Tycon: Type Constructor Classes and Monad Transformers

Brian Huffman

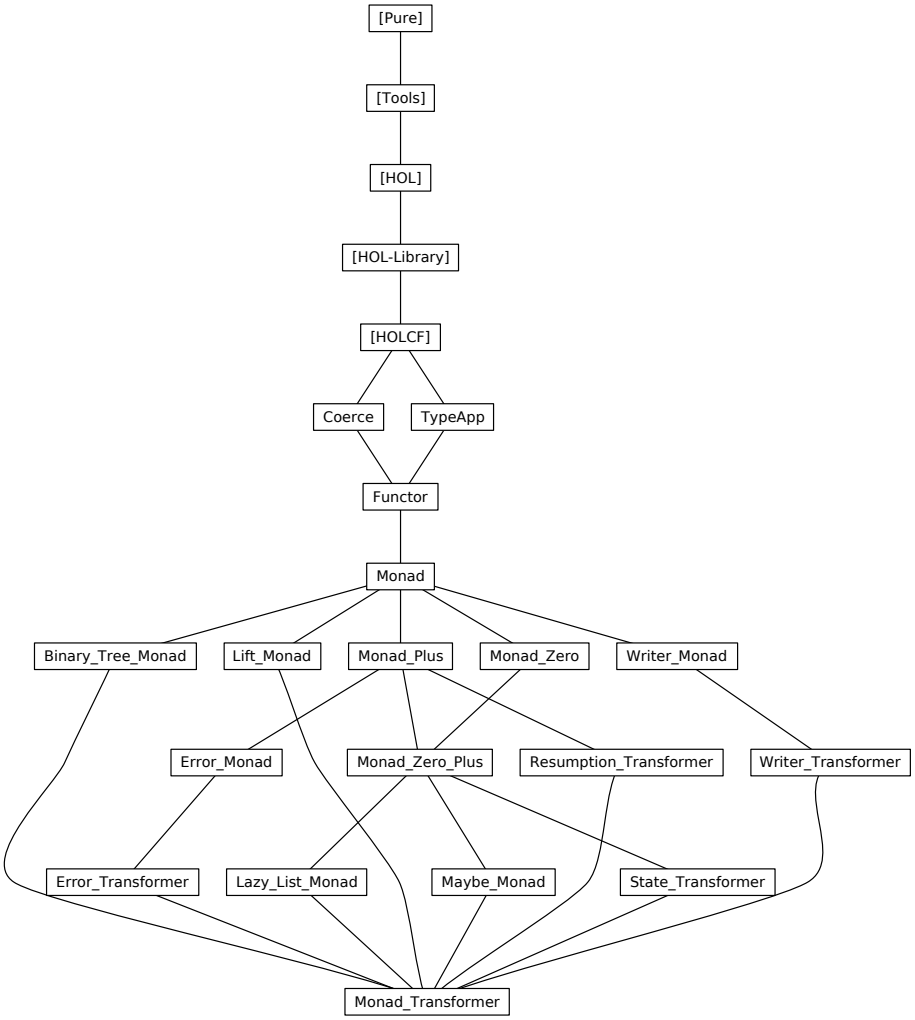
December 14, 2021

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


[Pure]
[Tools]
[HOL]
[HOL-Library]
[HOLCF]
Coerce
TypeApp
Functor
Monad
Binary_Tree_Monad
Lift_Monad
Monad_Plus
Monad_Zero
Writer_Monad
Error_Monad
Monad_Zero_Plus
Resumption_Transformer
Writer_Transformer
Error_Transformer
Lazy_List_Monad
Maybe_Monad
State_Transformer
Monad_Transformer

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* → *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* → *udom defl*.

class *tycon* =
  fixes *tc* :: (*'a*::*type*) *itself* ⇒ *udom defl* → *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* ⇒ *logic* ((*1TC*/(*1'*(-*'*))))

translations *TC*(*'a*) ⇌ *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* ⇒ *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*: $DEFL(?'a \cdot ?'t) = TC(?'t) \cdot DEFL(?'a)$, which we can use to derive other useful lemmas.

lemma *TC-DEFL*: $TC('t::tycon) \cdot DEFL('a) = DEFL('a \cdot 't)$
⟨*proof*⟩

lemma *DEFL-app-mono* [*simp*, *intro*]:
$DEFL('a) \sqsubseteq DEFL('b) \Longrightarrow DEFL('a \cdot 't::tycon) \sqsubseteq DEFL('b \cdot 't)$
⟨*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: $emb::'a \to udom$ and $prj::udom \to 'a$. By composing the *prj* and *emb* functions together, we can coerce values between any two types in class *domain*.

definition $coerce :: 'a \to 'b$
where $coerce \equiv prj \mathbin{oo} emb$

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* ⇒ *logic* ((*1EMB*/(*1* '(-'))))
-prj :: *type* ⇒ *logic* ((*1PRJ*/(*1* '(-'))))
-coerce :: *type* ⇒ *type* ⇒ *logic* ((*1COERCE*/(*1* '(-,/ -'))))

