Formal Verification of Monad Transformers

Brian Huffman
Institut für Informatik, Technische Universität München
huffman@in.tum.de

Abstract

We present techniques for reasoning about constructor classes that (like the monad class) fix polymorphic operations and assert polymorphic axioms. We do not require a logic with first-class type constructors, first-class polymorphism, or type quantification; instead, we rely on a domain-theoretic model of the type system in a universal domain to provide these features.

These ideas are implemented in the Tycon library for the Isabelle theorem prover, which builds on the HOLCF library of domain theory. The Tycon library provides various axiomatic type constructor classes, including functors and monads. It also provides automation for instantiating those classes, and for defining further subclasses.

We use the Tycon library to formalize three Haskell monad transformers: the error transformer, the writer transformer, and the resumption transformer. The error and writer transformers do not universally preserve the monad laws; however, we establish datatype invariants for each, showing that they are valid monads when viewed as abstract datatypes.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs – mechanical verification.

Keywords denotational semantics, monads, polymorphism, theorem proving, type classes

1. Introduction

As a pure functional language, Haskell promises to work well for equational reasoning and proofs. Having programs and libraries that satisfy equational laws is important, because it lets programmers think about the correctness of their code in a modular and composable way.

Type classes are a valuable abstraction mechanism for writing reusable code in Haskell. Many Haskell type classes also have laws associated with them. Haskell programs that use these type classes often rely on the assumption that the laws hold. For example, a library might implement a datatype of balanced search trees, with elements of type α. To permit comparisons between elements, the search tree operations use the class constraint $\mathit{Ord}\,\alpha$, which provides the comparison operator $(\leq) :: \alpha \to \alpha \to \mathit{Bool}$. But just having an operation of the right type is not enough: For the operations to work correctly, the library requires $(\leq)$ to satisfy some additional properties, e.g. that $(\leq)$ is a *total* order.

Much Haskell code is written with equational properties in mind: Programs, libraries, and class instances may be expected to satisfy some laws, but unfortunately, there is no formal connection between programs and properties in Haskell. Haskell compilers are not able to check that properties hold. One way to get around this limitation is to verify our Haskell programs in an interactive proof assistant, or theorem prover.

Isabelle/HOL. Isabelle/HOL (or simply "Isabelle") is a generic interactive theorem prover, with tools and automation for reasoning about inductive datatypes and terminating functions in higher-order logic [13]. Isabelle has an ML-like type system extended with *axiomatic* type classes [17]. In Isabelle, a type class fixes one or more overloaded constants, just like in Haskell. But Isabelle also allows us to specify additional *class axioms* about those constants.

As an example, here we have an axiomatic class *Ord* that fixes an order relation $(\leq)$ and asserts that it is a total order:

$$\begin{array}{l}\textbf{class}\ \mathit{Ord}\,\alpha\ \textbf{where}\\ \quad (\leq) :: \alpha \to \alpha \to \mathit{Bool}\\ \quad x \leq x\\ \quad x \leq y \wedge y \leq z \implies x \leq z\\ \quad x \leq y \wedge y \leq x \implies x = y\\ \quad x \leq y \vee y \leq x\end{array}$$

(Note that free variables appearing in class axioms are treated as universally quantified.) To establish an instance of class *Ord* in Isabelle, a user must not only provide definitions of the class operations, but also proofs that the operations satisfy the class axioms.

Isabelle/HOLCF. Isabelle/HOLCF is a library of domain theory, formalized within the logic of Isabelle/HOL [5, 12]. It is designed to support denotational reasoning about programs written in pure functional languages like Haskell. HOLCF can deal with programs that are beyond the scope of Isabelle/HOL's automation: HOLCF provides tools for defining and working with (possibly lazy) recursive datatypes, general recursive functions, partial and infinite values, and least fixed-points. These features make Isabelle/HOLCF a useful system for reasoning about a significant subset of Haskell programs. With the combination of HOLCF and axiomatic classes, users can directly formalize many Haskell programs that use ad hoc overloading, and verify generic programs that may rely on laws for class instances.

Type constructor classes. In addition to ordinary type classes, Haskell also supports type *constructor* classes. An ordinary type constraint like $\mathit{Ord}\,\alpha$ involves a type variable $\alpha :: *$. The operations in such a type class have relatively simple types like $(\leq) :: (\mathit{Ord}\,\alpha) \Rightarrow \alpha \to \alpha \to \mathit{Bool}$, where no other type variables besides the one in the class constraint are mentioned. That is, for a specific class instance, the operations are monomorphic: e.g. to define an in-

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.
ACM 978-1-4503-1054-3/12/09.

stance *Ord Int*, we have an operation $(\leq^{Int}) :: Int \to Int \to Bool$. (In other words, a dictionary for class *Ord* contains only monomorphic functions.) On the other hand, a constructor class like $Functor\,\tau$ fixes a type variable of a higher kind, in this case $\tau :: * \to *$. Furthermore, the operations in a constructor class are usually polymorphic. For example, $fmap :: (Functor\,\tau) \Rightarrow (\alpha \to \beta) \to \tau\,\alpha \to \tau\,\beta$ also is polymorphic over the type variables α and β. The laws for the functor class are likewise polymorphic: For functors we usually assume that $fmap\ id = id$ and $fmap\ (f \circ g) = fmap\ f \circ fmap\ g$. For a specific functor class instance, these laws can be instantiated at various types. For a proper, law-abiding functor, we expect these laws to hold at *all* possible type instantiations.

These additional requirements pose some real challenges for formal verification. While Isabelle has built-in support for ordinary axiomatic type classes, its type system does not natively support axiomatic constructor classes or type quantification—in fact, it does not even support higher-kinded type variables at all. Other interactive theorem provers exist with stronger type systems (e.g. Coq), but switching would mean giving up all the special support for reasoning about strictness, partial values, and general recursion in HOLCF. Coq and similar provers use a logic of total, terminating functions; thus proofs conducted in them are only applicable to the total, terminating fragment of the Haskell language.

Contributions. Using a universal domain and a domain-theoretic model of types, we construct a library for Isabelle/HOLCF that gives users first-class type constructors and axiomatic constructor classes. Users can instantiate constructor classes by defining the constants and proving the class axioms at a single type. Using a combination of type coercions and naturality laws, theorems can then be transferred automatically to other type instances.

This work builds upon and improves an earlier formalization of constructor classes in Isabelle, which was joint work with Matthews and White [7]. While some concepts (e.g. representable types, the type application operator, and coercions) remain unchanged, this paper also introduces several new contributions:

- New simplified definition of class *Functor*
- Fully automatic tools for constructing *Functor* class instances
- A general, practical method for defining subclasses of *Functor*
- Automation for transferring theorems between types, using coercions and naturality laws
- Verification of error and writer monad transformers as abstract datatypes

Outline. The remainder of the paper is organized as follows. We begin by reviewing relevant information about HOLCF: After a summary of basic domain-theoretic concepts (§2), we discuss the deflation model used to represent types in HOLCF (§3). The next sections cover the implementation of the Tycon library: We show how to define the various constructor classes (§4), and then how to instantiate them (§5). Next we discuss the verification of monad transformers with Tycon (§6). Finally, we conclude with a discussion of related and future work (§7).

2. Domain theory in HOLCF

We now review the basic domain theory definitions used in HOLCF. A partial order is a set or type with a binary relation $(\sqsubseteq)$ that is reflexive, transitive, and antisymmetric. A chain is a countable increasing sequence: $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$ A complete partial order (cpo) is a partial order where every chain has a least upper bound (lub). An admissible predicate P holds for the lub of a chain whenever it holds over the entire chain: $\forall n.\, P(x_n) \Longrightarrow P(\bigsqcup_n x_n)$. A continuous function f preserves lubs of chains: $f(\bigsqcup_n x_n) = \bigsqcup_n f(x_n)$. Note also that every continuous function is monotone. A pointed cpo (pcpo) or "domain" is a cpo with a least element $\bot$. Every continuous function f on a pcpo has a least fixed-point $fix(f) = f(fix(f)) = \bigsqcup_n f^n(\bot)$. In this paper we also use the binder notation $\mu x.\, f(x)$ to denote the least fixed-point of f.

HOLCF provides a few primitive type constructors, which correspond to basic domain constructions. First, we have the continuous function space $\alpha \to \beta$, which consists of the continuous functions from α to β ordered pointwise; this type is used to model Haskell's function space. Other constructions include strict sums, strict products, and lifting. They correspond to the Haskell datatype definitions here:

$$\textbf{data}\ \alpha \oplus \beta = SLeft\ !\alpha \mid SRight\ !\beta$$
$$\textbf{data}\ \alpha \otimes \beta = SPair\ !\alpha\ !\beta$$
$$\textbf{data}\ \alpha_\bot = Lift\ \alpha$$

Note that the constructors for $\alpha \oplus \beta$ and $\alpha \otimes \beta$ are strict, but the constructor for type $\alpha_\bot$ is non-strict: $Lift\ \bot \neq \bot$. Finally, HOLCF provides the type **1**, with two elements $\bot \sqsubseteq ()$; this models Haskell's unit type $()$.

2.1 The Domain package

Constructing recursive datatypes is one important application of domain theory. In HOLCF, user-defined recursive datatypes can be specified using the Domain package [5]. It can model many of the same datatypes that we can define in Haskell, e.g., lazy lists:

$$\textbf{data}\ List\,\alpha = Nil \mid Cons\,\alpha\,(List\,\alpha)$$

Given this datatype specification, the job of the Domain package is to construct a solution to the corresponding domain equation.

$$List\,\alpha \cong \mathbf{1} \oplus (\alpha_\bot \otimes (List\,\alpha)_\bot) \tag{1}$$

Since Isabelle 2011, the Domain package is completely definitional: It explicitly constructs a solution to this equation and defines $List\,\alpha$ without introducing any new axioms [5]. In addition to the type itself, the Domain package also defines the constructor functions and several other related constants. It also generates a large collection of useful lemmas and rewrite rules, including injectivity and exhaustiveness of constructors, and rules for order comparisons like this one:

$$Cons\ x\ xs \sqsubseteq Cons\ y\ ys \Longleftrightarrow x \sqsubseteq y \wedge xs \sqsubseteq ys \tag{2}$$

We also get some induction rules generated for us: Every type gets a low-level induction principle in the form of an approximation lemma [8]. For polynomial types (i.e., those expressible as a sum of products) we also get a high-level induction rule, with cases for each constructor plus a case for $\bot$. Induction rules for lazy datatypes have an admissibility side-condition.

$$\frac{\text{admissible}(P) \qquad P(\bot) \qquad P(Nil) \qquad \forall x\ xs.\ P(xs) \Longrightarrow P(Cons\ x\ xs)}{\forall xs.\ P(xs)} \tag{3}$$

The Domain package can handle any datatype expressible in Haskell—subject to the limitations of Isabelle's type system, of course. It supports both strict and lazy constructors, mutual recursion, indirect recursion, and even negative recursion.

$$\textbf{data}\ StrictList\ \alpha = SNil \mid SCons\ \alpha\ !(StrictList\ \alpha)$$
$$\textbf{data}\ Indirect\ \alpha = Leaf\ \alpha \mid Node\ (List\ (Indirect\ \alpha))$$
$$\textbf{data}\ Neg = App\ Neg\ Neg \mid Lam\ (Neg \to Neg)$$

For formalizing Haskell record definitions, it also conveniently supports selector functions for constructor arguments.

2.2 Notation

We avoid using Isabelle notation as much as possible, favoring a Haskell-style syntax for datatype and function definitions. We also use Haskell-style notation $(\ldots) \Rightarrow \ldots$ for class constraints on type variables. Isabelle's syntax follows Standard ML in writing type constructors postfix; however, for consistency we use Haskell-style prefix type application throughout. Isabelle type constructors with multiple arguments are shown as tupled.

