment of) equality in intensional type theory. Most of that work, however, is concerned with the theoretical study of the Martin-Löf identity type or with the implementation of variants of type theory and thus very different in nature from the present paper which takes a pragmatic user-level approach.

Closest to our approach from the theoretical point of view are perhaps works on formalization in type theory using *setoids*. These were originally introduced by Bishop (Bishop, 1967) for his development of constructive mathematics, and studied in (Hofmann, 1995) for the treatment of weaker notions of equality in intensional type theory. Setoids are sets equipped with an equivalence relation and mappings between setoids have to take equivalent arguments to equivalent results. The focus of our paper can thus be seen as one special case with extensional equality of functions as the equivalence relation of interest and thus its preservation as coherence condition on mappings. The price to pay when using a full-fledged setoid approach is the presence of a potentially huge amount of additional proof obligations, needed to coherently deal with sets (types) and their equivalence relations – this often is pointedly referred to as *setoid hell* (for instance in Altenkirch (2017), but it seems to have been used colloquially in the community for much longer).

Still, there are some large developments using setoids, e.g., the CoRN library (formalizing constructive mathematics) by Spitters & Semeria (2017) and the CoLoR library (for rewriting and termination) by Blanqui *et al.* (2020) in Coq where the proof assistant provides the user with some convenient tools for dealing with setoids (Sozeau, 2010). Setoids are also used in a number of formalizations of category theory, e.g. (Huet & Saïbi, 2000; Megacz, 2011; Wiegley, 2018; Carette & Hu, 2021; Hu & Carette, 2021).

Homotopy Type Theory with *univalence* (The Univalent Foundations Program, 2013) provides function extensionality as a by-product. However, in most languages (notably in Coq in which the Univalent Foundations library (Voevodsky *et al.*, 2021, UniMath) is developed), univalence is still an axiom and thus blocks computation. Moreover, univalence is incompatible with the principle of *Uniqueness of Identity Proofs* which, e.g., in Idris is built in, and in Agda has to be disabled using a special flag.

Finally, in *Cubical Type Theory* (Cohen *et al.*, 2018), function extensionality is provable because of the presence of the *interval primitive* and thus has computational content. Cubical type theory has recently been implemented as a special version of Agda (Vezzosi *et al.*, 2021). Another (similar) version of homotopy type theory is implemented in the theorem prover Arend (JetBrains Research, 2021). However, it is not clear at the present stage how long it will take for these advances in type theory to become available in mainstream functional programming.

On the topic of interfaces (type classes) and their laws there is related work in specifying (Jansson & Jeuring, 2002), rewriting (Peyton Jones *et al.*, 2001), testing (Jeuring *et al.*, 2012), and proving (Arvidsson *et al.*, 2019) type class laws in Haskell. The equality challenges here are often related to the semantics of nontermination as described in the Fast and Loose Reasoning paper (Danielsson *et al.*, 2006). In a dependently typed setting, there is related work on contrasting the power of testing and proving, including Agda code for the Functor interface with extensional equality for the identity and composition preservation but not preservation of extensional equality (Ionescu & Jansson, 2013).

Carette *et al.* (2014) nicely abstract the ideas about different (minimal) interfaces that we only exemplified using verified monads. Regarding the relation between different representations of monads, the reader might contrast the approach in the UniMath (Voevodsky *et al.*, 2021, `CategoryTheory.Monads.KTriplesEquiv`) library with our approach in Section 4. The UniMath development is part of a full-fledged formalization of category theory and relates "the traditional view" with the "Wadler view" of a monad (as we called

them) via a weak equivalence of categories. This approach is very satisfactory from an abstract mathematical perspective. Our equivalence result is much less general but still practically relevant and more lightweight: although it requires considerations about preservation of extensional equality, it does not require stronger axioms like univalence or a larger conceptual framework.

7 Conclusions, outlook

In dependently typed programming in the context of Martin-Löf type theories (Martin-Löf & Sambin, 1984; Nordström *et al.*, 1990), the problem of how to specify abstract data types for verified generic programming is still not well understood. In this work, we have shown that requiring functors to preserve extensional equality of arrows yields abstract data types that are strong enough to support the verification of nontrivial monadic laws and of generic results in domain specific languages for dynamical system and control theory.

We have shown that such a minimalist approach can be exploited to derive results that otherwise would require enforcing the relationships between monadic operators – *pure*, bind, join, Kleisli composition, etc. – through intensional equalities or, even worse, postulating function extensionality or similar *impossible* specifications.