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

⟨*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*]: $coerce{\cdot}\bot = \bot$
⟨*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*:
$DEFL('a) \sqsubseteq DEFL('b)$
$\Longrightarrow EMB('b){\cdot}(COERCE('a,'b){\cdot}x) = EMB('a){\cdot}x$
⟨*proof*⟩

lemma *coerce-prj*:
$DEFL('a) \sqsubseteq DEFL('b)$
$\Longrightarrow COERCE('b,'a){\cdot}(PRJ('b){\cdot}x) = PRJ('a){\cdot}x$
⟨*proof*⟩

lemma *coerce-idem* [*simp*]:
$DEFL('a) \sqsubseteq DEFL('b)$
$\Longrightarrow COERCE('b,'c){\cdot}(COERCE('a,'b){\cdot}x) = COERCE('a,'c){\cdot}x$
⟨*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-map* :: $('a \to 'c) \to ('b \to 'd) \to 'a \otimes 'b \to 'c \otimes 'd$. 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-map*·*coerce*
⟨*proof*⟩

lemma *coerce-sfun*: *coerce* = *sfun-map*·*coerce*·*coerce*
⟨*proof*⟩

lemma *coerce-cfun'*: *coerce* = *cfun-map*·*coerce*·*coerce*
⟨*proof*⟩

lemma *coerce-ssum*: *coerce* = *ssum-map*·*coerce*·*coerce*
⟨*proof*⟩

lemma *coerce-sprod*: *coerce* = *sprod-map*·*coerce*·*coerce*
⟨*proof*⟩

lemma *coerce-prod*: *coerce* = *prod-map*·*coerce*·*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*·(*up*·*x*) = *up*·(*coerce*·*x*)
⟨*proof*⟩

lemma *coerce-sinl* [*simp*]: *coerce*·(*sinl*·*x*) = *sinl*·(*coerce*·*x*)
⟨*proof*⟩

lemma *coerce-sinr* [*simp*]: *coerce*·(*sinr*·*x*) = *sinr*·(*coerce*·*x*)
⟨*proof*⟩

lemma *coerce-spair* [*simp*]: *coerce*·(:*x*, *y*:) = (:*coerce*·*x*, *coerce*·*y*:)
⟨*proof*⟩

lemma *coerce-Pair* [*simp*]: *coerce*·(*x*, *y*) = (*coerce*·*x*, *coerce*·*y*)
⟨*proof*⟩

lemma *beta-coerce-cfun* [*simp*]: *coerce*·*f*·*x* = *coerce*·(*f*·(*coerce*·*x*))
⟨*proof*⟩

lemma *coerce-cfun*: *coerce*·*f* = *coerce oo f oo coerce*
⟨*proof*⟩

lemma *coerce-cfun-app* [*coerce-simp*]:
coerce·*f* = (Λ *x*. *coerce*·(*f*·(*coerce*·*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 `fmap :: (a -> b) -> f a -> 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 $f :: 'a \to 'a$ with a deflation $t ::$ *udom defl*: *isodefl f t* = (*cast*·*t* = *EMB*('*a*) *oo f oo PRJ*('*a*)).

class *prefunctor* = *tycon* +
fixes *fmapU* :: (*udom* → *udom*) → *udom*·'*a* → *udom*·'*a*::*tycon*
assumes *isodefl-fmapU*:

isodefl (*fmapU*·(*cast*·*t*)) (*TC*('*a*::*tycon*)·*t*)

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

class *functor* = *prefunctor* +
  assumes *fmapU-fmapU* [*coerce-simp*]:
    ⋀*f g* (*xs*::*udom*·'*a*::*tycon*).
      fmapU·*f*·(*fmapU*·*g*·*xs*) = *fmapU*·(Λ *x*. *f*·(*g*·*x*))·*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·(*TC*('*f*)·*t*) = *emb oo fmapU*·(*cast*·*t*) *oo PRJ*(*udom*·'*f*::*prefunctor*)
⟨*proof*⟩

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

lemma *cast-cast-below1*: *A* ⊑ *B* ⟹ *cast*·*A*·(*cast*·*B*·*x*) = *cast*·*A*·*x*
⟨*proof*⟩

lemma *cast-cast-below2*: *A* ⊑ *B* ⟹ *cast*·*B*·(*cast*·*A*·*x*) = *cast*·*A*·*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*·(Λ *x*. *x*) = *ID*
⟨*proof*⟩

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

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

lemma *fmap-fmap*:
fixes $xs :: 'a{\cdot}'f{::}functor$ **and** $g :: 'a \to 'b$ **and** $f :: 'b \to 'c$
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*: $fmap{\cdot}(f\ oo\ g) = fmap{\cdot}f\ oo\ fmap{\cdot}g$
⟨*proof*⟩