We consistently use Greek letters α, β, γ for type variables of kind $*$, and τ for kind $* \to *$. Latin letters a, b, c are program variables, with f, g, h, k referring specifically to functions. We often use sub- and superscripts to annotate polymorphic functions with their types; e.g., $\mathit{fmap}^{\tau}_{\alpha,\beta}$ means $\mathit{fmap} :: (\alpha \to \beta) \to \tau\,\alpha \to \tau\,\beta$.

3. Deflation model of types

HOLCF provides a special domain $\mathcal{D}$ whose values are *deflations*, a certain kind of idempotent functions. Deflations are used to model types: To each "representable" domain type in HOLCF, we associate a representation of type $\mathcal{D}$. The primary reason for having this model in HOLCF is to implement the Domain package: The deflation model of types lets us reason about the existence of solutions to domain equations, because we can construct recursively defined deflations to represent them.

The Tycon library takes advantage of this existing model of types to derive further benefits. The deflation model gives us a way to express the relationship between different type instances of polymorphic functions, letting us reason about polymorphism. It also lets us reason about type quantification, by quantifying over deflations.

In the remainder of this section, we describe the underlying concepts behind the deflation model, as well as its implementation in HOLCF.

3.1 Embedding-projection pairs and deflations

Some cpos can be embedded within other cpos. The concept of an *embedding-projection pair* (often shortened to *ep-pair*) formalizes this notion.

Definition 1. Continuous functions $e :: \alpha \to \beta$ and $p :: \beta \to \alpha$ form an *embedding-projection pair* (or *ep-pair*) if $p \circ e = \mathit{id}_\alpha$ and $e \circ p \sqsubseteq \mathit{id}_\beta$. In this case, we write $(e, p) : \alpha \overset{ep}{\to} \beta$.

Ep-pairs have many useful properties: e is injective, p is surjective, both are strict, each function uniquely determines the other, and the image of e is a sub-cpo of β. The composition of two ep-pairs yields another ep-pair: If $(e_1, p_1) : \alpha \overset{ep}{\to} \beta$ and $(e_2, p_2) : \beta \overset{ep}{\to} \gamma$, then $(e_2 \circ e_1, p_1 \circ p_2) : \alpha \overset{ep}{\to} \gamma$. Ep-pairs can also be lifted over many type constructors, including strict sums, products, and continuous function space.

Definition 2. A continuous function $d :: \alpha \to \alpha$ is a *deflation* if it is idempotent and below the identity function: $d \circ d = d \sqsubseteq \mathit{id}_\alpha$.

Deflations and ep-pairs are closely related. Given an ep-pair $(e, p) : \alpha \overset{ep}{\to} \beta$, the composition $e \circ p$ is a deflation on β whose image is isomorphic to α. Conversely, every deflation $d :: \beta \to \beta$ also gives rise to an ep-pair. Let the cpo α be the image of d; also let e be the inclusion map from α to β, and let $p = d$. Then (e, p) is an embedding-projection pair. So saying that there exists an ep-pair from α to β is equivalent to saying that there exists a deflation on β whose image is isomorphic to α. Figure 1 shows the relationship between ep-pairs and deflations.

A deflation is a function, but it can also be viewed as a set: Just take the image of the function, or equivalently, its set of fixed points—for idempotent functions they are the same. The dashed outline in Fig. 1 shows the set defined by the deflation d. Every deflation on a cpo α gives a set that is a sub-cpo, and contains $\bot$ if α has a least element. Not all sub-cpos have a corresponding deflation, but if one exists then it is unique. The set-oriented and function-oriented views of deflations also give the same ordering: For any deflations f and g, $f \sqsubseteq g$ if and only if $\mathrm{Im}(f) \subseteq \mathrm{Im}(g)$.

$$
\begin{array}{ll}
\mathit{in}_{\to} :: (\mathcal{U} \to \mathcal{U}) \to \mathcal{U} & \mathit{out}_{\to} :: \mathcal{U} \to (\mathcal{U} \to \mathcal{U}) \\
\mathit{in}_{\otimes} :: (\mathcal{U} \otimes \mathcal{U}) \to \mathcal{U} & \mathit{out}_{\otimes} :: \mathcal{U} \to (\mathcal{U} \otimes \mathcal{U}) \\
\mathit{in}_{\oplus} :: (\mathcal{U} \oplus \mathcal{U}) \to \mathcal{U} & \mathit{out}_{\oplus} :: \mathcal{U} \to (\mathcal{U} \oplus \mathcal{U}) \\
\mathit{in}_{\bot} :: \mathcal{U}_{\bot} \to \mathcal{U} & \mathit{out}_{\bot} :: \mathcal{U} \to \mathcal{U}_{\bot} \\
\mathit{in}_{\mathbf{1}} :: \mathbf{1} \to \mathcal{U} & \mathit{out}_{\mathbf{1}} :: \mathcal{U} \to \mathbf{1}
\end{array}
$$

Figure 2. Embedding-projection pairs provided by the universal domain library in HOLCF

3.2 Representable types

We say that a type α is *representable* in domain $\mathcal{U}$ if there exists an ep-pair from α to $\mathcal{U}$, or equivalently if there exists a deflation d on $\mathcal{U}$ whose image $\mathrm{Im}(d)$ is isomorphic to α. We say that $\mathcal{U}$ is a *universal domain* for some class of cpos if every cpo in the class is representable in $\mathcal{U}$. Isabelle/HOLCF provides such a universal domain type $\mathcal{U}$, which can represent any bifinite domain—this is a large class of cpos that includes (but is not limited to) all Haskell datatypes [4].

HOLCF defines an axiomatic class of representable domains. The class fixes operations *emb* and *proj*, and assumes that they form an ep-pair into the universal domain.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Rep}\,\alpha\ \textbf{where} \\
\quad \mathit{emb} :: \alpha \to \mathcal{U} \\
\quad \mathit{proj} :: \mathcal{U} \to \alpha \\
\quad \mathit{proj} \circ \mathit{emb} = \mathit{id}_\alpha \\
\quad \mathit{emb} \circ \mathit{proj} \sqsubseteq \mathit{id}_{\mathcal{U}}
\end{array}
$$

The universal domain type itself is trivially representable, using identity functions. For other base types like $\mathbf{1}$, the HOLCF universal domain library provides appropriate ep-pairs (Fig. 2).

$$
\begin{array}{ll}
\textbf{instance}\ \mathit{Rep}\,\mathcal{U}\ \textbf{where} & \textbf{instance}\ \mathit{Rep}\,\mathbf{1}\ \textbf{where} \\
\quad \mathit{emb} = \mathit{id}_{\mathcal{U}} & \quad \mathit{emb} = \mathit{in}_{\mathbf{1}} \\
\quad \mathit{proj} = \mathit{id}_{\mathcal{U}} & \quad \mathit{proj} = \mathit{out}_{\mathbf{1}}
\end{array}
$$

HOLCF defines the domain $\mathcal{D}$ of deflations over the universal domain as a subtype of $\mathcal{U} \to \mathcal{U}$. (In the Isabelle formalization, explicit conversions between types $\mathcal{D}$ and $\mathcal{U} \to \mathcal{U}$ are always required, but we will keep those implicit here.)

$$\textbf{typedef}\ \mathcal{D} = \{\, d :: \mathcal{U} \to \mathcal{U} \mid d \circ d = d \sqsubseteq \mathit{id}_{\mathcal{U}} \,\} \tag{4}$$

Definition 3 (Representation of a type)**.** Given any representable type α, we can construct its *representation* (a deflation of type $\mathcal{D}$) by composing *emb* and *proj*. We denote the mapping from types to deflations as follows:

$$[\![\alpha]\!] \overset{\text{def}}{=} \mathit{emb}_\alpha \circ \mathit{proj}_\alpha$$

Note that the representation of the universal domain $[\![\mathcal{U}]\!]$ is therefore the identity deflation, which is maximal among all deflations. Thus we have $[\![\alpha]\!] \sqsubseteq [\![\mathcal{U}]\!]$ for all representable types α.

3.3 Representable type constructors

While types can be represented by deflations, type *constructors* (which are like functions from types to types) can be represented as functions from deflations to deflations. We say that a type constructor $F :: * \to *$ is representable in $\mathcal{U}$ if there exists a continuous

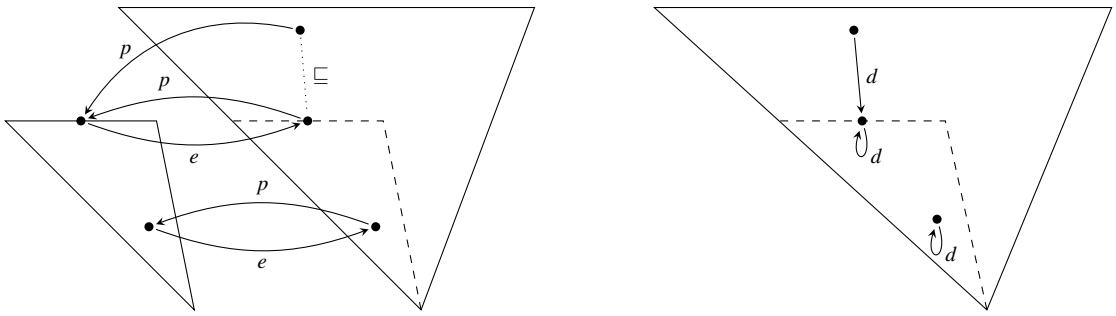

Figure 1. Embedding-projection pairs and deflations

function $F_{\mathcal{D}} :: \mathcal{D} \to \mathcal{D}$ such that $[\![F(\alpha)]\!] = F_{\mathcal{D}}\,[\![\alpha]\!]$. Such deflation combinators can be used to build deflations for recursive datatypes [2, §7]. This is precisely the technique used by recent versions of the Domain package [5, §6.6]; we will also use the same technique for creating new type constructors in Sec. 5.

We have a recipe for setting up a primitive HOLCF type as a representable type constructor: We just need an ep-pair to the universal domain and a map function. (The HOLCF universal domain library provides a selection of suitable ep-pairs; see Fig. 2.) We now demonstrate the recipe using the strict product type.

$$
\begin{aligned}
&map_{\otimes} :: (\alpha \to \alpha', \beta \to \beta') \to \alpha \otimes \beta \to \alpha' \otimes \beta' \\
&map_{\otimes}\,(f,g)\,(SPair\,x\,y) = SPair\,(f\,x)\,(g\,y)
\end{aligned}
$$

$$
\begin{aligned}
&(\otimes_{\mathcal{D}}) :: \mathcal{D} \to \mathcal{D} \to \mathcal{D} \\
&a \otimes_{\mathcal{D}} b = in_{\otimes} \circ map_{\otimes}\,(a,b) \circ out_{\otimes}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\,(Rep\,\alpha, Rep\,\beta) \Rightarrow Rep\,(\alpha \otimes \beta)\ \textbf{where} \\
&\quad emb = in_{\otimes} \circ map_{\otimes}\,(emb_{\alpha}, emb_{\beta}) \\
&\quad proj = map_{\otimes}\,(proj_{\alpha}, proj_{\beta}) \circ out_{\otimes}
\end{aligned}
$$

The reader can verify that $(\otimes_{\mathcal{D}})$ does in fact preserve deflations, that *emb* and *proj* do form an ep-pair for type $\alpha \otimes \beta$, and that $(\otimes_{\mathcal{D}})$ actually does represent the strict product type constructor: $[\![\alpha \otimes \beta]\!] = [\![\alpha]\!] \otimes_{\mathcal{D}} [\![\beta]\!]$.

Most other HOLCF type constructors work exactly like the strict product. However, the continuous function space is special because it is contravariant in its first argument.

$$
\begin{aligned}
&map_{\to} :: (\alpha' \to \alpha, \beta \to \beta') \to (\alpha \to \beta) \to (\alpha' \to \beta') \\
&map_{\to}\,(f,g)\,h = g \circ h \circ f
\end{aligned}
$$

$$
\begin{aligned}
&(\to_{\mathcal{D}}) :: \mathcal{D} \to \mathcal{D} \to \mathcal{D} \\
&a \to_{\mathcal{D}} b = in_{\to} \circ map_{\to}\,(a,b) \circ out_{\to}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\,(Rep\,\alpha, Rep\,\beta) \Rightarrow Rep\,(\alpha \to \beta)\ \textbf{where} \\
&\quad emb = in_{\to} \circ map_{\to}\,(proj_{\alpha}, emb_{\beta}) \\
&\quad proj = map_{\to}\,(emb_{\alpha}, proj_{\beta}) \circ out_{\to}
\end{aligned}
$$

Due to contravariance, the first argument to $map_{\to}$ has type $\alpha' \to \alpha$ instead of $\alpha \to \alpha'$. Also note that in the *Rep* instance, *emb* calls *proj* and vice versa. Otherwise everything works similarly to the other types.

