

Tycon: Type Constructor Classes and Monad Transformers

Brian Huffman

June 17, 2024

Abstract

These theories contain a formalization of first class type constructors and axiomatic constructor classes for HOLCF. This work is described in detail in the ICFP 2012 paper “Formal Verification of Monad Transformers” by the author [1]. The formalization is a revised and updated version of earlier joint work with Matthews and White [3].

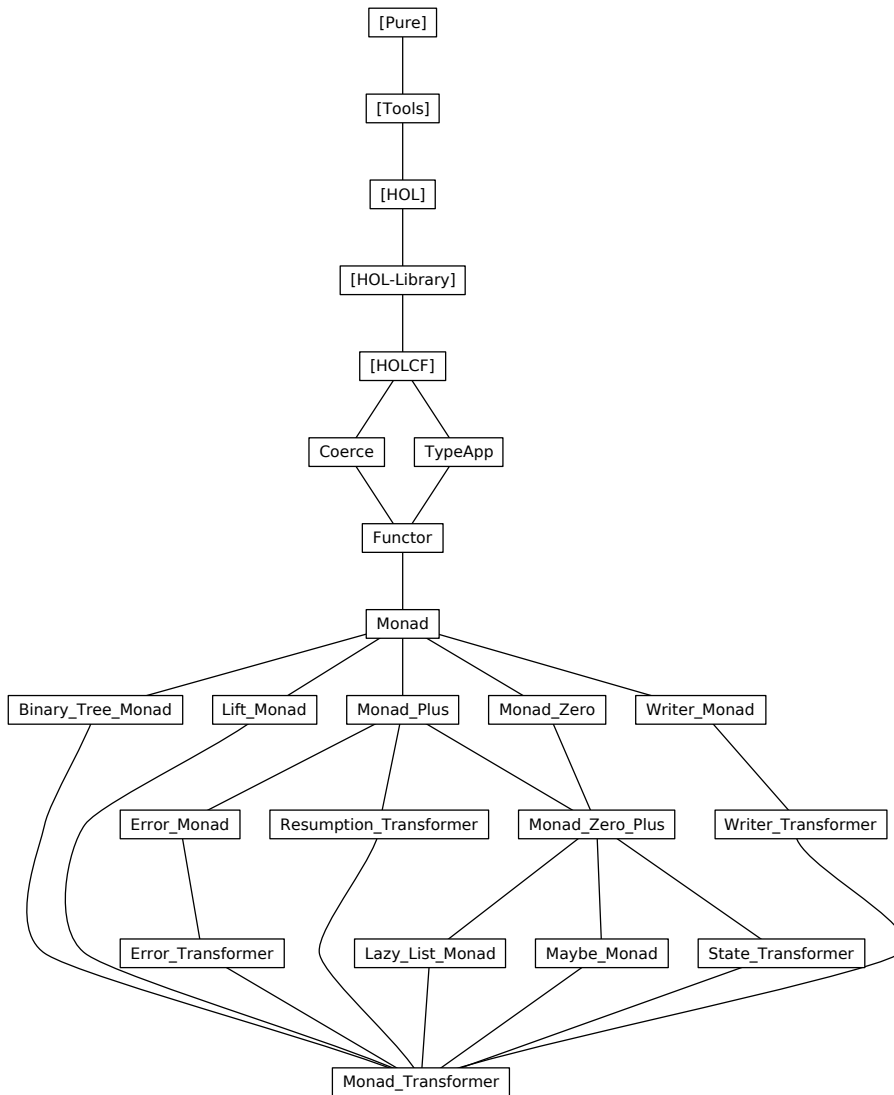
Based on the hierarchy of type classes in Haskell, we define classes for functors, monads, monad-plus, etc. Each one includes all the standard laws as axioms. We also provide a new user command, *tycondef*, for defining new type constructors in HOLCF. Using *tycondef*, we instantiate the type class hierarchy with various monads and monad transformers.

Contents

1	Type Application	5
1.1	Class of type constructors	5
1.2	Type constructor for type application	5
2	Coercion Operator	6
2.1	Coerce	6
2.2	More lemmas about <i>emb</i> and <i>prj</i>	7
2.3	Coercing various datatypes	8
2.4	Simplifying coercions	8
3	Functor Class	9
3.1	Class definition	9
3.2	Polymorphic functor laws	10
3.3	Derived properties of <i>fmap</i>	11
3.4	Proving that $fmap \cdot coerce = coerce$	11
3.5	Lemmas for reasoning about coercion	11
3.6	Configuration of Domain package	12
3.7	Configuration of the Tycon package	12

4	Monad Class	12
4.1	Class definition	12
4.2	Naturality of bind and return	13
4.3	Polymorphic versions of class assumptions	14
4.4	Derived rules	14
4.5	Laws for join	14
4.6	Equivalence of monad laws and fmap/join laws	15
4.7	Simplification of coercions	15
5	Monad-Zero Class	16
6	Monad-Plus Class	16
7	Monad-Zero-Plus Class	18
8	Lazy list monad	18
8.1	Type definition	19
8.2	Class instances	19
8.3	Transfer properties to polymorphic versions	21
9	Maybe monad	21
9.1	Type definition	22
9.2	Class instance proofs	22
9.3	Transfer properties to polymorphic versions	23
9.4	Maybe is not in <i>monad-plus</i>	23
10	Error monad	24
10.1	Type definition	24
10.2	Monad class instance	24
10.3	Transfer properties to polymorphic versions	25
11	Writer monad	25
11.1	Monoid class	26
11.2	Writer monad type	26
11.3	Class instance proofs	26
11.4	Transfer properties to polymorphic versions	27
11.5	Extra operations	27
12	Binary tree monad	27
12.1	Type definition	28
12.2	Class instance proofs	28
12.3	Transfer properties to polymorphic versions	28

13 Lift monad	29
13.1 Type definition	29
13.2 Class instance proofs	29
13.3 Transfer properties to polymorphic versions	30
14 Resumption monad transformer	30
14.1 Type definition	30
14.2 Class instance proofs	31
14.3 Transfer properties to polymorphic versions	32
14.4 Nondeterministic interleaving	32
15 State monad transformer	33
15.1 Functor class instance	34
15.2 Monad class instance	34
15.3 Monad zero instance	35
15.4 Monad plus instance	35
15.5 Monad zero plus instance	35
15.6 Transfer properties to polymorphic versions	35
16 Error monad transformer	36
16.1 Type definition	36
16.2 Functor class instance	37
16.3 Transfer properties to polymorphic versions	38
16.4 Monad operations	38
16.5 Laws	39
16.6 Error monad transformer invariant	40
16.7 Invariant expressed as a deflation	41
17 Writer monad transformer	42
17.1 Type definition	42
17.2 Functor class instance	43
17.3 Monad operations	44
17.4 Laws	45
17.5 Writer monad transformer invariant	46
17.6 Invariant expressed as a deflation	47



1 Type Application

```
theory TypeApp  
imports HOLCF  
begin
```

1.1 Class of type constructors

In HOLCF, the type *udom defl* consists of deflations over the universal domain—each value of type *udom defl* represents a bifinite domain. In turn, values of the continuous function type *udom defl* \rightarrow *udom defl* represent functions from domains to domains, i.e. type constructors.

Class *tycon*, defined below, will be populated with dummy types: For example, if the type *foo* is an instance of class *tycon*, then users will never deal with any values *x::foo* in practice. Such types are only used with the overloaded constant *tc*, which associates each type *'a::tycon* with a value of type *udom defl* \rightarrow *udom defl*.

```
class tycon =  
  fixes tc :: ('a::type) itself  $\Rightarrow$  udom defl  $\rightarrow$  udom defl
```

Type *'a itself* is defined in Isabelle’s meta-logic; it is inhabited by a single value, written *TYPE('a)*. We define the syntax *TC('a)* to abbreviate *tc TYPE('a)*.

```
syntax -TC :: type  $\Rightarrow$  logic ((1TC/(1'(-))))
```

```
translations TC('a)  $\equiv$  CONST tc TYPE('a)
```

1.2 Type constructor for type application

We now define a binary type constructor that models type application: Type *('a, 't) app* is the result of applying the type constructor *'t* (from class *tycon*) to the type argument *'a* (from class *domain*).

We define type *('a, 't) app* using *domaindef*, a low-level type-definition command provided by HOLCF (similar to *typedef* in Isabelle/HOL) that defines a new domain type represented by the given deflation. Note that in HOLCF, *DEFL('a)* is an abbreviation for *defl TYPE('a)*, where *defl* :: ('a::domain) *itself* \Rightarrow *udom defl* is an overloaded function from the *domain* type class that yields the deflation representing the given type.

```
domaindef ('a,'t) app = TC('t::tycon).DEFL('a::domain)
```

We define the infix syntax *'a.'t* for the type *('a,'t) app*. Note that for consistency with Isabelle’s existing type syntax, we have used postfix order for type application: type argument on the left, type constructor on the right.

type-notation *app* ((*...*) [*999,1000*] *999*)

The *domaindef* command generates the theorem *DEFL-app*: $DEFL(?'a.'?t) = TC(?'t) \cdot DEFL(?'a)$, which we can use to derive other useful lemmas.

lemma *TC-DEFL*: $TC('t::tycon) \cdot DEFL('a) = DEFL('a.'t)$
<proof>

lemma *DEFL-app-mono* [*simp, intro*]:
 $DEFL('a) \sqsubseteq DEFL('b) \implies DEFL('a.'t::tycon) \sqsubseteq DEFL('b.'t)$
<proof>

end

2 Coercion Operator

theory *Coerce*
imports *HOLCF*
begin

2.1 Coerce

The *domain* type class, which is the default type class in HOLCF, fixes two overloaded functions: $emb::'a \rightarrow udom$ and $prj::udom \rightarrow 'a$. By composing the *prj* and *emb* functions together, we can coerce values between any two types in class *domain*.

definition *coerce* :: $'a \rightarrow 'b$
where *coerce* $\equiv prj \circ emb$

When working with proofs involving *emb*, *prj*, and *coerce*, it is often difficult to tell at which types those constants are being used. To alleviate this problem, we define special input and output syntax to indicate the types.

syntax
-*emb* :: *type* \Rightarrow *logic* ((*1EMB*/*1'(-)*))
-*prj* :: *type* \Rightarrow *logic* ((*1PRJ*/*1'(-)*))
-*coerce* :: *type* \Rightarrow *type* \Rightarrow *logic* ((*1COERCE*/*1'(-, / -)*))

translations
 $EMB('a) \rightarrow CONST\ emb :: 'a \rightarrow udom$
 $PRJ('a) \rightarrow CONST\ prj :: udom \rightarrow 'a$
 $COERCE('a,'b) \rightarrow CONST\ coerce :: 'a \rightarrow 'b$

<ML>

lemma *beta-coerce*: $coerce \cdot x = prj \cdot (emb \cdot x)$
<proof>

lemma *prj-emb*: $prj \cdot (emb \cdot x) = coerce \cdot x$
<proof>

lemma *coerce-strict* [*simp*]: $coerce \cdot \perp = \perp$
<proof>

Certain type instances of *coerce* may reduce to the identity function, *emb*, or *prj*.

lemma *coerce-eq-ID* [*simp*]: $COERCE('a, 'a) = ID$
<proof>

lemma *coerce-eq-emb* [*simp*]: $COERCE('a, udom) = EMB('a)$
<proof>

lemma *coerce-eq-prj* [*simp*]: $COERCE(udom, 'a) = PRJ('a)$
<proof>

Cancellation rules

lemma *emb-coerce*:
 $DEFL('a) \sqsubseteq DEFL('b)$
 $\implies EMB('b) \cdot (COERCE('a, 'b) \cdot x) = EMB('a) \cdot x$
<proof>

lemma *coerce-prj*:
 $DEFL('a) \sqsubseteq DEFL('b)$
 $\implies COERCE('b, 'a) \cdot (PRJ('b) \cdot x) = PRJ('a) \cdot x$
<proof>

lemma *coerce-idem* [*simp*]:
 $DEFL('a) \sqsubseteq DEFL('b)$
 $\implies COERCE('b, 'c) \cdot (COERCE('a, 'b) \cdot x) = COERCE('a, 'c) \cdot x$
<proof>

2.2 More lemmas about *emb* and *prj*

lemma *prj-cast-DEFL* [*simp*]: $PRJ('a) \cdot (cast \cdot DEFL('a) \cdot x) = PRJ('a) \cdot x$
<proof>

lemma *cast-DEFL-emb* [*simp*]: $cast \cdot DEFL('a) \cdot (EMB('a) \cdot x) = EMB('a) \cdot x$
<proof>

$DEFL(udom)$

lemma *below-DEFL-udom* [*simp*]: $A \sqsubseteq DEFL(udom)$
<proof>

2.3 Coercing various datatypes

Coercing from the strict product type $'a \otimes 'b$ to another strict product type $'c \otimes 'd$ is equivalent to mapping the *coerce* function separately over each component using $sprod\text{-}map :: ('a \rightarrow 'c) \rightarrow ('b \rightarrow 'd) \rightarrow 'a \otimes 'b \rightarrow 'c \otimes 'd$. Each of the several type constructors defined in HOLCF satisfies a similar property, with respect to its own map combinator.

lemma *coerce-u*: $coerce = u\text{-}map \cdot coerce$
<proof>

lemma *coerce-sfun*: $coerce = sfun\text{-}map \cdot coerce \cdot coerce$
<proof>

lemma *coerce-cfun'*: $coerce = cfun\text{-}map \cdot coerce \cdot coerce$
<proof>

lemma *coerce-ssum*: $coerce = ssum\text{-}map \cdot coerce \cdot coerce$
<proof>

lemma *coerce-sprod*: $coerce = sprod\text{-}map \cdot coerce \cdot coerce$
<proof>

lemma *coerce-prod*: $coerce = prod\text{-}map \cdot coerce \cdot coerce$
<proof>

2.4 Simplifying coercions

When simplifying applications of the *coerce* function, rewrite rules are always oriented to replace *coerce* at complex types with other applications of *coerce* at simpler types.