As a consequence, we have proposed to carefully distinguish between functors whose associated *map* can be shown to preserve extensional equality (and identity arrows and arrow composition) and functors for which this is not the case.

Current work by two of the authors shows that preservation of extensional equality is useful in designing a verified ADT for applicative functors (McBride & Paterson, 2008) and that all Traversable functors satisfy *mapPresEE*.

We conjecture that carefully distinguishing between higher order functions that can be shown to preserve extensional equality and higher order functions for which this is not the case can pay high dividends (in terms of concise and correct generic implementations and avoidance of boilerplate code) also for other abstract data types.

Acknowledgments

The authors thank the JFP editors and reviewers, whose comments have led to significant improvements of the original manuscript.

The work presented in this paper heavily relies on free software, among others on Coq, Idris, Agda, GHC, git, vi, Emacs, LaTeX and on the FreeBSD and Debian GNU/Linux operating systems. It is our pleasure to thank all developers of these excellent products. This is TiPES contribution No 38. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 820970.

Conflicts of Interest

None.

References

Altenkirch, T. (2017) From setoid hell to homotopy heaven? Available at: https://www.cs.nott.ac.uk/~psztxa/talks/types-17-hell.pdf

Arvidsson, A., Johansson, M. & Touche, R. (2019) Proving type class laws for Haskell. In *Trends in Functional Programming, Van Horn, D. & Hughes, J. (eds)*. Springer, pp. 61–74.

Bellman, R. (1957) *Dynamic Programming*. Princeton University Press.

Bird, R. (2014) *Thinking Functionally with Haskell*. Cambridge University Press.

Bird, R. S. & de Moor, O. (1997) *Algebra of Programming*. Prentice Hall International Series in Computer Science. Prentice Hall.

Bishop, E. (1967) *Foundations of Constructive Analysis*. McGraw-Hill.

Blanqui, F., *et al.* (2020) CoLoR: A Coq Library on Rewriting and termination (Version 1.8.0). Available at: https://github.com/fblanqui/color

Botta, N., Jansson, P. & Ionescu, C. (2017) Contributions to a computational theory of policy advice and avoidability. *J. Funct. Program.* **27**, 1–52.

Boulier, S., Pédrot, P.-M. & Tabareau, N. (2017) The next 700 syntactical models of type theory. In *ACM SIGPLAN Conference on Certified Programs and Proofs*, *CPP 2017*. ACM, pp. 182–194.

Brady, E. (2017) *Type-Driven Development in Idris*. Manning Publications Co.

Brede, N. & Botta, N. (2021) On the Correctness of Monadic Backward Induction. Submitted to Journal of Functional Programming, Available at: https://arxiv.org/abs/2008.02143

Carette, J. & Hu, J. Z. S. (2021) A new Categories library for Agda (Version 0.1.5). https://github.com/agda/agda-categories

Carette, J., Farmer, W. M. & Kohlhase, M. (2014) Realms: A structure for consolidating knowledge about mathematical theories. *Intell. Comput. Math.* 252–266.

Cohen, C., Coquand, T., Huber, S. & Mörtberg, A. (2018) Cubical type theory: A constructive interpretation of the univalence axiom. In *Proc. TYPES 2015*. Leibniz International Proceedings in Informatics (LIPIcs) 69, pp. 5:1–5:34. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Danielsson, N. A., Hughes, J., Jansson, P. & Gibbons, J. (2006) Fast and loose reasoning is morally correct. *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)* pp. 206–217. ACM.

De Moor, O. (1995) A generic program for sequential decision processes. In *PLILPS 1995 Symposium on Programming Languages: Implementations, Logics and Programs.* Springer, pp. 1–23.

Erwig, M. and Kollmansberger, S. (2006) FUNCTIONAL PEARLS: Probabilistic functional programming in Haskell. *J. Funct. Program.* **16**(1), 21–34.

Giry, M. (1981) A categorial approach to probability theory. In *Categorical Aspects of Topology and Analysis*, Banaschewski, B. (ed). Lecture Notes in Mathematics 915. Springer, pp. 68–85.

Gnesi, S., Montanari, U. and Martelli, A. (1981) Dynamic programming as graph searching: An algebraic approach. *J. ACM* **28**(4), 737–751.

Hofmann, M. (1995) *Extensional Concepts in Intensional Type Theory*. PhD thesis, University of Edinburgh.

Hofmann, M. & Streicher, T. (1994) The groupoid model refutes uniqueness of identity proofs. In *Proc. Symposium on Logic in Computer Science (LICS 1994),* pp. 208–212.