3.4 Coercion

We can write a function to coerce between any two representable types: First embed into the universal domain $\mathcal{U}$, and then project out to a different type.

$$
\begin{aligned}
&coerce :: (Rep\,\alpha, Rep\,\beta) \Rightarrow \alpha \to \beta \\
&coerce_{\alpha,\beta} = proj_{\beta} \circ emb_{\alpha}
\end{aligned}
$$

Our primary use for coercion will be to relate different type instances of polymorphic functions. In the remainder of the paper, we will often need to prove properties about coerced values; to facilitate this, we assemble a collection of simplification rules. First of all, *coerce* may reduce to *emb*, *proj*, or *id*, depending on the type:

$$coerce_{\alpha,\alpha} = id_{\alpha} \tag{5}$$

$$coerce_{\alpha,\mathcal{U}} = emb_{\alpha} \tag{6}$$

$$coerce_{\mathcal{U},\alpha} = proj_{\alpha} \tag{7}$$

Other properties about *coerce* depend on the relative "sizes" of the source and target types. A coercion from a smaller to a larger type is injective (an embedding, in fact). Coercing twice in a row is the same as coercing once, as long as the intermediate type is larger than one of the source or target types.

$$\frac{[\![\alpha]\!] \sqsubseteq [\![\beta]\!] \vee [\![\gamma]\!] \sqsubseteq [\![\beta]\!]}{coerce_{\beta,\gamma} \circ coerce_{\alpha,\beta} = coerce_{\alpha,\gamma}} \tag{8}$$

Coercing between similar datatypes is the same as mapping *coerce* over the elements. (As an exercise, the reader may wish to verify Eq. (9) by expanding the definitions given earlier in Sec. 3.3.) Using these rules, it is easy to verify that *coerce* commutes with each data constructor.

$$coerce_{\alpha\otimes\beta,\gamma\otimes\delta} = map_{\otimes}(coerce_{\alpha,\gamma}, coerce_{\beta,\delta}) \tag{9}$$

$$coerce_{\alpha\oplus\beta,\gamma\oplus\delta} = map_{\oplus}(coerce_{\alpha,\gamma}, coerce_{\beta,\delta}) \tag{10}$$

$$coerce_{\alpha_\perp,\beta_\perp} = map_{\perp}(coerce_{\alpha,\beta}) \tag{11}$$

A similar rule holds for coercions between two function types. The expanded form in Eq. (13) will be particularly useful for simplifying coercions in later proofs.

$$coerce_{(\alpha\to\beta),(\gamma\to\delta)} = map_{\to}(coerce_{\gamma,\alpha}, coerce_{\beta,\delta}) \tag{12}$$

$$coerce_{(\alpha\to\beta),(\gamma\to\delta)}\,f = coerce_{\beta,\delta} \circ f \circ coerce_{\gamma,\alpha} \tag{13}$$

A note about the ubiquity of the *Rep* class: For the remainder of this paper, we will assume that all types $\alpha, \beta, \gamma, \ldots$ are in class *Rep*, without writing *Rep* class constraints explicitly. The reader may treat *emb*, *proj*, and *coerce* as if they were completely polymorphic. (HOLCF achieves a similar effect using the "default sort" mechanism, assigning all type variables to class *Rep* unless annotated otherwise.)

4. Type constructor classes in the Tycon library

4.1 Class Tycon and type application

In the Haskell type expression $\tau\,\alpha$, the two type variables have different kinds: Say α is an ordinary type of kind $*$; then τ may be a type constructor of kind $* \to *$. Isabelle's type system was not designed to be this expressive: All type variables in Isabelle represent ordinary types (corresponding to Haskell kind $*$).

Our solution to this limitation (originally introduced in [7]) is to define a binary Isabelle type constructor $(- \cdot -)$ that models

Haskell type application. The right argument must be in the Isabelle class *Rep*, which models Haskell kind $*$. The left argument must be in a new class *Tycon*, which models Haskell kind $* \to *$.

Class *Tycon* is defined as follows. It has no axioms, but fixes a single constant which is a deflation constructor.[1]

$$\textbf{class}\ \mathit{Tycon}\,\tau\ \textbf{where}\ \{\!|\tau|\!\} :: \mathcal{D} \to \mathcal{D}$$

Now we want to define type $\tau \cdot \alpha$ so that $[\![\tau \cdot \alpha]\!] = \{\!|\tau|\!\}[\![\alpha]\!]$. We therefore define $\tau \cdot \alpha$ as a subtype of $\mathcal{U}$, consisting of the image (or equivalently, the set of fixed-points) of the deflation $\{\!|\tau|\!\}[\![\alpha]\!]$.

$$\begin{aligned}&\textbf{typedef}\,(\mathit{Tycon}\,\tau, \mathit{Rep}\,\alpha) \Rightarrow \tau \cdot \alpha\\ &\quad = \{\, u :: \mathcal{U} \mid \{\!|\tau|\!\}[\![\alpha]\!]\, u = u \,\}\end{aligned}$$

$$\begin{aligned}&\textbf{instance}\,(\mathit{Tycon}\,\tau, \mathit{Rep}\,\alpha) \Rightarrow \mathit{Rep}\,(\tau \cdot \alpha)\ \textbf{where}\\ &\quad \mathit{emb}\,x = x\\ &\quad \mathit{proj}\,u = \{\!|\tau|\!\}[\![\alpha]\!]\, u\end{aligned}$$

Note that the definitions of *emb* and *proj* contain implicit coercions between $\mathcal{U}$ and $\tau \cdot \alpha$. The desired representation property then follows directly from the definitions of *emb* and *proj*:

$$[\![\tau \cdot \alpha]\!] = \{\!|\tau|\!\}[\![\alpha]\!] \tag{14}$$

It is worth pointing out that while the construction refers to the deflation combinator $\{\!|\tau|\!\}$, actual values of type τ are never used anywhere. This is consistent with Haskell, where there are no values inhabiting higher-kinded types.

4.2 Class Functor

The Haskell *Functor* class is for types that can be mapped over.

$$\begin{aligned}&\textbf{class}\ \mathit{Functor}\,\tau\ \textbf{where}\\ &\quad \mathit{fmap} :: (\alpha \to \beta) \to (\tau\,\alpha \to \tau\,\beta)\end{aligned}$$

Each Haskell *Functor* instance should satisfy the identity and composition laws:

$$\mathit{fmap}\,\mathit{id} = \mathit{id} \tag{15}$$

$$\mathit{fmap}\,(f \circ g) = \mathit{fmap}\,f \circ \mathit{fmap}\,g \tag{16}$$

How close are we to being able to formalize this in Isabelle? Using the type application machinery from Sec. 4.1, we can at least express the class constraint $\mathit{Functor}\,\tau$ and the result type of *fmap*. However, there are still some problems. First, let us examine the type of *fmap* more closely. The type constructor variable τ is fixed, but types α and β are actually universally quantified:

$$\mathit{fmap}^{\tau} :: \forall\,\alpha\,\beta.\,(\alpha \to \beta) \to (\tau \cdot \alpha \to \tau \cdot \beta)$$

The problem is that Isabelle's class system does not allow polymorphic class constants. Isabelle's type system does not support first-class polymorphism, and the type of class functions are only allowed to contain one free type variable, i.e., the one mentioned in the class constraint [17].

The solution is to move the polymorphism out of the class declaration. We replace the polymorphic fmap^{τ} with a single, monomorphic constant representing $\mathit{fmap}^{\tau}_{\mathcal{U},\mathcal{U}}$, the "largest" type instance of fmap^{τ}. (We use the underlined name $\underline{\mathit{fmap}}^{\tau}$ to refer to this monomorphic version.) We then define the polymorphic fmap^{τ} by coercion from $\underline{\mathit{fmap}}^{\tau}$.

$$\begin{aligned}&\textbf{class}\,(\mathit{Tycon}\,\tau) \Rightarrow \mathit{Functor}\,\tau\ \textbf{where}\\ &\quad \underline{\mathit{fmap}}^{\tau} :: (\mathcal{U} \to \mathcal{U}) \to (\tau \cdot \mathcal{U} \to \tau \cdot \mathcal{U})\end{aligned}$$

$$\begin{aligned}&\mathit{fmap} :: (\mathit{Functor}\,\tau) \Rightarrow (\alpha \to \beta) \to \tau \cdot \alpha \to \tau \cdot \beta\\ &\mathit{fmap}^{\tau}_{\alpha,\beta} = \mathit{coerce}\,\underline{\mathit{fmap}}^{\tau}\end{aligned}$$

[1] In the Isabelle formalization, we express the dependence of $\{\!|\tau|\!\}$ on type τ by adding a dummy function argument whose type is a phantom type mentioning τ.

In Haskell, polymorphically typed functions like fmap^{τ} are always *parametrically* polymorphic. That is, parametricity (a meta-property of the type system) ensures that all of the different type instances of $\mathit{fmap}^{\tau}_{\alpha,\beta}$ behave uniformly [16]. Isabelle's type system does not provide any automatic parametricity guarantees, but by defining all type instances of fmap^{τ} by coercion from a single constant, we ensure a similar kind of uniformity across type instances in the our library.

The formalization of class *Functor* is yet incomplete: We have the constant, but not the functor laws. We need to find a set of class axioms about $\underline{\mathit{fmap}}^{\tau}$ that will let us derive the polymorphic functor laws about fmap^{τ}.

As a first try, we might just write down the functor laws with all the types specialized to type $\mathcal{U}$:

$$\underline{\mathit{fmap}}^{\tau}\,\mathit{id}_{\mathcal{U}} = \mathit{id}_{\tau \cdot \mathcal{U}} \tag{17}$$

$$\underline{\mathit{fmap}}^{\tau}(f \circ g) = \underline{\mathit{fmap}}^{\tau} f \circ \underline{\mathit{fmap}}^{\tau} g \tag{18}$$

However, we shall treat these as tentative until we see whether they are sufficient to derive the polymorphic functor laws.

Theorem 1 (Identity)**.** *For any τ in class Functor and representable type α, the functor identity law holds:*

$$\mathit{fmap}^{\tau}_{\alpha,\alpha}\,\mathit{id}_{\alpha} = \mathit{id}_{\tau \cdot \alpha}$$

Proof. We start by unfolding the definition of *fmap* and rewriting with properties of *coerce*.

$$\begin{aligned}&\mathit{fmap}^{\tau}_{\alpha,\alpha}\,\mathit{id}_{\alpha}\\ &= (\mathit{coerce}\,\underline{\mathit{fmap}}^{\tau})\,\mathit{id}_{\alpha}\\ &= \mathit{coerce}_{(\tau \cdot \mathcal{U}, \tau \cdot \alpha)} \circ \underline{\mathit{fmap}}^{\tau}(\mathit{coerce}\,\mathit{id}_{\alpha}) \circ \mathit{coerce}_{(\tau \cdot \alpha, \tau \cdot \mathcal{U})}\\ &= \mathit{coerce}_{(\tau \cdot \mathcal{U}, \tau \cdot \alpha)} \circ \underline{\mathit{fmap}}^{\tau}[\![\alpha]\!] \circ \mathit{coerce}_{(\tau \cdot \alpha, \tau \cdot \mathcal{U})}\end{aligned}$$

At this point we are stuck. A law about $\underline{\mathit{fmap}}^{\tau}\,\mathit{id}_{\mathcal{U}}$ does not help here, because coercing id_{α} to type $\mathcal{U} \to \mathcal{U}$ does not yield $\mathit{id}_{\mathcal{U}}$; it gives $[\![\alpha]\!]$ instead. What we really need is a rewrite rule for $\underline{\mathit{fmap}}^{\tau}$ applied to an arbitrary deflation. The class axiom must assert that the map function $\underline{\mathit{fmap}}^{\tau}$ "agrees" with the deflation combinator $\{\!|\tau|\!\}$ in a certain sense.