The safest rewrite rules for *coerce* are given the *[simp]* attribute. For other rules that do not belong in the global simpset, we use dynamic theorem list called *coerce-simp*, which will collect additional rules for simplifying coercions.

named-theorems *coerce-simp rule for simplifying coercions*

The *coerce* function commutes with data constructors for various HOLCF datatypes.

lemma *coerce-up* *[simp]*: $coerce \cdot (up \cdot x) = up \cdot (coerce \cdot x)$
<proof>

lemma *coerce-sinl* *[simp]*: $coerce \cdot (sinl \cdot x) = sinl \cdot (coerce \cdot x)$
<proof>

lemma *coerce-sinr* [simp]: $\text{coerce} \cdot (\text{sinr} \cdot x) = \text{sinr} \cdot (\text{coerce} \cdot x)$
 ⟨proof⟩

lemma *coerce-spair* [simp]: $\text{coerce} \cdot (:x, y) = (:\text{coerce} \cdot x, \text{coerce} \cdot y)$
 ⟨proof⟩

lemma *coerce-Pair* [simp]: $\text{coerce} \cdot (x, y) = (\text{coerce} \cdot x, \text{coerce} \cdot y)$
 ⟨proof⟩

lemma *beta-coerce-cfun* [simp]: $\text{coerce} \cdot f \cdot x = \text{coerce} \cdot (f \cdot (\text{coerce} \cdot x))$
 ⟨proof⟩

lemma *coerce-cfun*: $\text{coerce} \cdot f = \text{coerce} \circ f \circ \text{coerce}$
 ⟨proof⟩

lemma *coerce-cfun-app* [coerce-simp]:
 $\text{coerce} \cdot f = (\lambda x. \text{coerce} \cdot (f \cdot (\text{coerce} \cdot x)))$
 ⟨proof⟩

end

3 Functor Class

theory *Functor*
imports *TypeApp Coerce*
keywords *tycondef* :: *thy-defn* **and** ·
begin

3.1 Class definition

Here we define the *functor* class, which models the Haskell class `Functor`. For technical reasons, we split the definition of *functor* into two separate classes: First, we introduce *prefunctor*, which only requires *fmap* to preserve the identity function, and not function composition.

The Haskell class `Functor f` fixes a polymorphic function $\text{fmap} :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$. Since functions in Isabelle type classes can only mention one type variable, we have the *prefunctor* class fix a function *fmapU* that fixes both of the polymorphic types to be the universal domain. We will use the coercion operator to recover a polymorphic *fmap*.

The single axiom of the *prefunctor* class is stated in terms of the HOLCF constant *isodefl*, which relates a function $f :: 'a \rightarrow 'a$ with a deflation $t :: \text{udom defl}$: $\text{isodefl } f\ t = (\text{cast} \cdot t = \text{EMB}('a) \circ f \circ \text{PRJ}('a))$.

class *prefunctor* = *tycon* +
fixes *fmapU* :: $(\text{udom} \rightarrow \text{udom}) \rightarrow \text{udom} \cdot 'a \rightarrow \text{udom} \cdot 'a :: \text{tycon}$
assumes *isodefl-fmapU*:

$isodefl (fmapU \cdot (cast \cdot t)) (TC('a::tycon) \cdot t)$

The *functor* class extends *prefunctor* with an axiom stating that *fmapU* preserves composition.

class *functor* = *prefunctor* +
assumes *fmapU-fmapU* [*coerce-simp*]:
 $\bigwedge f g (xs::udom \cdot 'a::tycon).$
 $fmapU \cdot f \cdot (fmapU \cdot g \cdot xs) = fmapU \cdot (\Lambda x. f \cdot (g \cdot x)) \cdot xs$

We define the polymorphic *fmap* by coercion from *fmapU*, then we proceed to derive the polymorphic versions of the functor laws.

definition *fmap* :: ('a → 'b) → 'a.'f → 'b.'f::*functor*
where *fmap* = *coerce* · (*fmapU* :: - → *udom*.'f → *udom*.'f)

3.2 Polymorphic functor laws

lemma *fmapU-eq-fmap*: *fmapU* = *fmap*
⟨*proof*⟩

lemma *fmap-eq-fmapU*: *fmap* = *fmapU*
⟨*proof*⟩

lemma *cast-TC*:
 $cast \cdot (TC('f) \cdot t) = emb \text{ oo } fmapU \cdot (cast \cdot t) \text{ oo } PRJ(udom \cdot 'f::prefunctor)$
⟨*proof*⟩

lemma *isodefl-cast*: *isodefl* (*cast* · *t*) *t*
⟨*proof*⟩

lemma *cast-cast-below1*: $A \sqsubseteq B \implies cast \cdot A \cdot (cast \cdot B \cdot x) = cast \cdot A \cdot x$
⟨*proof*⟩

lemma *cast-cast-below2*: $A \sqsubseteq B \implies cast \cdot B \cdot (cast \cdot A \cdot x) = cast \cdot A \cdot x$
⟨*proof*⟩

lemma *isodefl-fmap*:
assumes *isodefl d t*
shows *isodefl* (*fmap* · *d* :: 'a.'f → -) (*TC*('f::*functor*) · *t*)
⟨*proof*⟩

lemma *fmap-ID* [*simp*]: *fmap* · *ID* = *ID*
⟨*proof*⟩

lemma *fmap-ident* [*simp*]: *fmap* · ($\Lambda x. x$) = *ID*
⟨*proof*⟩

lemma *coerce-coerce-eq-fmapU-cast* [*coerce-simp*]:
fixes *xs* :: *udom*.'f::*functor*
shows $COERCE('a.'f, udom \cdot 'f) \cdot (COERCE(udom \cdot 'f, 'a.'f) \cdot xs) =$

$fmapU \cdot (cast \cdot DEFL('a)) \cdot xs$
 ⟨proof⟩

lemma *fmap-fmap*:

fixes $xs :: 'a \cdot 'f :: functor$ **and** $g :: 'a \rightarrow 'b$ **and** $f :: 'b \rightarrow 'c$
shows $fmap \cdot f \cdot (fmap \cdot g \cdot xs) = fmap \cdot (\Lambda x. f \cdot (g \cdot x)) \cdot xs$
 ⟨proof⟩

lemma *fmap-cfcomp*: $fmap \cdot (f \circ g) = fmap \cdot f \circ fmap \cdot g$
 ⟨proof⟩

3.3 Derived properties of *fmap*

Other theorems about *fmap* can be derived using only the abstract functor laws.

lemma *deflation-fmap*:

$deflation \ d \implies deflation \ (fmap \cdot d)$
 ⟨proof⟩

lemma *ep-pair-fmap*:

$ep\text{-}pair \ e \ p \implies ep\text{-}pair \ (fmap \cdot e) \ (fmap \cdot p)$
 ⟨proof⟩

lemma *fmap-strict*:

fixes $f :: 'a \rightarrow 'b$
assumes $f \cdot \perp = \perp$ **shows** $fmap \cdot f \cdot \perp = (\perp :: 'b \cdot 'f :: functor)$
 ⟨proof⟩

3.4 Proving that $fmap \cdot coerce = coerce$

lemma *fmapU-cast-eq*:

$fmapU \cdot (cast \cdot A) =$
 $PRJ(udom \cdot 'f) \circ cast \cdot (TC('f :: functor) \cdot A) \circ emb$
 ⟨proof⟩

lemma *fmapU-cast-DEFL*:

$fmapU \cdot (cast \cdot DEFL('a)) =$
 $PRJ(udom \cdot 'f) \circ cast \cdot DEFL('a \cdot 'f :: functor) \circ emb$
 ⟨proof⟩

lemma *coerce-functor*: $COERCE('a \cdot 'f, 'b \cdot 'f :: functor) = fmap \cdot coerce$
 ⟨proof⟩

3.5 Lemmas for reasoning about coercion

lemma *fmapU-cast-coerce* [*coerce-simp*]:

fixes $m :: 'a \cdot 'f :: functor$
shows $fmapU \cdot (cast \cdot DEFL('a)) \cdot (COERCE('a \cdot 'f, udom \cdot 'f) \cdot m) =$
 $COERCE('a \cdot 'f, udom \cdot 'f) \cdot m$

<proof>

```
lemma coerce-fmap [coerce-simp]:  
  fixes xs :: 'a.'f::functor and f :: 'a → 'b  
  shows COERCE('b.'f, 'c.'f).(fmap.f.xs) = fmap.(λ x. COERCE('b,'c).(f.x)).xs  
<proof>
```

```
lemma fmap-coerce [coerce-simp]:  
  fixes xs :: 'a.'f::functor and f :: 'b → 'c  
  shows fmap.f.(COERCE('a.'f, 'b.'f).xs) = fmap.(λ x. f.(COERCE('a,'b).x)).xs  
<proof>
```

3.6 Configuration of Domain package

We make various theorem declarations to enable Domain package definitions that involve *tycon* application.

<ML>

```
declare DEFL-app [domain-defl-simps]  
declare fmap-ID [domain-map-ID]  
declare deflation-fmap [domain-deflation]  
declare isodeft-fmap [domain-isodeft]
```

3.7 Configuration of the Tycon package

We now set up a new type definition command, which is used for defining new *tycon* instances. The *tycondef* command is implemented using much of the same code as the Domain package, and supports a similar input syntax. It automatically generates a *prefunctor* instance for each new type. (The user must provide a proof of the composition law to obtain a *functor* class instance.)

<ML>

end

4 Monad Class

```
theory Monad  
imports Functor  
begin
```

4.1 Class definition

In Haskell, class *Monad* is defined as follows:

```

class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

```

We formalize class *monad* in a manner similar to the *functor* class: We fix monomorphic versions of the class constants, replacing type variables with *udom*, and assume monomorphic versions of the class axioms.

Because the monad laws imply the composition rule for *fmap*, we declare *prefunctor* as the superclass, and separately prove a subclass relationship with *functor*.

```

class monad = prefunctor +
  fixes returnU :: udom -> udom.'a::tycon
  fixes bindU :: udom.'a -> (udom -> udom.'a) -> udom.'a
  assumes fmapU-eq-bindU:
     $\bigwedge xs. \text{fmapU} \cdot f \cdot xs = \text{bindU} \cdot xs \cdot (\lambda x. \text{returnU} \cdot (f \cdot x))$ 
  assumes bindU-returnU:
     $\bigwedge f x. \text{bindU} \cdot (\text{returnU} \cdot x) \cdot f = f \cdot x$ 
  assumes bindU-bindU:
     $\bigwedge xs f g. \text{bindU} \cdot (\text{bindU} \cdot xs \cdot f) \cdot g = \text{bindU} \cdot xs \cdot (\lambda x. \text{bindU} \cdot (f \cdot x) \cdot g)$ 

```

```

instance monad  $\subseteq$  functor
<proof>

```

As with *fmap*, we define the polymorphic *return* and *bind* by coercion from the monomorphic *returnU* and *bindU*.

```

definition return :: 'a -> 'a.'m::monad
  where return = coerce.(returnU :: udom -> udom.'m)

```

```

definition bind :: 'a.'m::monad -> ('a -> 'b.'m) -> 'b.'m
  where bind = coerce.(bindU :: udom.'m -> -)

```

```

abbreviation bind-syn :: 'a.'m::monad  $\Rightarrow$  ('a -> 'b.'m)  $\Rightarrow$  'b.'m (infixl  $\gg=$  55)
  where m  $\gg=$  f  $\equiv$  bind.m.f

```

4.2 Naturality of bind and return

The three class axioms imply naturality properties of *returnU* and *bindU*, i.e., that both commute with *fmapU*.

```

lemma fmapU-returnU [coerce-simp]:
  fmapU.f.(returnU.x) = returnU.(f.x)
<proof>

```

```

lemma fmapU-bindU [coerce-simp]:
  fmapU.f.(bindU.m.k) = bindU.m.( $\lambda x. \text{fmapU} \cdot f \cdot (k \cdot x)$ )
<proof>