Hu, J. Z. S. & Carette, J. (2021) Formalizing category theory in Agda. In *ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2021, pp. 327–342.

Huet, G. & Saïbi, A. (2000) Constructive category theory. In *Proof, Language, and Interaction. Essays in Honor of Robin Milnor*, Plotkin, G., Stirling, C. & Tofte, M. (eds). MIT, pp. 239–275.

Ionescu, C. (2009) *Vulnerability Modelling and Monadic Dynamical Systems*. PhD thesis, Freie Universität Berlin.

Ionescu, C. & Jansson, P. (2013) Testing versus proving in climate impact research. In *Proc. TYPES 2011*. Leibniz International Proceedings in Informatics (LIPIcs) 19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 41–54.

Jansson, P. & Jeuring, J. (2002) Polytypic data conversion programs. *Sci. Comput. Program.* **43**(1), 35–75.

JetBrains Research. (2021) Arend Theorem Prover (Version 1.6.0). Available at: https://arend-lang.github.io/

Jeuring, J., Jansson, P. & Amaral, C. (2012) Testing type class laws. In *Proceedings of the 2012 Haskell Symposium*. ACM, pp. 49–60.

Kuznetsov, Y. A. (1998) *Elements of Applied Bifurcation Theory*. 2nd ed. Springer.

Manes, E. G. (1976) *Algebraic Theories*. Springer.

Martin-Löf, P. & Sambin, G. (1984) *Intuitionistic Type Theory*, vol. 9. Bibliopolis Naples.

McBride, C. & Paterson, R. (2008) Applicative programming with effects. *J. Funct. Program.* **18**(1), 1–13.

Megacz, A. (2011) Category Theory in Coq (Coq-Categories). Available at: http://www.megacz.com/berkeley/coq-categories/

Mu, S.-C., Ko, H.-S. & Jansson, P. (2009) Algebra of programming in Agda: dependent types for relational program derivation. *J. Funct. Program* **19**(5), 545–579.

Nordström, B., Petersson, K. & Smith, J. M. (1990) *Programming in Martin-LöfâĂŹs type theory*. International Series of Monographs on Computer Science, vol. 200. Oxford University Press.

Norell, U. (2007) *Towards a Practical Programming Language Based on Dependent Type theory*. PhD thesis, Chalmers University of Technology.

Peyton Jones, S., Tolmach, A. & Hoare, T. (2001) Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop,* pp. 203–233. ACM SIGPLAN.

Pierce, B. C. (1991) *Basic Category Theory for Computer Scientists*. Foundations of Computing Series. MIT.

Sozeau, M. (2010) A new look at generalized rewriting in type theory. *J. Formal. Reason.* **2**(1), 41–62.

Spitters, B. & Semeria, V. M. (2017) Coq Repository at Nijmegen (Version 1.2.0). Available at: https://github.com/coq-community/corn

Streicher, T. (1991) *Semantics of Type Theory - Correctness, Completeness and Independence Results*. Progress in Theoretical Computer Science. Birkhäuser.

Streicher, T. (1993) *Investigations into Intensional Type Theory*. Habilitiation Thesis, Ludwig-Maximilians-Universität München.

Streicher, T. (2003) Category Theory and Categorical Logic. Lecture Notes, Technische Universität Darmstadt. Available at: https://www2.mathematik.tu-darmstadt.de/~streicher/CTCL.pdf.

The Coq Development Team. (2021) The Coq Proof Assistant (Version 8.13.0). Available at: https://doi.org/10.5281/zenodo.4501022

The Idris Community. (2020) Documentation for the Idris Language. Available at: http://docs.idris-lang.org/en/latest/

The Univalent Foundations Program. (2013) Homotopy Type Theory: Univalent Foundations of Mathematics. Available at: https://homotopytypetheory.org/book

Thomas, R. & Arnol'd, V. (2012) *Catastrophe Theory*. Springer.

Vezzosi, A., Mörtberg, A. & Abel, A. (2021) Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *J. Funct. Program.* **31**, e8.

Voevodsky, V., Ahrens, B., Grayson, D., *et al.* . (2021) *UniMath — A Computer-Checked Library of Univalent mathematics*. Available at https://github.com/UniMath/UniMath

von Glehn, T. (2015) *Polynomials and Models of Type Theory*. PhD thesis, University of Cambridge.

Wadler, P. (1992) The essence of functional programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* pp. 1–14.

Wiegley, J. (2018) *Category Theory in Coq*. Available from https://github.com/jwiegley/category-theory