Definition 4. We say that a function f on a representable type α *agrees with* a deflation d on the universal domain, if f coerced to type $\mathcal{U} \to \mathcal{U}$ is equal to d (regarded as a function).

$$(f :: \alpha \to \alpha) \Vdash (d :: \mathcal{D}) \overset{\text{def}}{\Longleftrightarrow} \mathit{emb}_{\alpha} \circ f \circ \mathit{proj}_{\alpha} = d$$

This agreement relation is already present in HOLCF: It is used internally by the Domain package for relating deflation combinators to map functions, for proving identity laws and deriving induction rules [5]. So it is fitting that it should appear in this situation, where we are again proving functor identity laws.

We replace Eq. (17) with this generalized class axiom, shown here also in its unfolded form:

$$\underline{\mathit{fmap}}^{\tau}\,(d :: \mathcal{D}) \Vdash \{\!|\tau|\!\}\,d \tag{19}$$

$$\mathit{emb}_{\tau \cdot \mathcal{U}} \circ \underline{\mathit{fmap}}^{\tau}\,(d :: \mathcal{D}) \circ \mathit{proj}_{\tau \cdot \mathcal{U}} = \{\!|\tau|\!\}\,d \tag{20}$$

Now we can continue where we left off:

$$\begin{aligned}&\mathit{coerce}_{(\tau \cdot \mathcal{U}, \tau \cdot \alpha)} \circ \underline{\mathit{fmap}}^{\tau}[\![\alpha]\!] \circ \mathit{coerce}_{(\tau \cdot \alpha, \tau \cdot \mathcal{U})}\\ &= \mathit{proj}_{\tau \cdot \alpha} \circ \mathit{emb}_{\tau \cdot \mathcal{U}} \circ \underline{\mathit{fmap}}^{\tau}[\![\alpha]\!] \circ \mathit{proj}_{\tau \cdot \mathcal{U}} \circ \mathit{emb}_{\tau \cdot \alpha}\\ &= \mathit{proj}_{\tau \cdot \alpha} \circ \{\!|\tau|\!\}[\![\alpha]\!] \circ \mathit{emb}_{\tau \cdot \alpha}\\ &= \mathit{proj}_{\tau \cdot \alpha} \circ [\![\tau \cdot \alpha]\!] \circ \mathit{emb}_{\tau \cdot \alpha}\\ &= \mathit{proj}_{\tau \cdot \alpha} \circ \mathit{emb}_{\tau \cdot \alpha} \circ \mathit{proj}_{\tau \cdot \alpha} \circ \mathit{emb}_{\tau \cdot \alpha}\\ &= \mathit{id}_{\tau \cdot \alpha} \circ \mathit{id}_{\tau \cdot \alpha}\\ &= \mathit{id}_{\tau \cdot \alpha}\end{aligned}$$

□

Theorem 2 (Composition)**.** *For any τ in class Functor and functions $f :: \beta \to \gamma$ and $g :: \alpha \to \beta$, the functor composition law holds:*

$$\mathit{fmap}^\tau_{\alpha,\gamma}\,(f \circ g) = \mathit{fmap}^\tau_{\beta,\gamma}\,f \circ \mathit{fmap}^\tau_{\alpha,\beta}\,g$$

Proof. We rewrite both sides of the equation, trying to reduce it to a trivial equality. We start by unfolding the definition of *fmap*.

$$\begin{aligned}\mathit{fmap}^\tau_{\alpha,\gamma}\,(f \circ g) &= \mathit{fmap}^\tau_{\beta,\gamma}\,f \circ \mathit{fmap}^\tau_{\alpha,\beta}\,g\\ (\mathit{coerce}\,\underline{\mathit{fmap}^\tau})(f \circ g) &= (\mathit{coerce}\,\underline{\mathit{fmap}^\tau})\,f \circ (\mathit{coerce}\,\underline{\mathit{fmap}^\tau})\,g\end{aligned}$$

After rewriting using Eq. (13), we have

$$\begin{aligned}&\mathit{coerce} \circ \underline{\mathit{fmap}^\tau}\,(\mathit{coerce}\,(f \circ g)) \circ \mathit{coerce}\\ &\quad= \mathit{coerce} \circ \underline{\mathit{fmap}^\tau}\,(\mathit{coerce}\,f) \circ \mathit{coerce}\\ &\qquad\circ \mathit{coerce} \circ \underline{\mathit{fmap}^\tau}\,(\mathit{coerce}\,g) \circ \mathit{coerce}\end{aligned}$$

In the middle of the right-hand side we have two adjacent coercions, where we go from type $\tau \cdot \mathcal{U}$ to $\tau \cdot \beta$ and back. Since the intermediate type $\tau \cdot \beta$ is smaller, they do not cancel completely. It turns out that they reduce to an application of $\underline{\mathit{fmap}^\tau}$:

$$\begin{aligned}&\mathit{coerce}_{\tau\cdot\beta,\tau\cdot\mathcal{U}} \circ \mathit{coerce}_{\tau\cdot\mathcal{U},\tau\cdot\beta}\\ &= \mathit{proj}_{\tau\cdot\mathcal{U}} \circ \mathit{emb}_{\tau\cdot\beta} \circ \mathit{proj}_{\tau\cdot\beta} \circ \mathit{emb}_{\tau\cdot\mathcal{U}}\\ &= \mathit{proj}_{\tau\cdot\mathcal{U}} \circ [\![\tau \cdot \beta]\!] \circ \mathit{emb}_{\tau\cdot\mathcal{U}}\\ &= \mathit{proj}_{\tau\cdot\mathcal{U}} \circ \{\!|\tau|\!\}\,[\![\beta]\!] \circ \mathit{emb}_{\tau\cdot\mathcal{U}}\\ &= \mathit{proj}_{\tau\cdot\mathcal{U}} \circ \mathit{emb}_{\tau\cdot\mathcal{U}} \circ \underline{\mathit{fmap}^\tau}\,[\![\beta]\!] \circ \mathit{proj}_{\tau\cdot\mathcal{U}} \circ \mathit{emb}_{\tau\cdot\mathcal{U}}\\ &= \mathit{id}_{\tau\cdot\mathcal{U}} \circ \underline{\mathit{fmap}^\tau}\,[\![\beta]\!] \circ \mathit{id}_{\tau\cdot\mathcal{U}}\\ &= \underline{\mathit{fmap}^\tau}\,[\![\beta]\!] \qquad (21)\end{aligned}$$

Rewriting the right-hand side with Eq. (21) yields three occurrences of $\underline{\mathit{fmap}^\tau}$ composed together. Using the composition rule (18) to collapse these, we get

$$\begin{aligned}&\mathit{coerce} \circ \underline{\mathit{fmap}^\tau}\,(\mathit{coerce}\,(f \circ g)) \circ \mathit{coerce}\\ &\quad= \mathit{coerce} \circ \underline{\mathit{fmap}^\tau}\,(\mathit{coerce}\,f \circ [\![\beta]\!] \circ \mathit{coerce}\,g) \circ \mathit{coerce}\end{aligned}$$

Finally, it only remains to show that the arguments to $\underline{\mathit{fmap}^\tau}$ on each side are equal. We work from right to left.

$$\begin{aligned}&\mathit{coerce}_{(\beta\to\gamma),(\mathcal{U}\to\mathcal{U})}\,f \circ [\![\beta]\!] \circ \mathit{coerce}_{(\alpha\to\beta),(\mathcal{U}\to\mathcal{U})}\,g\\ &= \mathit{emb}_\gamma \circ f \circ \mathit{proj}_\beta \circ [\![\beta]\!] \circ \mathit{emb}_\beta \circ g \circ \mathit{proj}_\alpha\\ &= \mathit{emb}_\gamma \circ f \circ \mathit{proj}_\beta \circ \mathit{emb}_\beta \circ \mathit{proj}_\beta \circ \mathit{emb}_\beta \circ g \circ \mathit{proj}_\alpha\\ &= \mathit{emb}_\gamma \circ f \circ \mathit{id}_\beta \circ \mathit{id}_\beta \circ g \circ \mathit{proj}_\alpha\\ &= \mathit{emb}_\gamma \circ f \circ g \circ \mathit{proj}_\alpha\\ &= \mathit{coerce}_{(\alpha\to\gamma),(\mathcal{U}\to\mathcal{U})}\,(f \circ g)\end{aligned}$$

□

The final formulation of class *Functor*, complete with the generalized identity law, is shown in Fig. 3.

We should note that while transfer proofs like Theorem 2 may look lengthy on paper, they are actually highly automated in Isabelle: Most such proofs require only a single call to Isabelle's simplifier, as long as the appropriate extra rewrite rules like Eq. (21) are in place. This is important for usability of the library, because users will need to perform similar transfer proofs often—not just when defining new constructor classes, but also when instantiating them.

We present proofs here in a point-free style, with liberal use of the function composition operator $(\circ)$, because it makes the proofs easier to read. However, Isabelle is not so good at reasoning modulo the associativity of function composition. Automatic proofs by rewriting work better with nested function applications, e.g. $f(g(h(x)))$ rather than $f \circ g \circ h$. Therefore, the rewrite rules and other theorems in our library are actually formalized using fully applied functions instead of function composition.

class $(\mathit{Tycon}\,\tau) \Rightarrow \mathit{Functor}\,\tau$ **where**
  $\underline{\mathit{fmap}^\tau} :: (\mathcal{U} \to \mathcal{U}) \to (\tau\cdot\mathcal{U} \to \tau\cdot\mathcal{U})$

  $\underline{\mathit{fmap}^\tau}\,(d :: \mathcal{D}) \Vdash \{\!|\tau|\!\}\,d$
  $\underline{\mathit{fmap}^\tau}\,(f \circ g) \;= \underline{\mathit{fmap}^\tau}\,f \circ \underline{\mathit{fmap}^\tau}\,g$

$\mathit{fmap} :: (\mathit{Functor}\,\tau) \Rightarrow (\alpha \to \beta) \to \tau\cdot\alpha \to \tau\cdot\beta$
$\mathit{fmap}^\tau_{\alpha,\beta} = \mathit{coerce}\,\underline{\mathit{fmap}^\tau}$

Figure 3. Isabelle *Functor* class

4.3 Generic theorems about functors

Now that we have a functor class, we can prove further theorems about *fmap*. Here is an example theorem, about its strictness. The proof uses only the functor laws and basic properties of domain theory; the result is applicable to any valid functor instance.

Theorem 3 (Strict *fmap*)**.** *If $f :: \alpha \to \beta$ is a strict function, then fmap f is also strict: $f\,\bot = \bot \implies \mathit{fmap}\,f\,\bot = \bot$.*

Proof. Fix $f :: \alpha \to \beta$, and assume $f\,\bot_\alpha = \bot_\beta$. Let $g :: \beta \to \alpha$ be the constant bottom function, $g\,x = \bot_\alpha$. From the strictness of f, it follows that $f \circ g = \mathit{const}\,\bot \sqsubseteq \mathit{id}_\beta$. We can now show the goal by antisymmetry and transitivity reasoning:

$$\begin{aligned}\mathit{fmap}\,f\,\bot \sqsubseteq\; &\mathit{fmap}\,f\,(\mathit{fmap}\,g\,\bot) && \{\text{monotonicity with } \bot\}\\ =\; &\mathit{fmap}\,(f \circ g)\,\bot && \{\text{composition law}\}\\ \sqsubseteq\; &\mathit{fmap}\,\mathit{id}\,\bot && \{\text{monotonicity, } f \circ g \sqsubseteq \mathit{id}\}\\ =\; &\bot && \{\text{identity law}\}\end{aligned}$$

Thus we have $\mathit{fmap}\,f\,\bot \sqsubseteq \bot$, which implies $\mathit{fmap}\,f\,\bot = \bot$. □

4.4 Subclasses of Functor

Users of the Tycon library can easily formalize additional constructor classes that are subclasses of *Functor*. The library already contains several examples, and they all follow the same general process.