```

lemma *bindU-fmapU*:
 $bindU \cdot (fmapU \cdot f \cdot xs) \cdot k = bindU \cdot xs \cdot (\Lambda x. k \cdot (f \cdot x))$
 ⟨proof⟩

4.3 Polymorphic versions of class assumptions

lemma *monad-fmap*:
 fixes $xs :: 'a \cdot 'm :: monad$ and $f :: 'a \rightarrow 'b$
 shows $fmap \cdot f \cdot xs = xs \ggg (\Lambda x. return \cdot (f \cdot x))$
 ⟨proof⟩

lemma *monad-left-unit [simp]*: $(return \cdot x \ggg f) = (f \cdot x)$
 ⟨proof⟩

lemma *bind-bind*:
 fixes $m :: 'a \cdot 'm :: monad$
 shows $((m \ggg f) \ggg g) = (m \ggg (\Lambda x. f \cdot x \ggg g))$
 ⟨proof⟩

4.4 Derived rules

The following properties can be derived using only the abstract monad laws.

lemma *monad-right-unit [simp]*: $(m \ggg return) = m$
 ⟨proof⟩

lemma *fmap-return*: $fmap \cdot f \cdot (return \cdot x) = return \cdot (f \cdot x)$
 ⟨proof⟩

lemma *fmap-bind*: $fmap \cdot f \cdot (bind \cdot xs \cdot k) = bind \cdot xs \cdot (\Lambda x. fmap \cdot f \cdot (k \cdot x))$
 ⟨proof⟩

lemma *bind-fmap*: $bind \cdot (fmap \cdot f \cdot xs) \cdot k = bind \cdot xs \cdot (\Lambda x. k \cdot (f \cdot x))$
 ⟨proof⟩

Bind is strict in its first argument, if its second argument is a strict function.

lemma *bind-strict*:
 assumes $k \cdot \perp = \perp$ shows $\perp \ggg k = \perp$
 ⟨proof⟩

lemma *congruent-bind*:
 $(\forall m. m \ggg k1 = m \ggg k2) = (k1 = k2)$
 ⟨proof⟩

4.5 Laws for join

definition *join* :: $('a \cdot 'm) \cdot 'm \rightarrow 'a \cdot 'm :: monad$
 where $join \equiv \Lambda m. m \ggg (\Lambda x. x)$

lemma *join-fmap-fmap*: $join \cdot (fmap \cdot (fmap \cdot f) \cdot xss) = fmap \cdot f \cdot (join \cdot xss)$
 ⟨proof⟩

lemma *join-return*: $join \cdot (return \cdot xs) = xs$
 ⟨proof⟩

lemma *join-fmap-return*: $join \cdot (fmap \cdot return \cdot xs) = xs$
 ⟨proof⟩

lemma *join-fmap-join*: $join \cdot (fmap \cdot join \cdot xsss) = join \cdot (join \cdot xsss)$
 ⟨proof⟩

lemma *bind-def2*: $m \gg= k = join \cdot (fmap \cdot k \cdot m)$
 ⟨proof⟩

4.6 Equivalence of monad laws and fmap/join laws

lemma $(return \cdot x \gg= f) = (f \cdot x)$
 ⟨proof⟩

lemma $(m \gg= return) = m$
 ⟨proof⟩

lemma $((m \gg= f) \gg= g) = (m \gg= (\lambda x. f \cdot x \gg= g))$
 ⟨proof⟩

4.7 Simplification of coercions

We configure rewrite rules that push coercions inwards, and reduce them to coercions on simpler types.

lemma *coerce-return* [*coerce-simp*]:
 $COERCE('a.'m, 'b.'m :: monad) \cdot (return \cdot x) = return \cdot (COERCE('a, 'b) \cdot x)$
 ⟨proof⟩

lemma *coerce-bind* [*coerce-simp*]:
fixes $m :: 'a.'m :: monad$ **and** $k :: 'a \rightarrow 'b.'m$
shows $COERCE('b.'m, 'c.'m) \cdot (m \gg= k) = m \gg= (\lambda x. COERCE('b.'m, 'c.'m) \cdot (k \cdot x))$
 ⟨proof⟩

lemma *bind-coerce* [*coerce-simp*]:
fixes $m :: 'a.'m :: monad$ **and** $k :: 'b \rightarrow 'c.'m$
shows $COERCE('a.'m, 'b.'m) \cdot m \gg= k = m \gg= (\lambda x. k \cdot (COERCE('a, 'b) \cdot x))$
 ⟨proof⟩

end

5 Monad-Zero Class

```
theory Monad-Zero
imports Monad
begin

class zeroU = tycon +
  fixes zeroU :: udom·'a::tycon

class functor-zero = zeroU + functor +
  assumes fmapU-zeroU [coerce-simp]:
    fmapU·f·zeroU = zeroU

class monad-zero = zeroU + monad +
  assumes bindU-zeroU:
    bindU·zeroU·f = zeroU

instance monad-zero  $\subseteq$  functor-zero
  ⟨proof⟩

definition fzero :: 'a·'f::functor-zero
  where fzero = coerce·(zeroU :: udom·'f)

lemma fmap-fzero:
  fmap·f·(fzero :: 'a·'f::functor-zero) = (fzero :: 'b·'f)
  ⟨proof⟩

abbreviation mzero :: 'a·'m::monad-zero
  where mzero  $\equiv$  fzero

lemmas mzero-def = fzero-def [where 'f'=m::monad-zero] for f
lemmas fmap-mzero = fmap-fzero [where 'f'=m::monad-zero] for f

lemma bindU-eq-bind: bindU = bind
  ⟨proof⟩

lemma bind-mzero:
  bind·(fzero :: 'a·'m::monad-zero)·k = (mzero :: 'b·'m)
  ⟨proof⟩

end
```

6 Monad-Plus Class

```
theory Monad-Plus
imports Monad
begin

hide-const (open) Fixrec.mplus
```



```

class plusU = tycon +
  fixes plusU :: udom.'a → udom.'a → udom.'a::tycon

class functor-plus = plusU + functor +
  assumes fmapU-plusU [coerce-simp]:
    fmapU.f.(plusU.a.b) = plusU.(fmapU.f.a).(fmapU.f.b)
  assumes plusU-assoc:
    plusU.(plusU.a.b).c = plusU.a.(plusU.b.c)

class monad-plus = plusU + monad +
  assumes bindU-plusU:
    bindU.(plusU.xs.ys).k = plusU.(bindU.xs.k).(bindU.ys.k)
  assumes plusU-assoc':
    plusU.(plusU.a.b).c = plusU.a.(plusU.b.c)

instance monad-plus ⊆ functor-plus
⟨proof⟩

definition fplus :: 'a.'f::functor-plus → 'a.'f → 'a.'f
  where fplus = coerce.(plusU :: udom.'f → -)

lemma fmap-fplus:
  fixes f :: 'a → 'b and a b :: 'a.'f::functor-plus
  shows fmap.f.(fplus.a.b) = fplus.(fmap.f.a).(fmap.f.b)
⟨proof⟩

lemma fplus-assoc:
  fixes a b c :: 'a.'f::functor-plus
  shows fplus.(fplus.a.b).c = fplus.a.(fplus.b.c)
⟨proof⟩

abbreviation mplus :: 'a.'m::monad-plus → 'a.'m → 'a.'m
  where mplus ≡ fplus

lemmas mplus-def = fplus-def [where 'f='m::monad-plus for f]
lemmas fmap-mplus = fmap-fplus [where 'f='m::monad-plus for f]
lemmas mplus-assoc = fplus-assoc [where 'f='m::monad-plus for f]

lemma bind-mplus:
  fixes a b :: 'a.'m::monad-plus
  shows bind.(mplus.a.b).k = mplus.(bind.a.k).(bind.b.k)
⟨proof⟩

lemma join-mplus:
  fixes xss yss :: ('a.'m).m::monad-plus
  shows join.(mplus.xss.yss) = mplus.(join.xss).(join.yss)
⟨proof⟩

```

end

7 Monad-Zero-Plus Class

```
theory Monad-Zero-Plus
imports Monad-Zero Monad-Plus
begin
```

```
hide-const (open) Fixrec.mplus
```

```
class functor-zero-plus = functor-zero + functor-plus +
  assumes plusU-zeroU-left:
    plusU·zeroU·m = m
  assumes plusU-zeroU-right:
    plusU·m·zeroU = m
```

```
class monad-zero-plus = monad-zero + monad-plus + functor-zero-plus
```

```
lemma fplus-fzero-left:
  fixes m :: 'a.'f::functor-zero-plus
  shows fplus·fzero·m = m
⟨proof⟩
```

```
lemma fplus-fzero-right:
  fixes m :: 'a.'f::functor-zero-plus
  shows fplus·m·fzero = m
⟨proof⟩
```

```
lemmas mplus-mzero-left =
  fplus-fzero-left [where 'f='m::monad-zero-plus] for f
```

```
lemmas mplus-mzero-right =
  fplus-fzero-right [where 'f='m::monad-zero-plus] for f
```

end

8 Lazy list monad

```
theory Lazy-List-Monad
imports Monad-Zero-Plus
begin
```

To illustrate the general process of defining a new type constructor, we formalize the datatype of lazy lists. Below are the Haskell datatype definition and class instances.

```
data List a = Nil | Cons a (List a)
```

```

instance Functor List where
  fmap f Nil = Nil
  fmap f (Cons x xs) = Cons (f x) (fmap f xs)

instance Monad List where
  return x      = Cons x Nil
  Nil          >>= k = Nil
  Cons x xs >>= k = mplus (k x) (xs >>= k)

instance MonadZero List where
  mzero = Nil

instance MonadPlus List where
  mplus Nil      ys = ys
  mplus (Cons x xs) ys = Cons x (mplus xs ys)

```

8.1 Type definition

The first step is to register the datatype definition with *tycondef*.

```
tycondef 'a·llist = LNil | LCons (lazy 'a) (lazy 'a·llist)
```

The *tycondef* command generates lots of theorems automatically, but there are a few more involving *coerce* and *fmapU* that we still need to prove manually. These proofs could be automated in a later version of *tycondef*.

```
lemma coerce-llist-abs [simp]: coerce·(llist-abs·x) = llist-abs·(coerce·x)
⟨proof⟩
```

```
lemma coerce-LNil [simp]: coerce·LNil = LNil
⟨proof⟩
```

```
lemma coerce-LCons [simp]: coerce·(LCons·x·xs) = LCons·(coerce·x)·(coerce·xs)
⟨proof⟩
```

```
lemma fmapU-llist-simps [simp]:
  fmapU·f·(⊥::udom·llist) = ⊥
  fmapU·f·LNil = LNil
  fmapU·f·(LCons·x·xs) = LCons·(f·x)·(fmapU·f·xs)
⟨proof⟩
```

8.2 Class instances

The *tycondef* command defines *fmapU* for us and proves a *prefunctor* class instance automatically. For the *functor* instance we only need to prove the composition law, which we can do by induction.

```
instance llist :: functor
⟨proof⟩
```

For the other class instances, we need to provide definitions for a few constants: $returnU$, $bindU$, $zeroU$, and $plusU$. We can use ordinary commands like *definition* and *fixrec* for this purpose. Finally we prove the class axioms, along with a few helper lemmas, using ordinary proof procedures like *induction*.

instantiation $l\text{list} :: \text{monad-zero-plus}$
begin

fixrec $plusU\text{-l\text{list}} :: \text{u\text{dom}\cdot\text{l\text{list}}} \rightarrow \text{u\text{dom}\cdot\text{l\text{list}}} \rightarrow \text{u\text{dom}\cdot\text{l\text{list}}}$
where $plusU\text{-l\text{list}}\cdot LNil\cdot ys = ys$
 $| plusU\text{-l\text{list}}\cdot (LCons\cdot x\cdot xs)\cdot ys = LCons\cdot x\cdot (plusU\text{-l\text{list}}\cdot xs\cdot ys)$

lemma $plusU\text{-l\text{list-strict}}$ [*simp*]: $plusU\cdot \perp\cdot ys = (\perp :: \text{u\text{dom}\cdot\text{l\text{list}}})$
 $\langle \text{proof} \rangle$

fixrec $bindU\text{-l\text{list}} :: \text{u\text{dom}\cdot\text{l\text{list}}} \rightarrow (\text{u\text{dom}} \rightarrow \text{u\text{dom}\cdot\text{l\text{list}}}) \rightarrow \text{u\text{dom}\cdot\text{l\text{list}}}$
where $bindU\text{-l\text{list}}\cdot LNil\cdot k = LNil$
 $| bindU\text{-l\text{list}}\cdot (LCons\cdot x\cdot xs)\cdot k = plusU\cdot (k\cdot x)\cdot (bindU\text{-l\text{list}}\cdot xs\cdot k)$

lemma $bindU\text{-l\text{list-strict}}$ [*simp*]: $bindU\cdot \perp\cdot k = (\perp :: \text{u\text{dom}\cdot\text{l\text{list}}})$
 $\langle \text{proof} \rangle$

definition $zeroU\text{-l\text{list-def}}$:
 $zeroU = LNil$

definition $returnU\text{-l\text{list-def}}$:
 $returnU = (\lambda x. LCons\cdot x\cdot LNil)$

lemma $plusU\text{-LNil-right}$: $plusU\cdot xs\cdot LNil = xs$
 $\langle \text{proof} \rangle$

lemma $plusU\text{-l\text{list-assoc}}$:
fixes $xs\ ys\ zs :: \text{u\text{dom}\cdot\text{l\text{list}}}$
shows $plusU\cdot (plusU\cdot xs\cdot ys)\cdot zs = plusU\cdot xs\cdot (plusU\cdot ys\cdot zs)$
 $\langle \text{proof} \rangle$

lemma $bindU\text{-plusU-l\text{list}}$:
fixes $xs\ ys :: \text{u\text{dom}\cdot\text{l\text{list}}}$ **shows**
 $bindU\cdot (plusU\cdot xs\cdot ys)\cdot f = plusU\cdot (bindU\cdot xs\cdot f)\cdot (bindU\cdot ys\cdot f)$
 $\langle \text{proof} \rangle$

instance $\langle \text{proof} \rangle$

end

8.3 Transfer properties to polymorphic versions

After proving the class instances, there is still one more step: We must transfer all the list-specific lemmas about the monomorphic constants (e.g., $fmapU$ and $bindU$) to the corresponding polymorphic constants ($fmap$ and $bind$). These lemmas primarily consist of the defining equations for each constant. The polymorphic constants are defined using $coerce$, so the proofs proceed by unfolding the definitions and simplifying with the $coerce-simp$ rules.

lemma *fmap-llist-simps* [simp]:
 $fmap \cdot f \cdot (\perp :: 'a \cdot llist) = \perp$
 $fmap \cdot f \cdot LNil = LNil$
 $fmap \cdot f \cdot (LCons \cdot x \cdot xs) = LCons \cdot (f \cdot x) \cdot (fmap \cdot f \cdot xs)$
(proof)

lemma *mplus-llist-simps* [simp]:
 $mplus \cdot (\perp :: 'a \cdot llist) \cdot ys = \perp$
 $mplus \cdot LNil \cdot ys = ys$
 $mplus \cdot (LCons \cdot x \cdot xs) \cdot ys = LCons \cdot x \cdot (mplus \cdot xs \cdot ys)$
(proof)

lemma *bind-llist-simps* [simp]:
 $bind \cdot (\perp :: 'a \cdot llist) \cdot f = \perp$
 $bind \cdot LNil \cdot f = LNil$
 $bind \cdot (LCons \cdot x \cdot xs) \cdot f = mplus \cdot (f \cdot x) \cdot (bind \cdot xs \cdot f)$
(proof)

lemma *return-llist-def*:
 $return = (\lambda x. LCons \cdot x \cdot LNil)$
(proof)

lemma *mzero-llist-def*:
 $mzero = LNil$
(proof)

lemma *join-llist-simps* [simp]:
 $join \cdot (\perp :: 'a \cdot llist \cdot llist) = \perp$
 $join \cdot LNil = LNil$
 $join \cdot (LCons \cdot xs \cdot xss) = mplus \cdot xs \cdot (join \cdot xss)$
(proof)

end

9 Maybe monad

theory *Maybe-Monad*
imports *Monad-Zero-Plus*
begin

9.1 Type definition

tycondef $'a \cdot \text{maybe} = \text{Nothing} \mid \text{Just } (\text{lazy } 'a)$

lemma *coerce-maybe-abs* [simp]: $\text{coerce} \cdot (\text{maybe-abs } x) = \text{maybe-abs} \cdot (\text{coerce } x)$
(proof)

lemma *coerce-Nothing* [simp]: $\text{coerce} \cdot \text{Nothing} = \text{Nothing}$
(proof)

lemma *coerce-Just* [simp]: $\text{coerce} \cdot (\text{Just } x) = \text{Just} \cdot (\text{coerce } x)$
(proof)

lemma *fmapU-maybe-simps* [simp]:
 $\text{fmapU} \cdot f \cdot (\perp :: \text{udom} \cdot \text{maybe}) = \perp$
 $\text{fmapU} \cdot f \cdot \text{Nothing} = \text{Nothing}$
 $\text{fmapU} \cdot f \cdot (\text{Just } x) = \text{Just} \cdot (f \cdot x)$
(proof)

9.2 Class instance proofs

instance *maybe* :: *functor*
(proof)

instantiation *maybe* :: {*functor-zero-plus*, *monad-zero*}
begin

fixrec *plusU-maybe* :: $\text{udom} \cdot \text{maybe} \rightarrow \text{udom} \cdot \text{maybe} \rightarrow \text{udom} \cdot \text{maybe}$
where $\text{plusU-maybe} \cdot \text{Nothing} \cdot \text{ys} = \text{ys}$
| $\text{plusU-maybe} \cdot (\text{Just } x) \cdot \text{ys} = \text{Just } x$

lemma *plusU-maybe-strict* [simp]: $\text{plusU} \cdot \perp \cdot \text{ys} = (\perp :: \text{udom} \cdot \text{maybe})$
(proof)

fixrec *bindU-maybe* :: $\text{udom} \cdot \text{maybe} \rightarrow (\text{udom} \rightarrow \text{udom} \cdot \text{maybe}) \rightarrow \text{udom} \cdot \text{maybe}$
where $\text{bindU-maybe} \cdot \text{Nothing} \cdot k = \text{Nothing}$
| $\text{bindU-maybe} \cdot (\text{Just } x) \cdot k = k \cdot x$

lemma *bindU-maybe-strict* [simp]: $\text{bindU} \cdot \perp \cdot k = (\perp :: \text{udom} \cdot \text{maybe})$
(proof)

definition *zeroU-maybe-def*:
 $\text{zeroU} = \text{Nothing}$

definition *returnU-maybe-def*:
 $\text{returnU} = \text{Just}$

lemma *plusU-Nothing-right*: $\text{plusU} \cdot \text{xs} \cdot \text{Nothing} = \text{xs}$
(proof)

lemma *bindU-plusU-maybe*:
fixes *xs ys* :: *udom*.*maybe* **shows**
 $bindU \cdot (plusU \cdot xs \cdot ys) \cdot f = plusU \cdot (bindU \cdot xs \cdot f) \cdot (bindU \cdot ys \cdot f)$
⟨*proof*⟩

instance ⟨*proof*⟩

end

9.3 Transfer properties to polymorphic versions

lemma *fmap-maybe-simps* [*simp*]:
 $fmap \cdot f \cdot (\perp :: 'a \cdot maybe) = \perp$
 $fmap \cdot f \cdot Nothing = Nothing$
 $fmap \cdot f \cdot (Just \cdot x) = Just \cdot (f \cdot x)$
⟨*proof*⟩

lemma *fplus-maybe-simps* [*simp*]:
 $fplus \cdot (\perp :: 'a \cdot maybe) \cdot ys = \perp$
 $fplus \cdot Nothing \cdot ys = ys$
 $fplus \cdot (Just \cdot x) \cdot ys = Just \cdot x$
⟨*proof*⟩

lemma *fplus-Nothing-right* [*simp*]:
 $fplus \cdot m \cdot Nothing = m$
⟨*proof*⟩

lemma *bind-maybe-simps* [*simp*]:
 $bind \cdot (\perp :: 'a \cdot maybe) \cdot f = \perp$
 $bind \cdot Nothing \cdot f = Nothing$
 $bind \cdot (Just \cdot x) \cdot f = f \cdot x$
⟨*proof*⟩

lemma *return-maybe-def*: $return = Just$
⟨*proof*⟩

lemma *mzero-maybe-def*: $mzero = Nothing$
⟨*proof*⟩

lemma *join-maybe-simps* [*simp*]:
 $join \cdot (\perp :: 'a \cdot maybe \cdot maybe) = \perp$
 $join \cdot Nothing = Nothing$
 $join \cdot (Just \cdot xs) = xs$
⟨*proof*⟩

9.4 Maybe is not in *monad-plus*

The *maybe* type does not satisfy the law *bind-mplus*.

lemma *maybe-counterexample1*:

$$\llbracket a = \text{Just}\cdot x; b = \perp; k\cdot x = \text{Nothing} \rrbracket$$

$$\implies \text{fplus}\cdot a\cdot b \gg= k \neq \text{fplus}\cdot (a \gg= k)\cdot (b \gg= k)$$
 <proof>

lemma *maybe-counterexample2*:

$$\llbracket a = \text{Just}\cdot x; b = \text{Just}\cdot y; k\cdot x = \text{Nothing}; k\cdot y = \text{Just}\cdot z \rrbracket$$

$$\implies \text{fplus}\cdot a\cdot b \gg= k \neq \text{fplus}\cdot (a \gg= k)\cdot (b \gg= k)$$
 <proof>

end

10 Error monad

theory *Error-Monad*
imports *Monad-Plus*
begin

10.1 Type definition

tycondef *'a' error* = *Err (lazy 'e) | Ok (lazy 'a)*

lemma *coerce-error-abs [simp]*: *coerce.(error-abs.x) = error-abs.(coerce.x)*
 <proof>

lemma *coerce-Err [simp]*: *coerce.(Err.x) = Err.(coerce.x)*
 <proof>

lemma *coerce-Ok [simp]*: *coerce.(Ok.m) = Ok.(coerce.m)*
 <proof>

lemma *fmapU-error-simps [simp]*:

$$\text{fmapU}\cdot f\cdot (\perp :: \text{udom}\cdot 'a \text{ error}) = \perp$$

$$\text{fmapU}\cdot f\cdot (\text{Err}\cdot e) = \text{Err}\cdot e$$

$$\text{fmapU}\cdot f\cdot (\text{Ok}\cdot x) = \text{Ok}\cdot (f\cdot x)$$
 <proof>

10.2 Monad class instance

instantiation *error* :: (*domain*) {*monad, functor-plus*}
begin

definition
returnU = *Ok*

fixrec *bindU-error* :: *udom' a error* → (*udom* → *udom' a error*) → *udom' a error*
where *bindU-error.(Err.e).f = Err.e*
 | *bindU-error.(Ok.x).f = f.x*

lemma *bindU-error-strict [simp]*: *bindU.⊥.k = (⊥ :: udom' a error)*

<proof>

fixrec *plusU-error* :: *udom* · 'a error → *udom* · 'a error → *udom* · 'a error
 where *plusU-error* · (Err · e) · f = f
 | *plusU-error* · (Ok · x) · f = Ok · x

lemma *plusU-error-strict* [*simp*]: *plusU* · (⊥ :: *udom* · 'a error) = ⊥
<proof>

instance *<proof>*

end

10.3 Transfer properties to polymorphic versions

lemma *fmap-error-simps* [*simp*]:
 fmap · f · (⊥ :: 'a · 'e error) = ⊥
 fmap · f · (Err · e :: 'a · 'e error) = Err · e
 fmap · f · (Ok · x :: 'a · 'e error) = Ok · (f · x)
<proof>

lemma *return-error-def*: *return* = Ok
<proof>

lemma *bind-error-simps* [*simp*]:
 bind · (⊥ :: 'a · 'e error) · f = ⊥
 bind · (Err · e :: 'a · 'e error) · f = Err · e
 bind · (Ok · x :: 'a · 'e error) · f = f · x
<proof>

lemma *join-error-simps* [*simp*]:
 join · ⊥ = (⊥ :: 'a · 'e error)
 join · (Err · e) = Err · e
 join · (Ok · x) = x
<proof>

lemma *fplus-error-simps* [*simp*]:
 fplus · ⊥ · r = (⊥ :: 'a · 'e error)
 fplus · (Err · e) · r = r
 fplus · (Ok · x) · r = Ok · x
<proof>

end

11 Writer monad

theory *Writer-Monad*
imports *Monad*
begin

11.1 Monoid class

```
class monoid = domain +
  fixes mempty :: 'a
  fixes mappend :: 'a → 'a → 'a
  assumes mempty-left:  $\bigwedge ys. mappend mempty ys = ys$ 
  assumes mempty-right:  $\bigwedge xs. mappend xs mempty = xs$ 
  assumes mappend-assoc:
     $\bigwedge xs\ ys\ zs. mappend (mappend xs ys) zs = mappend xs (mappend ys zs)$ 
```

11.2 Writer monad type

Below is the standard Haskell definition of a writer monad type; it is an isomorphic copy of the lazy pair type (a, w) .