A constructor class may fix some number of polymorphic constants, and assume a set of polymorphic class axioms. The formalized constructor class fixes a monomorphic version of each polymorphic function, with type variables instantiated to $\mathcal{U}$. Similarly, the formalized class assumes a monomorphic version of each class axiom. The polymorphic version of each functions is defined separately, using coercion. In general, we will also add a *naturality* law for each polymorphic function, which is related to the parametricity property, or free theorem, derived from its type [16]. The naturality laws are necessary for transferring properties about the monomorphic constants to the polymorphic ones.

The *Functor* class is a special case: No extra naturality law was needed for *fmap*, because the functor composition law *is* the naturality law for *fmap*. The *Monad* class is perhaps the primary motivation for this work, but the interactions and redundancies between its laws also make it a bit of a special case. The general principles are best illustrated with a more regular example. So here we present a class *FunctorPlus*, which fixes a binary append operation for combining functor values:

class $(\mathit{Functor}\,\tau) \Rightarrow \mathit{FunctorPlus}\,\tau$ **where**
  $(+\!\!+) :: \tau\,\alpha \to \tau\,\alpha \to \tau\,\alpha$

Each instance of *FunctorPlus* should also ensure that $(+\!\!+)$ is associative:

$$(x +\!\!+ y) +\!\!+ z = x +\!\!+ (y +\!\!+ z) \qquad (22)$$

Any implementation of $(+\!\!+)$ should also satisfy a naturality condition, which essentially states that it commutes with *fmap*. The

$$\textbf{class}\ (\mathit{Functor}\,\tau) \Rightarrow \mathit{FunctorPlus}\,\tau\ \textbf{where}$$
$$(\underline{+\!\!+}^{\tau}) :: \tau\cdot\mathcal{U} \to \tau\cdot\mathcal{U} \to \tau\cdot\mathcal{U}$$
$$\underline{\mathit{fmap}}^{\tau}\, f\,(x \underline{+\!\!+}^{\tau} y) = (\underline{\mathit{fmap}}^{\tau}\, f\, x) \underline{+\!\!+}^{\tau} (\underline{\mathit{fmap}}^{\tau}\, f\, y)$$
$$(x \underline{+\!\!+}^{\tau} y) \underline{+\!\!+}^{\tau} z \quad = x \underline{+\!\!+}^{\tau} (y \underline{+\!\!+}^{\tau} z)$$

$$(+\!\!+) :: (\mathit{FunctorPlus}\,\tau) \Rightarrow \tau\cdot\alpha \to \tau\cdot\alpha \to \tau\cdot\alpha$$
$$(+\!\!+^{\tau}_{\alpha}) = \mathit{coerce}\,(\underline{+\!\!+}^{\tau})$$

Figure 4. Isabelle *FunctorPlus* class

form of this law is derived from the polymorphic type of $(+\!\!+)$; it holds in Haskell as a consequence of parametricity.

$$\mathit{fmap}\, f\,(x +\!\!+ y) = (\mathit{fmap}\, f\, x) +\!\!+ (\mathit{fmap}\, f\, y) \tag{23}$$

We formalize class *FunctorPlus* in Isabelle according to the general pattern outlined above; the code is shown in Fig. 4.

The need for the naturality law becomes apparent when transferring laws to the polymorphic version of $(+\!\!+)$. When we transfer the associativity law, we get a situation similar to what we had with the proof of Theorem 2: Between the two occurrences of $(+\!\!+)$, we get a pair of coercions from $\tau\cdot\mathcal{U}$ to $\tau\cdot\alpha$ and back; these reduce to $\underline{\mathit{fmap}}^{\tau}\,[\![\alpha]\!]$. The naturality law lets us push the $\underline{\mathit{fmap}}^{\tau}$ into the arguments of the inner append, bringing the two appends together so that the monomorphic associativity rule can be applied. In the end, we are able to prove the polymorphic version of the associativity law with one call to the simplifier. Similarly, we can also derive the polymorphic version of the naturality law in one step.

4.5 Class Monad

The definition of the *Monad* class should be familiar to every Haskell programmer.

$$\textbf{class}\ \mathit{Monad}\,\tau\ \textbf{where}$$
$$\mathit{return} :: \alpha \to \tau\,\alpha$$
$$(\gg\!=) \ :: \tau\,\alpha \to (\alpha \to \tau\beta) \to \tau\beta$$

The standard monad laws are left unit, right unit, and associativity.

$$\mathit{return}\, a \gg\!= k = k\, a \tag{24}$$
$$m \gg\!= \mathit{return} = m \tag{25}$$
$$(m \gg\!= h) \gg\!= k = m \gg\!= (\lambda x.\, h\, x \gg\!= k) \tag{26}$$

To translate this Haskell class definition into Isabelle, we can follow the standard process established in Sec. 4.4: Replace the polymorphic operations with monomorphic ones, where each type variable is instantiated to $\mathcal{U}$; specialize the types in the class axioms to $\mathcal{U}$; and add naturality laws for each of the constants.

Below are the naturality laws for the monad operations, derived from their type signatures. Note that $(\gg\!=)$ has two naturality laws, because its type has two polymorphic variables.

$$\mathit{fmap}\, f\,(\mathit{return}\, a) = \mathit{return}\,(f\, a) \tag{27}$$
$$(\mathit{fmap}\, f\, m) \gg\!= k = m \gg\!= (k \circ f) \tag{28}$$
$$\mathit{fmap}\, f\,(m \gg\!= k) = m \gg\!= (\mathit{fmap}\, f \circ k) \tag{29}$$

Three monad laws plus three naturality laws would make six class axioms in total. However, it is possible to reduce this number. Using Eqs. (25) and (28), we can derive a simple definition of *fmap* in terms of $(\gg\!=)$ and *return*:

$$\mathit{fmap}\, f\, m = m \gg\!= (\mathit{return} \circ f) \tag{30}$$

This definition of *fmap* is often referred to as a fourth monad law; it is expected to hold for any Haskell type that is an instance of both the *Functor* and *Monad* classes.

$$\textbf{class}\ (\mathit{Functor}\,\tau) \Rightarrow \mathit{Monad}\,\tau\ \textbf{where}$$
$$\underline{\mathit{return}}^{\tau} :: \mathcal{U} \to \tau\cdot\mathcal{U}$$
$$(\underline{\gg\!=}^{\tau}) \ :: \tau\cdot\mathcal{U} \to (\mathcal{U} \to \tau\cdot\mathcal{U}) \to \tau\cdot\mathcal{U}$$
$$\underline{\mathit{return}}^{\tau}\, u \underline{\gg\!=}^{\tau} k = k\, u$$
$$(m \underline{\gg\!=}^{\tau} h) \gg\!= k = m \underline{\gg\!=}^{\tau} (\lambda x.\, h\, x \underline{\gg\!=}^{\tau} k)$$
$$\underline{\mathit{fmap}}^{\tau}\, f\, m \qquad = m \underline{\gg\!=}^{\tau} (\underline{\mathit{return}}^{\tau} \circ f)$$

$$\mathit{return} :: (\mathit{Monad}\,\tau) \Rightarrow \alpha \to \tau\cdot\alpha$$
$$\mathit{return}^{\tau}_{\alpha} = \mathit{coerce}\,\underline{\mathit{return}}^{\tau}$$
$$(\gg\!=) :: (\mathit{Monad}\,\tau) \Rightarrow \tau\cdot\alpha \to (\alpha \to \tau\cdot\beta) \to \tau\cdot\beta$$
$$(\gg\!=^{\tau}_{\alpha,\beta}) = \mathit{coerce}\,(\underline{\gg\!=}^{\tau})$$

Figure 5. Isabelle *Monad* class

It is simple to verify that from Eqs. (24), (26), and (30), we can derive all of the other monad and naturality laws. Thus we can use these three to formalize our *Monad* class (see Fig. 5).

From the class axioms, we re-derive the rest of the original six laws for the monomorphic constants. Then we transfer all of the laws to the polymorphic constants, using the automated method described previously in Sec. 4.4.

4.6 Generic theorems about monads

Using the polymorphic monad laws, we can proceed to prove further theorems about arbitrary monads—for example, a property about the strictness of the bind operator.

Theorem 4 (Strict $\gg\!=$)**.** *Bind is strict in its first argument, if its second argument is also strict:* $k\,\bot = \bot \Longrightarrow \bot \gg\!= k = \bot$.

Proof. By antisymmetry, it suffices to show $\bot \gg\!= k \sqsubseteq \bot$.

$$\begin{aligned} \bot \gg\!= k &\sqsubseteq \mathit{return}\,\bot \gg\!= k && \{\text{monotonicity}\} \\ &= k\,\bot && \{\text{left unit law}\} \\ &= \bot && \{\text{strictness of } k\} \end{aligned}$$ □

Within the context of class *Monad*, we can also define derived monadic constants, such as *join*.

$$\mathit{join} :: (\mathit{Monad}\,\tau) \Rightarrow \tau\cdot(\tau\cdot\alpha) \to \tau\cdot\alpha$$
$$\mathit{join}\, m = m \gg\!= \mathit{id}$$

We can derive a collection of standard lemmas about *join* by unfolding its definition and rewriting with the monad laws. These lemmas will then be valid for any type in the *Monad* class.

5. Instantiating type constructor classes

Type constructor classes like *Functor* and *Monad* are already useful on their own: For example, we can use them to formalize generic Haskell monadic operations like *sequence* and *foldM*, and prove properties about them. Using the ordinary HOLCF Domain package with the right class constraints, we can also define higher-order type constructors:

$$\textbf{data}\ \mathit{Tree}(\tau,\alpha) = \mathit{Tip} \mid \mathit{Node}\,\alpha\,(\tau\cdot(\mathit{Tree}(\tau,\alpha)))$$

This is good, but sooner or later, we will want to populate our constructor classes with some concrete instances. To show how a Tycon library user can define new functors and monads, we will now demonstrate the process with a recursive lazy list datatype.

$$\textbf{data}\ \mathit{List}\,\alpha = \mathit{Nil} \mid \mathit{Cons}\,\alpha\,(\mathit{List}\,\alpha)$$

The Domain package can handle this definition with no trouble. However, we do not want *List* to be an ordinary Isabelle type

constructor, which can only appear in fully applied form. We want *List* as a first-class type constructor, i.e., an instance of class *Tycon*. We really want to write *this* definition instead, which uses the type application operator:

$$\textbf{data}\, List \cdot \alpha = Nil \mid Cons\, \alpha\, (List \cdot \alpha)$$

The Tycon library now provides full automation for such type definitions, in the form of a new user-level type definition command. It works much like the HOLCF Domain package, and is implemented using much of the same code.

The process by which the Domain package defines new datatypes can be broken down roughly into four steps:

1. Define a deflation combinator, and use it to define a representable domain satisfying the domain equation.
2. Define constructors and related functions; generate theorems.
3. Define take function; derive induction rules.
4. Define map function; relate it to the deflation combinator.

Defining a usable *Tycon* involves essentially the same four steps. However, some of the steps are adapted slightly to deal with the *Tycon* instance and the type application operator. We now describe how our new command completes each of the four steps to make *List* into a *Tycon* and *Functor*.