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

Since HOLCF does not have a pre-defined lazy pair type, we will base this formalization on an equivalent, more direct definition:

```
data Writer w a = Writer w a
```

We can directly translate the above Haskell type definition using *tycondef*.

```
tycondef 'a.'w writer = Writer (lazy 'w) (lazy 'a)
```

```
lemma coerce-writer-abs [simp]: coerce (writer-abs x) = writer-abs (coerce x)
⟨proof⟩
```

```
lemma coerce-Writer [simp]:
  coerce (Writer w x) = Writer (coerce w) (coerce x)
⟨proof⟩
```

```
lemma fmapU-writer-simps [simp]:
  fmapU f (⊥ :: udom.'w writer) = ⊥
  fmapU f (Writer w x) = Writer w (f x)
⟨proof⟩
```

11.3 Class instance proofs

```
instance writer :: (domain) functor
⟨proof⟩
```

```
instantiation writer :: (monoid) monad
begin
```

```
fixrec bindU-writer ::
  udom.'a writer → (udom → udom.'a writer) → udom.'a writer
```

where $bindU\text{-}writer.(Writer.w.x).f =$
 $(case\ f.x\ of\ Writer.w'.y \Rightarrow Writer.(mappend.w.w').y)$

lemma $bindU\text{-}writer\text{-}strict$ [simp]: $bindU.\perp.k = (\perp :: udom.'a\ writer)$
 ⟨proof⟩

definition
 $returnU = Writer.empty$

instance ⟨proof⟩

end

11.4 Transfer properties to polymorphic versions

lemma $fmap\text{-}writer\text{-}simps$ [simp]:
 $fmap.f.(\perp :: 'a.'w\ writer) = \perp$
 $fmap.f.(Writer.w.x :: 'a.'w\ writer) = Writer.w.(f.x)$
 ⟨proof⟩

lemma $return\text{-}writer\text{-}def$: $return = Writer.empty$
 ⟨proof⟩

lemma $bind\text{-}writer\text{-}simps$ [simp]:
 $bind.(\perp :: 'a.'w::monoid\ writer).f = \perp$
 $bind.(Writer.w.x :: 'a.'w::monoid\ writer).k =$
 $(case\ k.x\ of\ Writer.w'.y \Rightarrow Writer.(mappend.w.w').y)$
 ⟨proof⟩

lemma $join\text{-}writer\text{-}simps$ [simp]:
 $join.\perp = (\perp :: 'a.'w::monoid\ writer)$
 $join.(Writer.w.(Writer.w'.x)) = Writer.(mappend.w.w').x$
 ⟨proof⟩

11.5 Extra operations

definition $tell :: 'w \rightarrow unit.('w::monoid\ writer)$
where $tell = (\lambda\ w.\ Writer.w.())$

end

12 Binary tree monad

theory $Binary\text{-}Tree\text{-}Monad$
imports $Monad$
begin

12.1 Type definition

tycondef 'a.btree =
 Leaf (lazy 'a) | Node (lazy 'a.btree) (lazy 'a.btree)

lemma *coerce-btree-abs* [simp]: $coerce.(btree-abs.x) = btree-abs.(coerce.x)$
⟨proof⟩

lemma *coerce-Leaf* [simp]: $coerce.(Leaf.x) = Leaf.(coerce.x)$
⟨proof⟩

lemma *coerce-Node* [simp]: $coerce.(Node.xs.ys) = Node.(coerce.xs).(coerce.ys)$
⟨proof⟩

lemma *fmapU-btree-simps* [simp]:
 $fmapU.f.(⊥::udom.btree) = ⊥$
 $fmapU.f.(Leaf.x) = Leaf.(f.x)$
 $fmapU.f.(Node.xs.ys) = Node.(fmapU.f.xs).(fmapU.f.ys)$
⟨proof⟩

12.2 Class instance proofs

instance *btree* :: *functor*
⟨proof⟩

instantiation *btree* :: *monad*
begin

definition
 $returnU = Leaf$

fixrec *bindU-btree* :: $udom.btree \rightarrow (udom \rightarrow udom.btree) \rightarrow udom.btree$
 where $bindU-btree.(Leaf.x).k = k.x$
 | $bindU-btree.(Node.xs.ys).k =$
 $Node.(bindU-btree.xs.k).(bindU-btree.ys.k)$

lemma *bindU-btree-strict* [simp]: $bindU.⊥.k = (⊥::udom.btree)$
⟨proof⟩

instance ⟨proof⟩

end

12.3 Transfer properties to polymorphic versions

lemma *fmap-btree-simps* [simp]:
 $fmap.f.(⊥::'a.btree) = ⊥$
 $fmap.f.(Leaf.x) = Leaf.(f.x)$
 $fmap.f.(Node.xs.ys) = Node.(fmap.f.xs).(fmap.f.ys)$
⟨proof⟩

lemma *bind-btree-simps* [*simp*]:
 $bind.(⊥::'a.btree).k = ⊥$
 $bind.(Leaf.x).k = k.x$
 $bind.(Node.xs.ys).k = Node.(bind.xs.k).(bind.ys.k)$
 ⟨*proof*⟩

lemma *return-btree-def*:
 $return = Leaf$
 ⟨*proof*⟩

lemma *join-btree-simps* [*simp*]:
 $join.(⊥::'a.btree.btree) = ⊥$
 $join.(Leaf.xs) = xs$
 $join.(Node.xss.yss) = Node.(join.xss).(join.yss)$
 ⟨*proof*⟩

end

13 Lift monad

theory *Lift-Monad*
imports *Monad*
begin

13.1 Type definition

tycondef *'a.lifted = Lifted* (**lazy** 'a)

lemma *coerce-lifted-abs* [*simp*]: $coerce.(lifted-abs.x) = lifted-abs.(coerce.x)$
 ⟨*proof*⟩

lemma *coerce-Lifted* [*simp*]: $coerce.(Lifted.x) = Lifted.(coerce.x)$
 ⟨*proof*⟩

lemma *fmapU-lifted-simps* [*simp*]:
 $fmapU.f.(⊥::udom.lifted) = ⊥$
 $fmapU.f.(Lifted.x) = Lifted.(f.x)$
 ⟨*proof*⟩

13.2 Class instance proofs

instance *lifted :: functor*
 ⟨*proof*⟩

instantiation *lifted :: monad*
begin

fixrec *bindU-lifted :: udom.lifted → (udom → udom.lifted) → udom.lifted*

where $bindU\text{-lifted}\cdot(Lifted\cdot x)\cdot k = k\cdot x$

lemma $bindU\text{-lifted-strict}$ [simp]: $bindU\cdot\perp\cdot k = (\perp::udom\cdot lifted)$
 $\langle proof \rangle$

definition $returnU\text{-lifted-def}$:
 $returnU = Lifted$

instance $\langle proof \rangle$

end

13.3 Transfer properties to polymorphic versions

lemma $fmap\text{-lifted-simps}$ [simp]:
 $fmap\cdot f\cdot(\perp::'a\cdot lifted) = \perp$
 $fmap\cdot f\cdot(Lifted\cdot x) = Lifted\cdot(f\cdot x)$
 $\langle proof \rangle$

lemma $bind\text{-lifted-simps}$ [simp]:
 $bind\cdot(\perp::'a\cdot lifted)\cdot f = \perp$
 $bind\cdot(Lifted\cdot x)\cdot f = f\cdot x$
 $\langle proof \rangle$

lemma $return\text{-lifted-def}$: $return = Lifted$
 $\langle proof \rangle$

lemma $join\text{-lifted-simps}$ [simp]:
 $join\cdot(\perp::'a\cdot lifted\cdot lifted) = \perp$
 $join\cdot(Lifted\cdot xs) = xs$
 $\langle proof \rangle$

end

14 Resumption monad transformer

theory $Resumption\text{-Transformer}$
imports $Monad\text{-Plus}$
begin

14.1 Type definition

The standard Haskell libraries do not include a resumption monad transformer type; below is the Haskell definition for the one we will use here.

```
data ResT m a = Done a | More (m (ResT m a))
```

The above datatype definition can be translated directly into HOLCF using *tycondef*.

tycondef 'a.(f::functor) resT =
 Done (lazy 'a) | More (lazy ('a.'f resT).'f)

lemma coerce-resT-abs [simp]: coerce.(resT-abs.x) = resT-abs.(coerce.x)
 ⟨proof⟩

lemma coerce-Done [simp]: coerce.(Done.x) = Done.(coerce.x)
 ⟨proof⟩

lemma coerce-More [simp]: coerce.(More.m) = More.(coerce.m)
 ⟨proof⟩

lemma resT-induct [case-names adm bottom Done More]:
 fixes P :: 'a.'f::functor resT ⇒ bool
 assumes adm: adm P
 assumes bottom: P ⊥
 assumes Done: ∧x. P (Done.x)
 assumes More: ∧m f. (∧(r::'a.'f resT). P (f.r)) ⇒ P (More.(fmap.f.m))
 shows P r
 ⟨proof⟩

14.2 Class instance proofs

lemma fmapU-resT-simps [simp]:
 fmapU.f.(⊥::udom.'f::functor resT) = ⊥
 fmapU.f.(Done.x) = Done.(f.x)
 fmapU.f.(More.m) = More.(fmap.(fmapU.f).m)
 ⟨proof⟩

instance resT :: (functor) functor
 ⟨proof⟩

instantiation resT :: (functor) monad
begin

fixrec bindU-resT :: udom.'a resT → (udom → udom.'a resT) → udom.'a resT
where bindU-resT.(Done.x).f = f.x
 | bindU-resT.(More.m).f = More.(fmap.(λ r. bindU-resT.r.f).m)

lemma bindU-resT-strict [simp]: bindU.⊥.k = (⊥::udom.'a resT)
 ⟨proof⟩

definition
 returnU = Done

instance ⟨proof⟩

end

14.3 Transfer properties to polymorphic versions

lemma *fmap-resT-simps* [simp]:

$$\begin{aligned} \text{fmap}\cdot f\cdot(\perp :: 'a\cdot'f::\text{functor } \text{resT}) &= \perp \\ \text{fmap}\cdot f\cdot(\text{Done}\cdot x :: 'a\cdot'f::\text{functor } \text{resT}) &= \text{Done}\cdot(f\cdot x) \\ \text{fmap}\cdot f\cdot(\text{More}\cdot m :: 'a\cdot'f::\text{functor } \text{resT}) &= \text{More}\cdot(\text{fmap}\cdot(\text{fmap}\cdot f)\cdot m) \end{aligned}$$

<proof>

lemma *return-resT-def*: *return* = *Done*

<proof>

lemma *bind-resT-simps* [simp]:

$$\begin{aligned} \text{bind}\cdot(\perp :: 'a\cdot'f::\text{functor } \text{resT})\cdot f &= \perp \\ \text{bind}\cdot(\text{Done}\cdot x :: 'a\cdot'f::\text{functor } \text{resT})\cdot f &= f\cdot x \\ \text{bind}\cdot(\text{More}\cdot m :: 'a\cdot'f::\text{functor } \text{resT})\cdot f &= \text{More}\cdot(\text{fmap}\cdot(\lambda r.\ \text{bind}\cdot r\cdot f)\cdot m) \end{aligned}$$

<proof>

lemma *join-resT-simps* [simp]:

$$\begin{aligned} \text{join}\cdot\perp &= (\perp :: 'a\cdot'f::\text{functor } \text{resT}) \\ \text{join}\cdot(\text{Done}\cdot x) &= x \\ \text{join}\cdot(\text{More}\cdot m) &= \text{More}\cdot(\text{fmap}\cdot\text{join}\cdot m) \end{aligned}$$

<proof>

14.4 Nondeterministic interleaving

In this section we present a more general formalization of the nondeterministic interleaving operation presented in Chapter 7 of the author's PhD thesis [2]. If both arguments are *Done*, then *zipRT* combines the results with the function *f* and terminates. While either argument is *More*, *zipRT* nondeterministically chooses one such argument, runs it for one step, and then calls itself recursively.

fixrec *zipRT* ::

$$'a \rightarrow 'b \rightarrow 'c \rightarrow 'a\cdot('m::\text{functor-plus}) \text{resT} \rightarrow 'b\cdot'm \text{resT} \rightarrow 'c\cdot'm \text{resT}$$

where *zipRT-Done-Done*:

$$\text{zipRT}\cdot f\cdot(\text{Done}\cdot x)\cdot(\text{Done}\cdot y) = \text{Done}\cdot(f\cdot x\cdot y)$$

| *zipRT-Done-More*:

$$\begin{aligned} \text{zipRT}\cdot f\cdot(\text{Done}\cdot x)\cdot(\text{More}\cdot b) &= \\ \text{More}\cdot(\text{fmap}\cdot(\lambda r.\ \text{zipRT}\cdot f\cdot(\text{Done}\cdot x)\cdot r)\cdot b) & \end{aligned}$$

| *zipRT-More-Done*:

$$\begin{aligned} \text{zipRT}\cdot f\cdot(\text{More}\cdot a)\cdot(\text{Done}\cdot y) &= \\ \text{More}\cdot(\text{fmap}\cdot(\lambda r.\ \text{zipRT}\cdot f\cdot r\cdot(\text{Done}\cdot y))\cdot a) & \end{aligned}$$

| *zipRT-More-More*:

$$\begin{aligned} \text{zipRT}\cdot f\cdot(\text{More}\cdot a)\cdot(\text{More}\cdot b) &= \\ \text{More}\cdot(\text{fplus}\cdot(\text{fmap}\cdot(\lambda r.\ \text{zipRT}\cdot f\cdot(\text{More}\cdot a)\cdot r)\cdot b) & \\ \cdot(\text{fmap}\cdot(\lambda r.\ \text{zipRT}\cdot f\cdot r\cdot(\text{More}\cdot b))\cdot a)) & \end{aligned}$$

lemma *zipRT-strict1* [simp]: *zipRT*·*f*· \perp ·*r* = \perp

<proof>

lemma *zipRT-strict2* [*simp*]: $\text{zipRT}\cdot f\cdot r\cdot \perp = \perp$
(*proof*)

abbreviation *apR* (**infixl** \diamond 70)
where $a \diamond b \equiv \text{zipRT}\cdot \text{ID}\cdot a\cdot b$