1. Just like the Domain package, it constructs a deflation as a least fixed-point, based on the recursive domain equation. However, instead of defining a type $List\,\alpha$ directly from this deflation, it defines *List* as a singleton type, and makes it an instance of class *Tycon*. The constructed deflation is used to define $\{\!|List|\!\}$.

$$\{\!|List|\!\}(a) = (\mu t.\, \mathbf{1}_{\mathcal{D}} \oplus_{\mathcal{D}} (a_{\perp_{\mathcal{D}}} \otimes_{\mathcal{D}} t_{\perp_{\mathcal{D}}})) \tag{31}$$

$$[\![List \cdot \alpha]\!] = [\![\mathbf{1} \oplus (\alpha_\perp \otimes (List \cdot \alpha)_\perp)]\!] \tag{32}$$

   By unfolding the fixed-point, the desired domain equation (32) is derived. It then follows that the coercions *absList* and *repList*, defined as shown here, form an isomorphism.

$$absList_\alpha = coerce_{(\mathbf{1} \oplus (\alpha_\perp \otimes (List \cdot \alpha)_\perp), List \cdot \alpha)}$$
$$repList_\alpha = coerce_{(List \cdot \alpha, \mathbf{1} \oplus (\alpha_\perp \otimes (List \cdot \alpha)_\perp))}$$

2. Using these isomorphism theorems, a component of the Domain package is called to generate the multitude of definitions and theorems related to the constructors *Nil* and *Cons*. This step works exactly the same as with ordinary domain definitions.
3. A call to another Domain package component generates a chain of *listTake* functions:

$$\begin{aligned} &listTake :: Nat \to List \cdot \alpha \to List \cdot \alpha \\ &listTake\, 0 \quad\; xs \quad\;\; = \perp \\ &listTake\, (n+1)\, Nil \quad\;\; = Nil \\ &listTake\, (n+1)\, (Cons\, x\, xs) = Cons\, x\, (listTake\, n\, xs) \end{aligned}$$

   By reasoning about the deflation agreement relation ($\Vdash$), we can show $\bigsqcup_n listTake\, n = id$ from the definitions of *listTake* and the deflation combinator. From this, the approximation lemma [8] and induction rules are then derived, just as they are in the Domain package.
4. The final step is to instantiate the *Functor* class. The $\underline{fmap}$ function is defined in a stylized way, which exactly matches the structure of the definition of $\{\!|List|\!\}$.

$$\begin{aligned} \underline{fmap}^{List} f = (\mu t.\, &absList \circ \\ &map_\oplus(id_{\mathbf{1}}, map_\otimes(map_\perp f, map_\perp t)) \circ repList) \end{aligned} \tag{33}$$

   The Domain package would normally generate the same definition, but would define it as a separate constant *mapList*.

$$id_\alpha \Vdash [\![\alpha]\!] \qquad \frac{f \Vdash d \qquad [\![\alpha]\!] = [\![\beta]\!]}{coerce_{\beta,\alpha} \circ f \circ coerce_{\alpha,\beta} \Vdash d}$$

$$\frac{f \Vdash d}{map_\perp(f) \Vdash (d_{\perp_{\mathcal{D}}})} \qquad \frac{f_1 \Vdash d_1 \qquad f_2 \Vdash d_2}{map_\oplus(f_1, f_2) \Vdash (d_1 \oplus_{\mathcal{D}} d_2)}$$

$$\frac{f_1 \Vdash d_1 \qquad f_2 \Vdash d_2}{map_\otimes(f_1, f_2) \Vdash (d_1 \otimes_{\mathcal{D}} d_2)} \qquad \frac{f \Vdash d}{fmap^\tau f \Vdash \{\!|\tau|\!\}\, d}$$

Figure 6. Agreement rules between map functions and deflations

   The *Functor* class requires a proof of the agreement law $\underline{fmap}^{List} d \Vdash \{\!|List|\!\}\, d$. Because the definitions of $\underline{fmap}^{List}$ and $\{\!|List|\!\}$ have the same structure, the proof can be discharged using a collection of structural rules, some of which are listed in Fig. 6. The Domain package maintains this list of rules for use in its own internal proofs [5, §6.6].

It is not always possible to automatically prove the functor composition law: For some strict datatypes, the composition law can fail when used with non-strict functions. To avoid this difficulty, we split off a separate *Prefunctor* superclass that asserts only the identity law. Our new command can then always succeed in generating a *Prefunctor* instance for each new datatype; we leave it to the user instantiate the *Functor* class by proving the composition law separately.

For the *List* type constructor, composition can be proved using the ordinary HOLCF technique of induction over the datatype.

Further class instantiations. Compared to *Tycon* and *Functor*, instantiations of subclasses like *FunctorPlus* and *Monad* are relatively straightforward. We write definitions of $(\underline{+\!\!+})$, $\underline{return}$, and $(\underline{\gg\!=})$ using ordinary user-level methods: the standard Isabelle definition command for non-recursive functions, and the HOLCF Fixrec package [5] for the recursive ones. The class axioms for these subclasses are all ordinary equations, so they can be proved using ordinary techniques like induction.

Transferring theorems. We now have a type constructor *List* with instances of the *Functor*, *FunctorPlus*, and *Monad* classes. This means that we can use the polymorphic functions *fmap*, $(+\!\!+)$, *return*, and $(\gg\!=)$ at type $List \cdot \alpha$. We can also apply any generic theorems from those classes to the *List* type.

However, we do not have any *List*-specific theorems about the polymorphic functions yet. For example, if $Cons\ x\ xs \mathbin{\underline{+\!\!+}} ys = Cons\ x\ (xs \mathbin{\underline{+\!\!+}} ys)$ is one of the defining equations for $(\underline{+\!\!+})$, we should like to have a version of this theorem for $(+\!\!+)$ as well.

To obtain the polymorphic versions of such lemmas, we need to do a transfer process, much like we did with Theorem 2 and for the class axioms in Sec. 4.4. The proofs can generally be completed with one call to the simplifier, using a collection of simplification rules for coercions. To transfer theorems that mention *Nil* or *Cons*, we must first prove some additional simplification rules stating that *coerce* commutes with those data constructors. These proofs are also simple, and potentially could be generated automatically.

6. Verifying monad transformers

In addition to simple type constructors like *List*, the Tycon library can also be used to define *Tycon* instances with additional type parameters, some of which may be type constructors themselves. In particular, this means that we can define a monad transformer—i.e., a monad that is parameterized by another inner monad.

The resumption monad transformer was covered in our previous work [7], but we have some improvements here. With the improved

$$\textbf{data}\ \mathit{ResT}\ \tau\ \alpha = \mathit{Done}\ \alpha \mid \mathit{More}\ (\tau\ (\mathit{ResT}\ \tau\ \alpha))$$

$$\begin{aligned}&\textbf{instance}\ (\mathit{Functor}\ \tau) \Rightarrow \mathit{Monad}\ (\mathit{ResT}\ \tau)\ \textbf{where}\\ &\quad \mathit{return}\ x \qquad = \mathit{Done}\ x\\ &\quad \mathit{Done}\ x \gg\!= k\ = k\ x\\ &\quad \mathit{More}\ m \gg\!= k = \mathit{More}\ (\mathit{fmap}\ (\lambda r.\ r \gg\!= k)\ m)\end{aligned}$$

Figure 7. Haskell definition of *ResT* monad transformer

class definitions and better proof automation, we can now prove more with less effort: In addition to instantiating the monad class, we also proceed to define an interleaving operator and prove laws about it.

The new automation provided by the Tycon library has made it easier to test out definitions of new type constructors. Experimentation with the error and writer monad transformers has revealed that neither one truly preserves the monad laws. However, we have also found that the monad laws for both of those types actually *are* preserved for values constructible from standard operations. That is, it is possible to view each as an abstract datatype whose operations maintain an invariant; in this abstract view, each one actually does form a lawful monad.

6.1 Resumption monad transformer

The resumption monad transformer [14] augments an inner monad with the ability to suspend, resume, and interleave threads of computations. The Haskell definitions for the resumption monad transformer are shown in Fig. 7. (Note that although we call it a monad transformer, the *Monad* instance only requires τ to be a functor.)

The constructor *Done x* represents a computation that has run to completion, yielding the result x. *More c* represents a suspended computation that still has more work to do: When c is evaluated, it may produce some side effects (according to the monad τ) and eventually returns a new resumption of type $\mathit{ResT}\ \tau \cdot \alpha$. Resumptions are a bit like threads in a cooperative multitasking system: A running thread may either terminate (*Done x*) or voluntarily yield to the operating system, waiting to be resumed later (*More c*).

We formalize the Haskell type $\mathit{ResT}\ \tau\ \alpha$ as $\mathit{ResT}\ \tau \cdot \alpha$ in our library. The type constructor definition generates an *fmap* function satisfying these rules:

$$\begin{aligned}&\mathit{fmap}\ f\ (\mathit{Done}\ x) = \mathit{Done}\ (f\ x)\\ &\mathit{fmap}\ f\ (\mathit{More}\ m) = \mathit{More}\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ m)\end{aligned}$$

From the low-level principle of take induction, we derive a high-level induction rule for type $\mathit{ResT}\ \tau \cdot \alpha$:

$$\frac{\begin{array}{ll}\text{admissible}(P) & \forall x.\, P(\mathit{Done}\ x)\\ P(\bot) & \forall m\, f.\, (\forall r.\, P(f\ r)) \Longrightarrow P(\mathit{More}\ (\mathit{fmap}\ f\ m))\end{array}}{\forall r.\, P(r)} \tag{34}$$

We then proceed to instantiate the *Monad* class for $\mathit{ResT}\ \tau$; the proofs of the monad laws are all proved using the high-level induction rule. With this class instance, we have shown that *ResT* is a valid monad transformer.

Some new features of our library are nicely demonstrated by the definition and verification of an interleaving operator for resumptions [14]. The Haskell definition can be seen in Fig. 8. If both arguments are *Done*, then we combine the results and terminate.[2] While either argument is *More*, we nondeterministically choose one such argument, run it for one step, and then recurse. Note that the definition uses a *FunctorPlus* class constraint—a type class whose formalization was made possible by the new Tycon library.

[2] We combine the results with function application so that we get an applicative functor; in other contexts a pair constructor might make more sense.

$$\begin{aligned}&(\circledast) :: (\mathit{FunctorPlus}\ \tau) \Rightarrow\\ &\quad \mathit{ResT}\ \tau\ (\alpha \to \beta) \to \mathit{ResT}\ \tau\ \alpha \to \mathit{ResT}\ \tau\ \beta\\ &\mathit{Done}\ f \circledast \mathit{Done}\ x = \mathit{Done}\ (f\ x)\\ &\mathit{Done}\ f \circledast \mathit{More}\ v = \mathit{More}\ (\mathit{fmap}\ (\lambda r.\ \mathit{Done}\ f \circledast r)\ v)\\ &\mathit{More}\ u \circledast \mathit{Done}\ x = \mathit{More}\ (\mathit{fmap}\ (\lambda r.\ r \circledast \mathit{Done}\ x)\ u)\\ &\mathit{More}\ u \circledast \mathit{More}\ v = \mathit{More}\ (\mathit{fmap}\ (\lambda r.\ \mathit{More}\ u \circledast r)\ v\\ &\qquad\qquad +\!\!+ \mathit{fmap}\ (\lambda r.\ r \circledast \mathit{More}\ v)\ u)\end{aligned}$$

Figure 8. Haskell definition of interleaving operator for *ResT*

$$\textbf{data}\ \mathit{Error}\ \varepsilon\ \alpha = \mathit{Err}\ \varepsilon \mid \mathit{Ok}\ \alpha$$

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ (\mathit{Error}\ \varepsilon)\ \textbf{where}\\ &\quad \mathit{fmap}\ f\ (\mathit{Err}\ e) = \mathit{Err}\ e\\ &\quad \mathit{fmap}\ f\ (\mathit{Ok}\ a)\ = \mathit{Ok}\ (f\ a)\end{aligned}$$

$$\begin{aligned}&\textbf{instance}\ \mathit{Monad}\ (\mathit{Error}\ \varepsilon)\ \textbf{where}\\ &\quad \mathit{return}\ a \quad = \mathit{Ok}\ a\\ &\quad \mathit{Err}\ e \gg\!= k = \mathit{Err}\ e\\ &\quad \mathit{Ok}\ a \gg\!= k\ = k\ a\end{aligned}$$

Figure 9. Haskell definition of *Error* monad

$$\textbf{newtype}\ \mathit{ErrorT}\ \varepsilon\ \tau\ \alpha = \mathit{ErrorT}\ \{\mathit{runET} :: \tau\ (\mathit{Error}\ \varepsilon\ \alpha)\}$$

$$\begin{aligned}&\textbf{instance}\ (\mathit{Monad}\ \tau) \Rightarrow \mathit{Functor}\ (\mathit{ErrorT}\ \varepsilon\ \tau)\ \textbf{where}\\ &\quad \mathit{fmap}\ f\ (\mathit{ErrorT}\ t) = \mathit{ErrorT}\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ t)\end{aligned}$$

$$\begin{aligned}&\textbf{instance}\ (\mathit{Monad}\ \tau) \Rightarrow \mathit{Monad}\ (\mathit{ErrorT}\ \varepsilon\ \tau)\ \textbf{where}\\ &\quad \mathit{return}\ a = \mathit{ErrorT}\ (\mathit{return}\ (\mathit{Ok}\ a))\\ &\quad m \gg\!= k\ = \mathit{ErrorT}\ (\mathit{runET}\ m \gg\!= \lambda n.\\ &\qquad\qquad \text{case}\ n\ \text{of}\ \mathit{Err}\ e \to \mathit{return}\ (\mathit{Err}\ e)\\ &\qquad\qquad\qquad\quad \mathit{Ok}\ a \to \mathit{runET}\ (k\ a))\end{aligned}$$

Figure 10. Haskell definition of *ErrorT* monad transformer

It turns out that $(\circledast)$ satisfies all the laws of an applicative functor [11]. The trickiest to prove is the associativity law:

$$\mathit{Done}\ (\circ) \circledast u \circledast v \circledast w = u \circledast (v \circledast w) \tag{35}$$

The proof proceeds by nested inductions on u, v, and w; subproofs for the non-trivial cases rely on the naturality and associativity laws from the *FunctorPlus* class. A formalization of the same theorem was presented in the author's Ph.D thesis [5], although there it was defined with a fixed inner monad. This version is more general and more abstract. We assume exactly what we need to about the type constructor τ, nothing more.

6.2 Error monad transformer

The error monad transformer appears in Andy Gill's mtl library, inspired by Jones [9]. It is simply a composition of the inner monad with an ordinary error monad. The Haskell definition of the *Error* monad that we use is shown in Fig. 9. It is parameterized by an extra type ε, the type of error values.

We define an instance $\mathit{Monad}\ (\mathit{Error}\ \varepsilon)$ using the standard procedure outlined in Sec. 5. The formal proofs of the monad laws proceed as expected. The resulting error monad type satisfies the following domain equation:

$$[\![\mathit{Error}\ \varepsilon \cdot \alpha]\!] = [\![\varepsilon_\bot \oplus \alpha_\bot]\!] \tag{36}$$

Using the error monad type, we can now proceed to define the error monad transformer. We follow the Haskell definitions from Fig. 10, defining *ErrorT* as a newtype (i.e., a datatype with a single strict constructor).

$$\begin{aligned}&\textbf{newtype}\,(\mathit{Monad}\,\tau) \Rightarrow \mathit{ErrorT}(\varepsilon,\tau)\cdot\alpha\\&\quad = \mathit{ErrorT}\,\{\,\mathit{runET} :: \tau\cdot(\mathit{Error}\,\varepsilon\cdot\alpha)\,\}\end{aligned}$$

The HOLCF error transformer type satisfies the following domain equation. Note that as a newtype, the right-hand side of its domain equation is not lifted.

$$[\![\mathit{ErrorT}(\varepsilon,\tau)\cdot\alpha]\!] = [\![\tau\cdot(\mathit{Error}\,\varepsilon\cdot\alpha)]\!] \tag{37}$$

Building an instance of $\mathit{Functor}\,(\mathit{ErrorT}(\varepsilon,\tau))$ in the standard way, we get a definition of *fmap* that satisfies the following rule, as we would expect:

$$\begin{aligned}&\mathit{fmap}^{(\mathit{ErrorT}(\varepsilon,\tau))}\,f\,(\mathit{ErrorT}\,t) =\\&\qquad\qquad \mathit{ErrorT}\,(\mathit{fmap}^{\tau}\,(\mathit{fmap}^{\mathit{Error}(\varepsilon)}\,f)\,t)\end{aligned} \tag{38}$$

Problems with monad instance. Unfortunately, we run into difficulty when trying to prove an instance of $\mathit{Monad}\,(\mathit{ErrorT}(\varepsilon,\tau))$. Not all of the class axioms are provable. The *Monad* class will not let us define constants *return* and $(\gg\!=)$ that do not satisfy the laws, so instead we define the *return* and $(\gg\!=)$ from Fig. 10 as separate constants *unitET* and *bindET*. These and other HOLCF definitions for the error monad transformer type are shown in Fig. 11.

Using this collection of non-overloaded constants, we can examine in detail the situations where the laws fail. In fact, most of the expected laws, e.g. the left unit law, do hold in general. All of the lemmas shown below can be proven by showing that *runET* applied to each side yields the same value.

$$\mathit{bindET}\,(\mathit{unitET}\,a)\,k = k\,a \tag{39}$$

$$\mathit{catchET}\,(\mathit{throwET}\,e)\,h = h\,e \tag{40}$$

$$\mathit{bindET}\,(\mathit{throwET}\,e)\,k = \mathit{throwET}\,e \tag{41}$$

$$\mathit{catchET}\,(\mathit{unitET}\,a)\,h = \mathit{unitET}\,a \tag{42}$$

$$\mathit{liftET}\,(\mathit{return}^{\tau}\,a) = \mathit{unitET}\,a \tag{43}$$

$$\mathit{liftET}\,(t \gg\!=^{\tau} k) = \mathit{bindET}\,(\mathit{liftET}\,t)\,(\mathit{liftET}\circ k) \tag{44}$$

A more involved proof shows that associativity also holds for *bindET*.

Theorem 5. *The error monad transformer satisfies the monad associativity law.*

$$\mathit{bindET}\,(\mathit{bindET}\,m\,h)\,k = \mathit{bindET}\,m\,(\lambda a.\,\mathit{bindET}\,(h\,a)\,k)$$

Proof. Let $R(k)$ abbreviate the lambda expression in the definition of *bindET*, so that $\mathit{runET}\,(\mathit{bindET}\,m\,k) = \mathit{runET}\,m \gg\!= R(k)$. Also note that $R(k)$ is strict. The proof then proceeds by applying *runET* to both sides of the equation. After simplification, we have:

$$\begin{aligned}&(\mathit{runET}\,m \gg\!= R(h)) \gg\!= R(k)\\&\qquad\qquad = \mathit{runET}\,m \gg\!= R(\lambda a.\,\mathit{bindET}\,(h\,a)\,k)\end{aligned}$$

After rewriting the left-hand side with the associativity law, both sides have the form $\mathit{runET}\,m \gg\!= f$. It then suffices to show that the functions on both sides are equal for all arguments:

$$\forall x.\,R(h)\,x \gg\!= R(k) = R(\lambda a.\,\mathit{bindET}\,(h\,a)\,k)\,x$$

We proceed by cases on x. If $x = \bot$, then using Theorem 4 we see that both sides reduce to $\bot$. If $x = \mathit{Err}\,e$, then both sides reduce to $\mathit{return}^{\tau}\,(\mathit{Err}\,e)$. Finally, if $x = \mathit{Ok}\,a$, then both sides evaluate to $\mathit{runET}\,(h\,a) \gg\!= R(k)$. □

On the other hand, the right unit monad law is not satisfied in general. Unless the inner monad τ has a strict *return* function, $m = \mathit{ErrorT}\,(\mathit{return}\,\bot)$ is a counterexample to the right unit law.

$$\mathit{unitET}\,a \in \textsc{Inv} \qquad \frac{m \in \textsc{Inv} \qquad \forall a.\,k\,a \in \textsc{Inv}}{\mathit{bindET}\,m\,k \in \textsc{Inv}}$$

$$\mathit{throwET}\,e \in \textsc{Inv} \qquad \frac{m \in \textsc{Inv} \qquad \forall e.\,h\,e \in \textsc{Inv}}{\mathit{catchET}\,m\,h \in \textsc{Inv}}$$

$$\mathit{liftET}\,t \in \textsc{Inv} \qquad \bot \in \textsc{Inv} \qquad \frac{\forall i.\,m_i \in \textsc{Inv}}{\bigsqcup_i m_i \in \textsc{Inv}}$$

Figure 12. Inductive invariant based on *ErrorT* abstract interface

Theorem 6. *The error monad transformer satisfies the right unit law if and only if the inner monad has a strict return.*

$$(\forall m.\,\mathit{bindET}^{\tau}\,m\,\mathit{unitET}^{\tau} = m) \Longleftrightarrow (\mathit{return}^{\tau}\,\bot = \bot)$$

Proof. Case $(\Longrightarrow)$: If we instantiate $m = \mathit{ErrorT}\,(\mathit{return}\,\bot)$, then the equation reduces to $\bot = \mathit{return}\,\bot$. Case $(\Longleftarrow)$: As above, let $R(k)$ abbreviate the lambda expression in the definition of *bindET*. We proceed to show $\mathit{bindET}\,m\,\mathit{unitET} = m$ by applying *runET* to both sides. After simplification, we get:

$$\mathit{runET}\,m \gg\!= R(\mathit{unitET}) = \mathit{runET}\,m$$

After expanding the right-hand side with the right unit law, both sides have the form $\mathit{runET}\,m \gg\!= f$. It then suffices to show that the functions on both sides are equal for all arguments:

$$\forall x.\,R(\mathit{unitET})\,x = \mathit{return}\,x$$

If $x = \bot$, then the equation reduces to $\bot = \mathit{return}\,\bot$, which we solve by assumption. In case $x = \mathit{Err}\,e$ or $x = \mathit{Ok}\,a$, then the equation reduces to a trivial equality. □

We could prove a monad class instance for the error transformer by creating a subclass for monads-with-strict-return, and putting a stronger constraint on type τ:

$$\textbf{instance}\,(\mathit{StrictMonad}\,\tau) \Rightarrow \mathit{Monad}\,(\mathit{ErrorT}(\varepsilon,\tau))$$

However, this is not very useful in practice, because most monads do not have a strict *return* function (although there are a few that do, e.g. the Identity monad and some varieties of powerdomains).

Data abstraction to the rescue. It turns out that it is impossible to construct the offending value $\mathit{ErrorT}\,(\mathit{return}\,\bot)$ using only the standard operations listed in Fig. 11. Furthermore, we can show that for all values constructible using those operations, the monad laws do always hold. This means that when viewed as an abstract datatype, we could still consider *ErrorT* to be a valid monad.

We define an inductive set INV that includes all values that can be constructed with functions in the abstract interface (see Fig. 12). We must also include rules for $\bot$ and lubs, to ensure that the set INV is a pcpo: In Haskell it is possible to define recursive values (i.e., least fixed-points) at any type, abstract or not.