Proofs that *zipRT* satisfies the applicative functor laws:

lemma *zipRT-homomorphism*: $\text{Done}\cdot f \diamond \text{Done}\cdot x = \text{Done}\cdot (f\cdot x)$
(*proof*)

lemma *zipRT-identity*: $\text{Done}\cdot \text{ID} \diamond r = r$
(*proof*)

lemma *zipRT-interchange*: $r \diamond \text{Done}\cdot x = \text{Done}\cdot (\lambda f. f\cdot x) \diamond r$
(*proof*)

The associativity rule is the hard one!

lemma *zipRT-associativity*: $\text{Done}\cdot \text{cfcomp} \diamond r1 \diamond r2 \diamond r3 = r1 \diamond (r2 \diamond r3)$
(*proof*)

end

15 State monad transformer

theory *State-Transformer*
imports *Monad-Zero-Plus*
begin

This version has non-lifted product, and a non-lifted function space.

tycondef $'a\cdot('f::\text{functor}, 's)$ *stateT* =
StateT (*runStateT* :: $'s \rightarrow ('a \times 's)\cdot'f$)

lemma *coerce-stateT-abs* [*simp*]: $\text{coerce}\cdot(\text{stateT}\cdot\text{abs}\cdot x) = \text{stateT}\cdot\text{abs}\cdot(\text{coerce}\cdot x)$
(*proof*)

lemma *coerce-StateT* [*simp*]: $\text{coerce}\cdot(\text{StateT}\cdot k) = \text{StateT}\cdot(\text{coerce}\cdot k)$
(*proof*)

lemma *stateT-cases* [*case-names StateT*]:
obtains *k* **where** $y = \text{StateT}\cdot k$
(*proof*)

lemma *stateT-induct* [*case-names StateT*]:
fixes $P :: 'a\cdot('f::\text{functor}, 's)$ *stateT* \Rightarrow *bool*
assumes $\bigwedge k. P (\text{StateT}\cdot k)$
shows $P y$
(*proof*)

lemma *stateT-eqI*:
 $(\bigwedge s. \text{runStateT} \cdot a \cdot s = \text{runStateT} \cdot b \cdot s) \implies a = b$
 ⟨*proof*⟩

lemma *runStateT-coerce* [*simp*]:
 $\text{runStateT} \cdot (\text{coerce} \cdot k) \cdot s = \text{coerce} \cdot (\text{runStateT} \cdot k \cdot s)$
 ⟨*proof*⟩

15.1 Functor class instance

lemma *fmapU-StateT* [*simp*]:
 $\text{fmapU} \cdot f \cdot (\text{StateT} \cdot k) =$
 $\text{StateT} \cdot (\bigwedge s. \text{fmap} \cdot (\bigwedge (x, s'). (f \cdot x, s')) \cdot (k \cdot s))$
 ⟨*proof*⟩

lemma *runStateT-fmapU* [*simp*]:
 $\text{runStateT} \cdot (\text{fmapU} \cdot f \cdot m) \cdot s =$
 $\text{fmap} \cdot (\bigwedge (x, s'). (f \cdot x, s')) \cdot (\text{runStateT} \cdot m \cdot s)$
 ⟨*proof*⟩

instantiation *stateT* :: (functor, domain) functor
begin

instance
 ⟨*proof*⟩

end

15.2 Monad class instance

instantiation *stateT* :: (monad, domain) monad
begin

definition *returnU-stateT-def*:
 $\text{returnU} = (\bigwedge x. \text{StateT} \cdot (\bigwedge s. \text{return} \cdot (x, s)))$

definition *bindU-stateT-def*:
 $\text{bindU} = (\bigwedge m k. \text{StateT} \cdot (\bigwedge s. \text{runStateT} \cdot m \cdot s \gg= (\bigwedge (x, s'). \text{runStateT} \cdot (k \cdot x) \cdot s')))$

lemma *bindU-stateT-StateT* [*simp*]:
 $\text{bindU} \cdot (\text{StateT} \cdot f) \cdot k =$
 $\text{StateT} \cdot (\bigwedge s. f \cdot s \gg= (\bigwedge (x, s'). \text{runStateT} \cdot (k \cdot x) \cdot s'))$
 ⟨*proof*⟩

lemma *runStateT-bindU* [*simp*]:
 $\text{runStateT} \cdot (\text{bindU} \cdot m \cdot k) \cdot s = \text{runStateT} \cdot m \cdot s \gg= (\bigwedge (x, s'). \text{runStateT} \cdot (k \cdot x) \cdot s')$
 ⟨*proof*⟩

instance ⟨*proof*⟩

end

15.3 Monad zero instance

instantiation $stateT :: (monad-zero, domain) monad-zero$
begin

definition $zeroU-stateT-def$:
 $zeroU = StateT \cdot (\lambda s. mzero)$

lemma $runStateT-zeroU [simp]$:
 $runStateT \cdot zeroU \cdot s = mzero$
 $\langle proof \rangle$

instance $\langle proof \rangle$

end

15.4 Monad plus instance

instantiation $stateT :: (monad-plus, domain) monad-plus$
begin

definition $plusU-stateT-def$:
 $plusU = (\lambda a b. StateT \cdot (\lambda s. mplus \cdot (runStateT \cdot a \cdot s) \cdot (runStateT \cdot b \cdot s)))$

lemma $runStateT-plusU [simp]$:
 $runStateT \cdot (plusU \cdot a \cdot b) \cdot s =$
 $mplus \cdot (runStateT \cdot a \cdot s) \cdot (runStateT \cdot b \cdot s)$
 $\langle proof \rangle$

instance $\langle proof \rangle$

end

15.5 Monad zero plus instance

instance $stateT :: (monad-zero-plus, domain) monad-zero-plus$
 $\langle proof \rangle$

15.6 Transfer properties to polymorphic versions

lemma $coerce-csplit [coerce-simp]$:
shows $coerce \cdot (csplit \cdot f \cdot p) = csplit \cdot (\lambda x y. coerce \cdot (f \cdot x \cdot y)) \cdot p$
 $\langle proof \rangle$

lemma $csplit-coerce [coerce-simp]$:
fixes $p :: 'a \times 'b$
shows $csplit \cdot f \cdot (COERCE ('a \times 'b, 'c \times 'd) \cdot p) =$

$csplit \cdot (\Lambda x y. f \cdot (COERCE('a, 'c) \cdot x) \cdot (COERCE('b, 'd) \cdot y)) \cdot p$
 $\langle proof \rangle$

lemma *fmap-stateT-simps* [*simp*]:
 $fmap \cdot f \cdot (StateT \cdot m :: 'a \cdot ('f :: functor, 's) \text{ stateT}) =$
 $StateT \cdot (\Lambda s. fmap \cdot (\Lambda (x, s'). (f \cdot x, s')) \cdot (m \cdot s))$
 $\langle proof \rangle$

lemma *runStateT-fmap* [*simp*]:
 $runStateT \cdot (fmap \cdot f \cdot m) \cdot s = fmap \cdot (\Lambda (x, s'). (f \cdot x, s')) \cdot (runStateT \cdot m \cdot s)$
 $\langle proof \rangle$

lemma *return-stateT-def*:
 $(return :: - \rightarrow 'a \cdot ('m :: monad, 's) \text{ stateT}) =$
 $(\Lambda x. StateT \cdot (\Lambda s. return \cdot (x, s)))$
 $\langle proof \rangle$

lemma *bind-stateT-def*:
 $bind = (\Lambda m k. StateT \cdot (\Lambda s. runStateT \cdot m \cdot s \gg= (\Lambda (x, s'). runStateT \cdot (k \cdot x) \cdot s')))$
 $\langle proof \rangle$

TODO: add *coerce-idem* to *coerce-simps*, along with monotonicity rules for DEFL.

lemma *bind-stateT-simps* [*simp*]:
 $bind \cdot (StateT \cdot m :: 'a \cdot ('m :: monad, 's) \text{ stateT}) \cdot k =$
 $StateT \cdot (\Lambda s. m \cdot s \gg= (\Lambda (x, s'). runStateT \cdot (k \cdot x) \cdot s'))$
 $\langle proof \rangle$

lemma *runStateT-bind* [*simp*]:
 $runStateT \cdot (m \gg= k) \cdot s = runStateT \cdot m \cdot s \gg= (\Lambda (x, s'). runStateT \cdot (k \cdot x) \cdot s')$
 $\langle proof \rangle$

end

16 Error monad transformer

theory *Error-Transformer*
imports *Error-Monad*
begin

16.1 Type definition

The error monad transformer is defined in Haskell by composing the given monad with a standard error monad:

```
data Error e a = Err e | Ok a
newtype ErrorT e m a = ErrorT { runErrorT :: m (Error e a) }
```

We can formalize this definition directly using *tycondef*.

tycondef 'a.(f::functor,'e::domain) errorT =
 ErrorT (runErrorT :: ('a.'e error)·f)

lemma coerce-errorT-abs [simp]: coerce·(errorT-abs·x) = errorT-abs·(coerce·x)
 ⟨proof⟩

lemma coerce-ErrorT [simp]: coerce·(ErrorT·k) = ErrorT·(coerce·k)
 ⟨proof⟩

lemma errorT-cases [case-names ErrorT]:
obtains k **where** y = ErrorT·k
 ⟨proof⟩

lemma ErrorT-runErrorT [simp]: ErrorT·(runErrorT·m) = m
 ⟨proof⟩

lemma errorT-induct [case-names ErrorT]:
fixes P :: 'a.(f::functor,'e) errorT ⇒ bool
assumes $\bigwedge k. P (ErrorT·k)$
shows P y
 ⟨proof⟩

lemma errorT-eq-iff:
 $a = b \longleftrightarrow runErrorT·a = runErrorT·b$
 ⟨proof⟩

lemma errorT-eqI:
 $runErrorT·a = runErrorT·b \implies a = b$
 ⟨proof⟩

lemma runErrorT-coerce [simp]:
 $runErrorT·(coerce·k) = coerce·(runErrorT·k)$
 ⟨proof⟩

16.2 Functor class instance

lemma fmap-error-def: fmap = error-map·ID
 ⟨proof⟩

lemma fmapU-ErrorT [simp]:
 $fmapU·f·(ErrorT·m) = ErrorT·(fmap·(fmap·f)·m)$
 ⟨proof⟩

lemma runErrorT-fmapU [simp]:
 $runErrorT·(fmapU·f·m) = fmap·(fmap·f)·(runErrorT·m)$
 ⟨proof⟩

instance errorT :: (functor, domain) functor

<proof>

16.3 Transfer properties to polymorphic versions

lemma *fmap-ErrorT* [*simp*]:

fixes $f :: 'a \rightarrow 'b$ **and** $m :: 'a \cdot 'e \text{ error} \cdot ('m :: \text{functor})$

shows $fmap \cdot f \cdot (\text{ErrorT} \cdot m) = \text{ErrorT} \cdot (fmap \cdot (fmap \cdot f) \cdot m)$

<proof>

lemma *runErrorT-fmap* [*simp*]:

fixes $f :: 'a \rightarrow 'b$ **and** $m :: 'a \cdot ('m :: \text{functor}, 'e) \text{ errorT}$

shows $runErrorT \cdot (fmap \cdot f \cdot m) = fmap \cdot (fmap \cdot f) \cdot (runErrorT \cdot m)$

<proof>

lemma *errorT-fmap-strict* [*simp*]:

shows $fmap \cdot f \cdot (\perp :: 'a \cdot ('m :: \text{monad}, 'e) \text{ errorT}) = \perp$

<proof>

16.4 Monad operations

The error monad transformer does not yield a monad in the usual sense: We cannot prove a *monad* class instance, because type $'a \cdot ('m, 'e) \text{ errorT}$ contains values that break the monad laws. However, it turns out that such values are inaccessible: The monad laws are satisfied by all values constructible from the abstract operations.

To explore the properties of the error monad transformer operations, we define them all as non-overloaded functions.

definition *unitET* :: $'a \rightarrow 'a \cdot ('m :: \text{monad}, 'e) \text{ errorT}$

where $unitET = (\lambda x. \text{ErrorT} \cdot (\text{return} \cdot (\text{Ok} \cdot x)))$

definition *bindET* :: $'a \cdot ('m :: \text{monad}, 'e) \text{ errorT} \rightarrow$

$('a \rightarrow 'b \cdot ('m, 'e) \text{ errorT}) \rightarrow 'b \cdot ('m, 'e) \text{ errorT}$

where $bindET = (\lambda m k. \text{ErrorT} \cdot (\text{bind} \cdot (runErrorT \cdot m) \cdot$

$(\lambda n. \text{case } n \text{ of } \text{Err} \cdot e \Rightarrow \text{return} \cdot (\text{Err} \cdot e) \mid \text{Ok} \cdot x \Rightarrow runErrorT \cdot (k \cdot x))))$

definition *liftET* :: $'a \cdot 'm :: \text{monad} \rightarrow 'a \cdot ('m, 'e) \text{ errorT}$

where $liftET = (\lambda m. \text{ErrorT} \cdot (fmap \cdot \text{Ok} \cdot m))$

definition *throwET* :: $'e \rightarrow 'a \cdot ('m :: \text{monad}, 'e) \text{ errorT}$

where $throwET = (\lambda e. \text{ErrorT} \cdot (\text{return} \cdot (\text{Err} \cdot e)))$

definition *catchET* :: $'a \cdot ('m :: \text{monad}, 'e) \text{ errorT} \rightarrow$

$('e \rightarrow 'a \cdot ('m, 'e) \text{ errorT}) \rightarrow 'a \cdot ('m, 'e) \text{ errorT}$

where $catchET = (\lambda m h. \text{ErrorT} \cdot (\text{bind} \cdot (runErrorT \cdot m) \cdot (\lambda n. \text{case } n \text{ of}$

$\text{Err} \cdot e \Rightarrow runErrorT \cdot (h \cdot e) \mid \text{Ok} \cdot x \Rightarrow \text{return} \cdot (\text{Ok} \cdot x))))$

definition *fmapET* :: $('a \rightarrow 'b) \rightarrow$

'a.('m::monad,'e) errorT → 'b.('m,'e) errorT
where fmapET = (λ f m. bindET·m·(λ x. unitET·(f·x)))

lemma runErrorT-unitET [simp]:

runErrorT·(unitET·x) = return·(Ok·x)
⟨proof⟩

lemma runErrorT-bindET [simp]:

runErrorT·(bindET·m·k) = bind·(runErrorT·m)·
(λ n. case n of Err·e ⇒ return·(Err·e) | Ok·x ⇒ runErrorT·(k·x))
⟨proof⟩

lemma runErrorT-liftET [simp]:

runErrorT·(liftET·m) = fmap·Ok·m
⟨proof⟩

lemma runErrorT-throwET [simp]:

runErrorT·(throwET·e) = return·(Err·e)
⟨proof⟩

lemma runErrorT-catchET [simp]:

runErrorT·(catchET·m·h) =
bind·(runErrorT·m)·(λ n. case n of
Err·e ⇒ runErrorT·(h·e) | Ok·x ⇒ return·(Ok·x))
⟨proof⟩

lemma runErrorT-fmapET [simp]:

runErrorT·(fmapET·f·m) =
bind·(runErrorT·m)·(λ n. case n of
Err·e ⇒ return·(Err·e) | Ok·x ⇒ return·(Ok·(f·x)))
⟨proof⟩

16.5 Laws

lemma bindET-unitET [simp]:

bindET·(unitET·x)·k = k·x
⟨proof⟩

lemma catchET-unitET [simp]:

catchET·(unitET·x)·h = unitET·x
⟨proof⟩

lemma catchET-throwET [simp]:

catchET·(throwET·e)·h = h·e
⟨proof⟩

lemma liftET-return:

liftET·(return·x) = unitET·x
⟨proof⟩

lemma *liftET-bind*:

$liftET \cdot (bind \cdot m \cdot k) = bindET \cdot (liftET \cdot m) \cdot (liftET \text{ oo } k)$
 $\langle proof \rangle$

lemma *bindET-throwET*:

$bindET \cdot (throwET \cdot e) \cdot k = throwET \cdot e$
 $\langle proof \rangle$

lemma *bindET-bindET*:

$bindET \cdot (bindET \cdot m \cdot h) \cdot k = bindET \cdot m \cdot (\Lambda x. bindET \cdot (h \cdot x) \cdot k)$
 $\langle proof \rangle$

lemma *fmapET-fmapET*:

$fmapET \cdot f \cdot (fmapET \cdot g \cdot m) = fmapET \cdot (\Lambda x. f \cdot (g \cdot x)) \cdot m$
 $\langle proof \rangle$

Right unit monad law is not satisfied in general.

lemma *bindET-unitET-right-counterexample*:

fixes $m :: 'a \cdot ('m :: monad, 'e) \text{ error } T$
assumes $m = ErrorT \cdot (return \cdot \perp)$
assumes $return \cdot \perp \neq (\perp :: ('a \cdot 'e \text{ error}) \cdot 'm)$
shows $bindET \cdot m \cdot unitET \neq m$
 $\langle proof \rangle$

Right unit is satisfied for inner monads with strict return.

lemma *bindET-unitET-right-restricted*:

fixes $m :: 'a \cdot ('m :: monad, 'e) \text{ error } T$
assumes $return \cdot \perp = (\perp :: ('a \cdot 'e \text{ error}) \cdot 'm)$
shows $bindET \cdot m \cdot unitET = m$
 $\langle proof \rangle$

16.6 Error monad transformer invariant

This inductively-defined invariant is supposed to represent the set of all values constructible using the standard *errorT* operations.

inductive *invar* :: $'a \cdot ('m :: monad, 'e) \text{ error } T \Rightarrow bool$

where *invar-bottom*: $invar \perp$

| *invar-lub*: $\bigwedge Y. \llbracket chain \ Y; \bigwedge i. invar \ (Y \ i) \rrbracket \Longrightarrow invar \ (\bigsqcup i. \ Y \ i)$

| *invar-unitET*: $\bigwedge x. invar \ (unitET \cdot x)$

| *invar-bindET*: $\bigwedge m \ k. \llbracket invar \ m; \bigwedge x. invar \ (k \cdot x) \rrbracket \Longrightarrow invar \ (bindET \cdot m \cdot k)$

| *invar-throwET*: $\bigwedge e. invar \ (throwET \cdot e)$

| *invar-catchET*: $\bigwedge m \ h. \llbracket invar \ m; \bigwedge e. invar \ (h \cdot e) \rrbracket \Longrightarrow invar \ (catchET \cdot m \cdot h)$

| *invar-liftET*: $\bigwedge m. invar \ (liftET \cdot m)$

Right unit is satisfied for arguments built from standard functions.

lemma *bindET-unitET-right-invar*:

assumes $invar\ m$
shows $bindET \cdot m \cdot unitET = m$
 $\langle proof \rangle$

Monad-fmap is satisfied for arguments built from standard functions.

lemma $errorT\text{-monad-fmap-invar}$:
fixes $f :: 'a \rightarrow 'b$ **and** $m :: 'a \cdot ('m :: monad, 'e)\ errorT$
assumes $invar\ m$
shows $fmap \cdot f \cdot m = bindET \cdot m \cdot (\Lambda x. unitET \cdot (f \cdot x))$
 $\langle proof \rangle$

16.7 Invariant expressed as a deflation

We can also define an invariant in a more semantic way, as the set of fixed-points of a deflation.

definition $invar' :: 'a \cdot ('m :: monad, 'e)\ errorT \Rightarrow bool$
where $invar' m \longleftrightarrow fmapET \cdot ID \cdot m = m$

All standard operations preserve the invariant.

lemma $invar'\text{-unitET}$: $invar' (unitET \cdot x)$
 $\langle proof \rangle$

lemma $invar'\text{-fmapET}$: $invar' m \Longrightarrow invar' (fmapET \cdot f \cdot m)$
 $\langle proof \rangle$

lemma $invar'\text{-bindET}$: $\llbracket invar' m; \bigwedge x. invar' (k \cdot x) \rrbracket \Longrightarrow invar' (bindET \cdot m \cdot k)$
 $\langle proof \rangle$

lemma $invar'\text{-throwET}$: $invar' (throwET \cdot e)$
 $\langle proof \rangle$

lemma $invar'\text{-catchET}$: $\llbracket invar' m; \bigwedge e. invar' (h \cdot e) \rrbracket \Longrightarrow invar' (catchET \cdot m \cdot h)$
 $\langle proof \rangle$

lemma $invar'\text{-liftET}$: $invar' (liftET \cdot m)$
 $\langle proof \rangle$

lemma $invar'\text{-bottom}$: $invar' \perp$
 $\langle proof \rangle$

lemma $adm\text{-invar'}$: $adm\ invar'$
 $\langle proof \rangle$

All monad laws are preserved by values satisfying the invariant.

lemma $bindET\text{-fmapET-unitET}$:
shows $bindET \cdot (fmapET \cdot f \cdot m) \cdot unitET = fmapET \cdot f \cdot m$
 $\langle proof \rangle$

lemma *invar'-right-unit*: $invar' m \implies bindET \cdot m \cdot unitET = m$
 ⟨proof⟩

lemma *invar'-monad-fmap*:
 $invar' m \implies fmapET \cdot f \cdot m = bindET \cdot m \cdot (\lambda x. unitET \cdot (f \cdot x))$
 ⟨proof⟩

lemma *invar'-bind-assoc*:
 $\llbracket invar' m; \bigwedge x. invar' (f \cdot x); \bigwedge y. invar' (g \cdot y) \rrbracket$
 $\implies bindET \cdot (bindET \cdot m \cdot f) \cdot g = bindET \cdot m \cdot (\lambda x. bindET \cdot (f \cdot x) \cdot g)$
 ⟨proof⟩

end

17 Writer monad transformer

theory *Writer-Transformer*
imports *Writer-Monad*
begin

17.1 Type definition

Below is the standard Haskell definition of a writer monad transformer:

```
newtype WriterT w m a = WriterT { runWriterT :: m (a, w) }
```

In this development, since a lazy pair type is not pre-defined in HOLCF, we will use an equivalent formulation in terms of our previous `Writer` type:

```
data Writer w a = Writer w a
newtype WriterT w m a = WriterT { runWriterT :: m (Writer w a) }
```

We can translate this definition directly into HOLCF using *tycondef*.

```
tycondef 'a.('m::functor,'w) writerT =
  WriterT (runWriterT :: ('a.'w writer).'m)
```

lemma *coerce-writerT-abs* [simp]:
 $coerce \cdot (writerT-abs \cdot x) = writerT-abs \cdot (coerce \cdot x)$
 ⟨proof⟩

lemma *coerce-WriterT* [simp]: $coerce \cdot (WriterT \cdot k) = WriterT \cdot (coerce \cdot k)$
 ⟨proof⟩

lemma *writerT-cases* [case-names *WriterT*]:
obtains *k* **where** $y = WriterT \cdot k$

$\langle proof \rangle$

lemma *WriterT-runWriterT* [simp]: $WriterT \cdot (run\ WriterT \cdot m) = m$
 $\langle proof \rangle$

lemma *writerT-induct* [case-names *WriterT*]:
fixes $P :: 'a \cdot ('f :: functor, 'e) \ writerT \Rightarrow bool$
assumes $\bigwedge k. P (WriterT \cdot k)$
shows $P y$
 $\langle proof \rangle$

lemma *writerT-eq-iff*:
 $a = b \iff run\ WriterT \cdot a = run\ WriterT \cdot b$
 $\langle proof \rangle$

lemma *writerT-below-iff*:
 $a \sqsubseteq b \iff run\ WriterT \cdot a \sqsubseteq run\ WriterT \cdot b$
 $\langle proof \rangle$

lemma *writerT-eqI*:
 $run\ WriterT \cdot a = run\ WriterT \cdot b \implies a = b$
 $\langle proof \rangle$

lemma *writerT-belowI*:
 $run\ WriterT \cdot a \sqsubseteq run\ WriterT \cdot b \implies a \sqsubseteq b$
 $\langle proof \rangle$

lemma *runWriterT-coerce* [simp]:
 $run\ WriterT \cdot (coerce \cdot k) = coerce \cdot (run\ WriterT \cdot k)$
 $\langle proof \rangle$

17.2 Functor class instance

lemma *fmap-writer-def*: $fmap = writer\ map \cdot ID$
 $\langle proof \rangle$

lemma *fmapU-WriterT* [simp]:
 $fmapU \cdot f \cdot (WriterT \cdot m) = WriterT \cdot (fmap \cdot (fmap \cdot f) \cdot m)$
 $\langle proof \rangle$

lemma *runWriterT-fmapU* [simp]:
 $run\ WriterT \cdot (fmapU \cdot f \cdot m) = fmap \cdot (fmap \cdot f) \cdot (run\ WriterT \cdot m)$
 $\langle proof \rangle$

instance *writerT* :: (functor, domain) functor
 $\langle proof \rangle$

17.3 Monad operations

The writer monad transformer does not yield a monad in the usual sense: We cannot prove a *monad* class instance, because type $'a \cdot ('m, 'w)$ *writerT* contains values that break the monad laws. However, it turns out that such values are inaccessible: The monad laws are satisfied by all values constructible from the abstract operations.