Finally, we can prove a restricted form of the right unit law by induction on INV. The proof is straightforward, and uses techniques similar to those used for Theorems 5 and 6.

$$m \in \textsc{Inv} \implies \mathit{bindET}\,m\,\mathit{unitET} = m \tag{45}$$

Besides using an inductive set, there is another, more direct way of defining the invariant. We can define INV simply as the set of all values satisfying the right unit law:

$$\textsc{Inv} = \{\,m \mid \mathit{bindET}\,m\,\mathit{unitET} = m\,\} \tag{46}$$

It turns out that $(\lambda m.\,\mathit{bindET}\;m\;\mathit{unitET})$ is actually a deflation, of which this version of INV is the corresponding set. (The reader may wish to verify that it is idempotent and below *id*.)

$$unitET :: (Monad\,\tau) \Rightarrow \alpha \to ErrorT(\varepsilon,\tau)\cdot\alpha$$
$$unitET\,a = ErrorT\,(return^{\tau}\,(Ok\,a))$$

$$bindET :: (Monad\,\tau) \Rightarrow ErrorT(\varepsilon,\tau)\cdot\alpha \to (\alpha \to ErrorT(\varepsilon,\tau)\cdot\beta) \to ErrorT(\varepsilon,\tau)\cdot\beta$$
$$bindET\,m\,k = ErrorT\,(runET\,m \gg\!=^{\tau} \lambda x.\,\text{case } x \text{ of } Err\,e \to return^{\tau}\,(Err\,e); Ok\,a \to runET\,(k\,a))$$

$$liftET :: (Monad\,\tau) \Rightarrow \tau\cdot\alpha \to ErrorT(\varepsilon,\tau)\cdot\alpha$$
$$liftET\,t = ErrorT\,(fmap^{\tau}\,Ok\,t)$$

$$throwET :: (Monad\,\tau) \Rightarrow \varepsilon \to ErrorT(\varepsilon,\tau)\cdot\alpha$$
$$throwET\,e = ErrorT\,(return^{\tau}\,(Err\,e))$$

$$catchET :: (Monad\,\tau) \Rightarrow ErrorT(\varepsilon,\tau)\cdot\alpha \to (\varepsilon \to ErrorT(\varepsilon,\tau)\cdot\alpha) \to ErrorT(\varepsilon,\tau)\cdot\alpha$$
$$catchET\,m\,h = ErrorT\,(runET\,m \gg\!=^{\tau} \lambda x.\,\text{case } x \text{ of } Err\,e \to runET\,(h\,e); Ok\,a \to return^{\tau}\,(Ok\,a))$$

Figure 11. Isabelle definitions of error monad transformer operations

class $Monoid\,\omega$ **where**
$\varnothing :: \omega$
$(\bullet) :: \omega \to \omega \to \omega$

data $Writer\,\omega\,\alpha = Result\,\omega\,\alpha$

newtype $WriterT\,\omega\,\tau\,\alpha = WriterT\,\{runWT :: \tau\,(Writer\,\omega\,\alpha)\}$

instance $(Monoid\,\omega, Monad\,\tau) \Rightarrow Monad\,(WriterT\,\omega\,\tau)$ **where**
$return\,a = WriterT\,(return\,(Result\,\varnothing\,a))$
$m \gg\!= k \;= WriterT\,(runWT\,m \gg\!= \lambda(Result\,w_1\,a).$
$\quad runWT\,(k\,a) \gg\!= \lambda(Result\,w_2\,b).$
$\quad return\,(Result\,(w_1 \bullet w_2)\,b))$

$tell :: \omega \to WriterT\,\omega\,\tau\,()$
$tell\,w = WriterT\,(return\,(Result\,w\,()))$

$listen :: WriterT\,\omega\,\tau\,\alpha \to WriterT\,\omega\,\tau\,(Writer\,\omega\,\alpha)$
$listen\,m = WriterT\,(runWT\,m \gg\!= \lambda(Result\,w\,a).$
$\quad Result\,w\,(Result\,w\,a))$

Figure 13. Haskell definition of *WriterT* monad transformer

Conveniently, we are already using deflations as our model of types. Therefore, we can use this deflation to define a new representable subtype of $ErrorT(\varepsilon,\tau)\cdot\alpha$ that is isomorphic to the set INV. The representation of the new type $ErrorT'(\varepsilon,\tau)\cdot\alpha$ as a deflation is therefore as follows:

$$[\![ErrorT'(\varepsilon,\tau)\cdot\alpha]\!] = emb \circ (\lambda m.\,bindET\,m\,unitET) \circ proj \quad (47)$$

We have implemented such a type definition using the Tycon library, and proven a *Monad* class instance for it. However, we do not yet have a principled technique for transferring definitions or theorems between the *ErrorT* and *ErrorT'* types, so working with such subtypes is impractical for casual users. Exploring ways to automate this process will be an area for future research.

6.3 Writer monad transformer

The writer monad allows a program to output a string (or more generally, any *Monoid* type) along with its ordinary result [9]. The bind operation of the monad concatenates the strings output by each sub-computation. The writer monad transformer composes the writer monad with an inner monad, extending the inner monad with a string output capability. The Haskell definitions are shown in Fig. 13.

The Haskell *Monoid* class has a set of customary axioms: Instances should ensure that $(\bullet)$ is associative, with $\varnothing$ as the identity element, so $\varnothing \bullet x = x \bullet \varnothing = x$. Note that *Monoid* is not a constructor class, so we can formalize it as an ordinary Isabelle type class.

The formalization of the writer monad transformer works out in almost exactly the same way as the error monad transformer: The type definitions and *Functor* instances work fine, but the monad instance fails because neither the left nor the right unit law holds in general. To reason about *return* and *bind* without a *Monad* class instance, we define functions *unitWT* and *bindWT* according to the definitions in Fig. 13.

Theorem 7. *The writer monad transformer satisfies the right unit law if and only if the inner monad has a strict return.*

$$(\forall m.\,bindWT^{\tau}\,m\,unitWT^{\tau} = m) \Longleftrightarrow (return^{\tau}\,\bot = \bot)$$

Proof. Similar to Theorem 6. In the case that *return* is not strict, instantiating $m = WriterT\,(return\,\bot)$ gives the counterexample. □

Theorem 8. *The writer monad transformer satisfies the left unit law if and only if the inner monad has a strict return.*

$$(\forall x\,k.\,bindWT^{\tau}\,(unitWT^{\tau}\,x)\,k = k\,x) \Longleftrightarrow (return^{\tau}\,\bot = \bot)$$

Proof. Similar to Theorem 7. In case *return* is not strict, instantiating $k = \lambda x.\,WriterT\,(return\,\bot)$ gives the counterexample. □

As with the error monad transformer, we can define a subset of type consisting of those values that satisfy the right unit law:

$$\text{INV} = \{\,m \mid bindWT\,m\,unitWT = m\,\} \quad (48)$$

It is straightforward to check that all writer transformer operations preserve this invariant, including *unitWT*, *bindWT*, and the formalized versions of *tell* and *listen*.

The reader may verify that the function $\lambda m.\,bindWT\,m\,unitWT$ is indeed a deflation. But we are not quite done showing that the subtype defined by INV is a monad: Because the left unit law does not hold universally for the writer transformer, we must also verify that all values in INV satisfy the left unit law as well.

$$k\,x \in \text{INV} \implies bindWT\,(unitWT\,x)\,k = k\,x \quad (49)$$

Unfolding the definition of INV, we see that it is sufficient to show $bindWT\,(unitWT\,x)\,k = bindWT\,(k\,x)\,unitWT$. This can easily be proven by applying *runWT* to both sides and simplifying.

In summary, we have seen that the writer monad transformer is not quite a true monad, because the type contains values that do not respect the monad laws. But when we view it as an abstract datatype, with an interface that exports only operations that preserve the datatype invariant, it is valid to treat it as a real monad.

7. Conclusions and related work

The Tycon library for Isabelle/HOLCF is now available at the Archive of Formal Proofs [6]. It allows users to define, reason about, and instantiate constructor classes with little effort. It models polymorphism using coercion from a universal domain, which allows it to work in ordinary higher-order logic.

A different domain-theoretic model of polymorphism is presented by Amadio and Curien [1]. Here, polymorphic functions are modeled as functions from types (i.e. deflations) to values. However, this model allows non-parametric polymorphic functions that depend non-trivially on the type argument. Also, building a Tycon library around this model would also require users to write explicit type abstractions and applications when instantiating constructor classes; it is not clear whether this would be practical for users.

Sozeau and Oury [15] have recently developed a type class mechanism for the Coq theorem prover. Coq has a powerful dependent type system that allows reasoning about type constructors, first-class polymorphic values and type quantification. They define a monad class, including monad laws. Their system has the capability to formalize the whole monad class hierarchy, and it appears that it could be used to verify monad transformers; however, we are unaware of any published work in that direction.

Formalizing monad transformers in Coq does have some limitations compared to the Tycon library. For example, Coq does not accept the type definition of the resumption monad transformer: To ensure the strict positivity requirement, indirect recursion can only be used with known type constructors, not a monad parameter. Another difference (not necessarily a limitation) is that Coq is a logic of terminating functions, and does not include notions of bottoms, strictness, or partial values. Results proved in such a logic must be interpreted differently.

Our earlier formalization of axiomatic constructor classes [7] could express many of the same definitions as the current work. However, it did not provide as many features or as much automation for users. Instead of naturality laws, it used a deflation membership relation, written $x ::: d$, to express the fact that polymorphic functions had the right type. For example, the *Monad* class there had a rule for $\underline{return}$ that stated $\forall x ::: d.\ \underline{return}\ x ::: \{\!|\tau|\!\}(d)$. Transfer proofs to establish polymorphic laws were lengthy and unprincipled, making subclass definitions impractical for users. Automation for instantiating the *Functor* and *Monad* classes was present, but it required users to define $\underline{fmap}$ on a separate copy of the datatype first. Users were then left without a good way to transfer properties to the new *Tycon* version of the type.

Automation for theorem transfer in the Tycon library is much smoother than it was in our earlier work, but there is still room for further improvement. Currently we rely on a set of rewrite rules, which works well in practice so far. However, the behavior of such rewriting strategies is often hard to predict, the rules were assembled in an ad hoc fashion, and we have no convincing reason to trust that the method will work in all situations.

A better approach would be to use a more principled theorem transfer method, like the quotient packages developed recently by Homeier [3] and Kaliszyk & Urban [10]. For functions respecting an equivalence relation, theorems can be transferred from the underlying "raw" type to a quotient type. In HOLCF, type $\mathcal{U}$ could be considered as a raw type, with a representable type α as a quotient type; the *proj* function induces an equivalence relation on $\mathcal{U}$. The naturality laws for operations on $\mathcal{U}$ could then serve as the respectfulness theorems required by the quotient package.

Acknowledgments

Thanks to John Matthews for many discussions about HOLCF which helped to develop the ideas in this paper. Thanks also to Jasmin Blanchette for reading an early draft and providing helpful comments.

References

[1] Roberto M. Amadio and Pierre-Louis Curien. *Domains and Lambda-Calculi*. Cambridge University Press, New York, NY, USA, 1998.

[2] Carl A. Gunter and Dana S. Scott. Semantic domains. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 633–674. MIT Press, 1990.

[3] Peter V. Homeier. A design structure for higher order quotients. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '05)*, volume 3603 of *LNCS*, pages 130–146. Springer-Verlag, 2005.

[4] Brian Huffman. A purely definitional universal domain. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*, volume 5674 of *LNCS*, pages 260–275. Springer, 2009.

[5] Brian Huffman. *HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs*. Ph.D. thesis, Portland State University, 2012.

[6] Brian Huffman. Type constructor classes and monad transformers. *Archive of Formal Proofs*, June 2012. `http://afp.sf.net/entries/Tycon.shtml`, Formal proof development.

[7] Brian Huffman, John Matthews, and Peter White. Axiomatic constructor classes in Isabelle/HOLCF. In Joe Hurd and Tom Melham, editors, *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '05)*, volume 3603 of *LNCS*, pages 147–162. Springer, 2005.

[8] Graham Hutton and Jeremy Gibbons. The generic approximation lemma. *Information Processing Letters*, 79:2001, 2001.

[9] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *LNCS*, Båstad, Sweden, May 1995. Springer-Verlag.

[10] Cezary Kaliszyk and Christian Urban. Quotients revisited for Isabelle/HOL. In William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proc. of the 26th ACM Symposium on Applied Computing (SAC'11)*, pages 1639–1644. ACM, 2011.

[11] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[12] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[13] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[14] Nikolaos S. Papaspyrou. A resumption monad transformer and its applications in the semantics of concurrency. In *Proceedings of the 3rd Panhellenic Logic Symposium*, Anogia, Greece, July 2001.

[15] Matthieu Sozeau and Nicolas Oury. First-class type classes. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference (TPHOLs '08)*, volume 5170 of *LNCS*, pages 278–293. Springer, August 2008.

[16] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.

[17] Markus Wenzel. Type classes and overloading in higher-order logic. In E. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '97)*, volume 1275 of *LNCS*, pages 307–322, Murray Hill, New Jersey, 1997.