To explore the properties of the writer monad transformer operations, we define them all as non-overloaded functions.

definition *unitWT* :: $'a \rightarrow 'a \cdot ('m :: \text{monad}, 'w :: \text{monoid})$ *writerT*
where *unitWT* = $(\Lambda x. \text{WriterT} \cdot (\text{return} \cdot (\text{Writer} \cdot \text{empty} \cdot x)))$

definition *bindWT* :: $'a \cdot ('m :: \text{monad}, 'w :: \text{monoid})$ *writerT* $\rightarrow ('a \rightarrow 'b \cdot ('m, 'w)$ *writerT*) $\rightarrow 'b \cdot ('m, 'w)$ *writerT*
where *bindWT* = $(\Lambda m k. \text{WriterT} \cdot (\text{bind} \cdot (\text{runWriterT} \cdot m) \cdot (\Lambda (\text{Writer} \cdot w \cdot x). \text{bind} \cdot (\text{runWriterT} \cdot (k \cdot x)) \cdot (\Lambda (\text{Writer} \cdot w' \cdot y). \text{return} \cdot (\text{Writer} \cdot (\text{mappend} \cdot w \cdot w') \cdot y))))))$

definition *liftWT* :: $'a \cdot 'm \rightarrow 'a \cdot ('m :: \text{monad}, 'w :: \text{monoid})$ *writerT*
where *liftWT* = $(\Lambda m. \text{WriterT} \cdot (\text{fmap} \cdot (\text{Writer} \cdot \text{empty}) \cdot m))$

definition *tellWT* :: $'a \rightarrow 'w \rightarrow 'a \cdot ('m :: \text{monad}, 'w :: \text{monoid})$ *writerT*
where *tellWT* = $(\Lambda x w. \text{WriterT} \cdot (\text{return} \cdot (\text{Writer} \cdot w \cdot x)))$

definition *fmapWT* :: $('a \rightarrow 'b) \rightarrow 'a \cdot ('m :: \text{monad}, 'w :: \text{monoid})$ *writerT* $\rightarrow 'b \cdot ('m, 'w)$ *writerT*
where *fmapWT* = $(\Lambda f m. \text{bindWT} \cdot m \cdot (\Lambda x. \text{unitWT} \cdot (f \cdot x)))$

lemma *runWriterT-fmap* [*simp*]:
 $\text{runWriterT} \cdot (\text{fmap} \cdot f \cdot m) = \text{fmap} \cdot (\text{fmap} \cdot f) \cdot (\text{runWriterT} \cdot m)$
 ⟨*proof*⟩

lemma *runWriterT-unitWT* [*simp*]:
 $\text{runWriterT} \cdot (\text{unitWT} \cdot x) = \text{return} \cdot (\text{Writer} \cdot \text{empty} \cdot x)$
 ⟨*proof*⟩

lemma *runWriterT-bindWT* [*simp*]:
 $\text{runWriterT} \cdot (\text{bindWT} \cdot m \cdot k) = \text{bind} \cdot (\text{runWriterT} \cdot m) \cdot (\Lambda (\text{Writer} \cdot w \cdot x). \text{bind} \cdot (\text{runWriterT} \cdot (k \cdot x)) \cdot (\Lambda (\text{Writer} \cdot w' \cdot y). \text{return} \cdot (\text{Writer} \cdot (\text{mappend} \cdot w \cdot w') \cdot y)))$
 ⟨*proof*⟩

lemma *runWriterT-liftWT* [*simp*]:
 $\text{runWriterT} \cdot (\text{liftWT} \cdot m) = \text{fmap} \cdot (\text{Writer} \cdot \text{empty}) \cdot m$
 ⟨*proof*⟩

lemma *runWriterT-tellWT* [*simp*]:

$run\ WriterT.(tell\ WT.x.w) = return.(Writer.w.x)$
 ⟨proof⟩

lemma $run\ WriterT-fmap\ WT$ [simp]:
 $run\ WriterT.(fmap\ WT.f.m) =$
 $run\ WriterT.m \gg= (\lambda (Writer.w.x). return.(Writer.w.(f.x)))$
 ⟨proof⟩

17.4 Laws

The $lift\ WT$ function maps $return$ and $bind$ on the inner monad to $unit\ WT$ and $bind\ WT$, as expected.

lemma $lift\ WT-return$:
 $lift\ WT.(return.x) = unit\ WT.x$
 ⟨proof⟩

lemma $lift\ WT-bind$:
 $lift\ WT.(bind.m.k) = bind\ WT.(lift\ WT.m).(lift\ WT\ oo\ k)$
 ⟨proof⟩

The composition rule holds unconditionally for $fmap$. The $fmap$ function also interacts as expected with $unit$ and $bind$.

lemma $fmap\ WT-fmap\ WT$:
 $fmap\ WT.f.(fmap\ WT.g.m) = fmap\ WT.(\lambda x. f.(g.x)).m$
 ⟨proof⟩

lemma $fmap\ WT-unit\ WT$:
 $fmap\ WT.f.(unit\ WT.x) = unit\ WT.(f.x)$
 ⟨proof⟩

lemma $fmap\ WT-bind\ WT$:
 $fmap\ WT.f.(bind\ WT.m.k) = bind\ WT.m.(\lambda x. fmap\ WT.f.(k.x))$
 ⟨proof⟩

lemma $bind\ WT-fmap\ WT$:
 $bind\ WT.(fmap\ WT.f.m).k = bind\ WT.m.(\lambda x. k.(f.x))$
 ⟨proof⟩

The left unit monad law is not satisfied in general.

lemma $bind\ WT-unit\ WT-counterexample$:
fixes $k :: 'a \rightarrow 'b.(m::monad, w::monoid)\ writerT$
assumes $1: k.x = WriterT.(return.\perp)$
assumes $2: return.\perp \neq (\perp :: ('b.'w\ writer).m::monad)$
shows $bind\ WT.(unit\ WT.x).k \neq k.x$
 ⟨proof⟩

However, left unit is satisfied for inner monads with a strict $return$ function.

lemma *bindWT-unitWT-restricted*:
fixes $k :: 'a \rightarrow 'b \cdot ('m::\text{monad}, 'w::\text{monoid}) \text{writerT}$
assumes $\text{return} \cdot \perp = (\perp :: ('b \cdot 'w \text{writer}) \cdot 'm)$
shows $\text{bindWT} \cdot (\text{unitWT} \cdot x) \cdot k = k \cdot x$
 $\langle \text{proof} \rangle$

The associativity of *bindWT* holds unconditionally.

lemma *bindWT-bindWT*:
 $\text{bindWT} \cdot (\text{bindWT} \cdot m \cdot h) \cdot k = \text{bindWT} \cdot m \cdot (\Lambda x. \text{bindWT} \cdot (h \cdot x) \cdot k)$
 $\langle \text{proof} \rangle$

The right unit monad law is not satisfied in general.

lemma *bindWT-unitWT-right-counterexample*:
fixes $m :: 'a \cdot ('m::\text{monad}, 'w::\text{monoid}) \text{writerT}$
assumes $m = \text{WriterT} \cdot (\text{return} \cdot \perp)$
assumes $\text{return} \cdot \perp \neq (\perp :: ('a \cdot 'w \text{writer}) \cdot 'm)$
shows $\text{bindWT} \cdot m \cdot \text{unitWT} \neq m$
 $\langle \text{proof} \rangle$

Right unit is satisfied for inner monads with a strict *return* function.

lemma *bindWT-unitWT-right-restricted*:
fixes $m :: 'a \cdot ('m::\text{monad}, 'w::\text{monoid}) \text{writerT}$
assumes $\text{return} \cdot \perp = (\perp :: ('a \cdot 'w \text{writer}) \cdot 'm)$
shows $\text{bindWT} \cdot m \cdot \text{unitWT} = m$
 $\langle \text{proof} \rangle$

17.5 Writer monad transformer invariant

We inductively define a predicate that includes all values that can be constructed from the standard *writerT* operations.

inductive $\text{invar} :: 'a \cdot ('m::\text{monad}, 'w::\text{monoid}) \text{writerT} \Rightarrow \text{bool}$
where *invar-bottom*: $\text{invar} \perp$
| *invar-lub*: $\bigwedge Y. \llbracket \text{chain } Y; \bigwedge i. \text{invar } (Y \ i) \rrbracket \Longrightarrow \text{invar } (\bigsqcup i. Y \ i)$
| *invar-unitWT*: $\bigwedge x. \text{invar } (\text{unitWT} \cdot x)$
| *invar-bindWT*: $\bigwedge m \ k. \llbracket \text{invar } m; \bigwedge x. \text{invar } (k \cdot x) \rrbracket \Longrightarrow \text{invar } (\text{bindWT} \cdot m \cdot k)$
| *invar-tellWT*: $\bigwedge x \ w. \text{invar } (\text{tellWT} \cdot x \cdot w)$
| *invar-liftWT*: $\bigwedge m. \text{invar } (\text{liftWT} \cdot m)$

Right unit is satisfied for arguments built from standard functions.

lemma *bindWT-unitWT-right-invar*:
fixes $m :: 'a \cdot ('m::\text{monad}, 'w::\text{monoid}) \text{writerT}$
assumes $\text{invar } m$
shows $\text{bindWT} \cdot m \cdot \text{unitWT} = m$
 $\langle \text{proof} \rangle$

Left unit is also satisfied for arguments built from standard functions.

lemma *writerT-left-unit-invar-lemma*:

assumes *invar m*

shows $\text{run WriterT} \cdot m \gg (\lambda (Writer \cdot w \cdot x). \text{return} \cdot (Writer \cdot w \cdot x)) = \text{run WriterT} \cdot m$
 ⟨*proof*⟩

lemma *bindWT-unitWT-invar*:

assumes *invar (k·x)*

shows $\text{bindWT} \cdot (\text{unitWT} \cdot x) \cdot k = k \cdot x$
 ⟨*proof*⟩

17.6 Invariant expressed as a deflation

definition *invar'* :: '*a*·('m::monad, 'w::monoid) *writerT* ⇒ *bool*

where $\text{invar}' \ m \longleftrightarrow \text{fmapWT} \cdot \text{ID} \cdot m = m$

All standard operations preserve the invariant.

lemma *invar'-bottom*: *invar' ⊥*

⟨*proof*⟩

lemma *adm-invar'*: *adm invar'*

⟨*proof*⟩

lemma *invar'-unitWT*: *invar' (unitWT·x)*

⟨*proof*⟩

lemma *invar'-bindWT*: $\llbracket \text{invar}' \ m; \bigwedge x. \text{invar}' \ (k \cdot x) \rrbracket \implies \text{invar}' \ (\text{bindWT} \cdot m \cdot k)$

⟨*proof*⟩

lemma *invar'-tellWT*: *invar' (tellWT·x·w)*

⟨*proof*⟩

lemma *invar'-liftWT*: *invar' (liftWT·m)*

⟨*proof*⟩

Left unit is satisfied for arguments built from fmap.

lemma *bindWT-unitWT-fmapWT*:

$\text{bindWT} \cdot (\text{unitWT} \cdot x) \cdot (\lambda x. \text{fmapWT} \cdot f \cdot (k \cdot x))$
 $= \text{fmapWT} \cdot f \cdot (k \cdot x)$

⟨*proof*⟩

Right unit is satisfied for arguments built from fmap.

lemma *bindWT-fmapWT-unitWT*:

shows $\text{bindWT} \cdot (\text{fmapWT} \cdot f \cdot m) \cdot \text{unitWT} = \text{fmapWT} \cdot f \cdot m$

⟨*proof*⟩

All monad laws are preserved by values satisfying the invariant.

lemma *invar'-right-unit*: $invar' m \implies bindWT \cdot m \cdot unitWT = m$
<proof>

lemma *invar'-monad-fmap*:
 $invar' m \implies fmapWT \cdot f \cdot m = bindWT \cdot m \cdot (\lambda x. unitWT \cdot (f \cdot x))$
<proof>

lemma *invar'-bind-assoc*:
 $\llbracket invar' m; \bigwedge x. invar' (f \cdot x); \bigwedge y. invar' (g \cdot y) \rrbracket$
 $\implies bindWT \cdot (bindWT \cdot m \cdot f) \cdot g = bindWT \cdot m \cdot (\lambda x. bindWT \cdot (f \cdot x) \cdot g)$
<proof>

end

References

- [1] B. Huffman. Formal verification of monad transformers. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*. Publication pending.
- [2] B. Huffman. *HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs*. Ph.D. thesis, Portland State University, 2012.
- [3] B. Huffman, J. Matthews, and P. White. Axiomatic constructor classes in Isabelle/HOLCF. In J. Hurd and T. 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.