3.3 Derived properties of *fmap*

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

lemma *deflation-fmap*:
$deflation\ d \Longrightarrow deflation\ (fmap{\cdot}d)$
⟨*proof*⟩

lemma *ep-pair-fmap*:
$ep\text{-}pair\ e\ p \Longrightarrow ep\text{-}pair\ (fmap{\cdot}e)\ (fmap{\cdot}p)$
⟨*proof*⟩

lemma *fmap-strict*:
fixes $f :: 'a \to 'b$
assumes $f{\cdot}\bot = \bot$ **shows** $fmap{\cdot}f{\cdot}\bot = (\bot{::}'b{\cdot}'f{::}functor)$
⟨*proof*⟩

3.4 Proving that *fmap·coerce = coerce*

lemma *fmapU-cast-eq*:
$fmapU{\cdot}(cast{\cdot}A) =$
$PRJ(udom{\cdot}'f)\ oo\ cast{\cdot}(TC('f{::}functor){\cdot}A)\ oo\ emb$
⟨*proof*⟩

lemma *fmapU-cast-DEFL*:
$fmapU{\cdot}(cast{\cdot}DEFL('a)) =$
$PRJ(udom{\cdot}'f)\ oo\ cast{\cdot}DEFL('a{\cdot}'f{::}functor)\ oo\ emb$
⟨*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 *isodefl-fmap* [*domain-isodefl*]

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*:
    ⋀*f xs*. *fmapU*·*f*·*xs* = *bindU*·*xs*·(Λ *x*. *returnU*·(*f*·*x*))
  assumes *bindU-returnU*:
    ⋀*f x*. *bindU*·(*returnU*·*x*)·*f* = *f*·*x*
  assumes *bindU-bindU*:
    ⋀*xs f g*. *bindU*·(*bindU*·*xs*·*f*)·*g* = *bindU*·*xs*·(Λ *x*. *bindU*·(*f*·*x*)·*g*)

instance *monad* ⊆ *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* ⇒ ('*a* → '*b*·'*m*) ⇒ '*b*·'*m* (**infixl** ⋙ *55*)
  where *m* ⋙ *f* ≡ *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*·(Λ *x*. *fmapU*·*f*·(*k*·*x*))
⟨*proof*⟩

lemma *bindU-fmapU*:
bindU·(*fmapU*·*f*·*xs*)·*k* = *bindU*·*xs*·(Λ *x*. *k*·(*f*·*x*))
⟨*proof*⟩

4.3 Polymorphic versions of class assumptions

lemma *monad-fmap*:
fixes *xs* :: 'a·'m::*monad* **and** *f* :: 'a → 'b
shows *fmap*·*f*·*xs* = *xs* ⤜ (Λ *x*. *return*·(*f*·*x*))
⟨*proof*⟩

lemma *monad-left-unit* [*simp*]: (*return*·*x* ⤜ *f*) = (*f*·*x*)
⟨*proof*⟩

lemma *bind-bind*:
fixes *m* :: 'a·'m::*monad*
shows ((*m* ⤜ *f*) ⤜ *g*) = (*m* ⤜ (Λ *x*. *f*·*x* ⤜ *g*))
⟨*proof*⟩

4.4 Derived rules

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

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

lemma *fmap-return*: *fmap*·*f*·(*return*·*x*) = *return*·(*f*·*x*)
⟨*proof*⟩

lemma *fmap-bind*: *fmap*·*f*·(*bind*·*xs*·*k*) = *bind*·*xs*·(Λ *x*. *fmap*·*f*·(*k*·*x*))
⟨*proof*⟩

lemma *bind-fmap*: *bind*·(*fmap*·*f*·*xs*)·*k* = *bind*·*xs*·(Λ *x*. *k*·(*f*·*x*))
⟨*proof*⟩

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

lemma *bind-strict*:
assumes *k*·⊥ = ⊥ **shows** ⊥ ⤜ *k* = ⊥
⟨*proof*⟩

lemma *congruent-bind*:
(∀ *m*. *m* ⤜ *k1* = *m* ⤜ *k2*) = (*k1* = *k2*)
⟨*proof*⟩

4.5 Laws for join

definition *join* :: ('a·'m)·'m → 'a·'m::*monad*
where *join* ≡ Λ *m*. *m* ⤜ (Λ *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*]:
$COERCE('a \cdot 'm, 'b \cdot 'm :: monad) \cdot (return \cdot x) = return \cdot (COERCE('a, 'b) \cdot x)$
⟨*proof*⟩

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

lemma *bind-coerce* [*coerce-simp*]:
fixes $m :: 'a \cdot 'm :: monad$ **and** $k :: 'b \to 'c \cdot 'm$
shows $COERCE('a \cdot 'm, 'b \cdot 'm) \cdot m \gg\!= k = m \gg\!= (\Lambda\ x.\ k \cdot (COERCE('a, 'b) \cdot x))$
⟨*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* ⊆ *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* ≡ *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 *llist* :: *monad-zero-plus*
begin

fixrec *plusU-llist* :: *udom·llist* → *udom·llist* → *udom·llist*
  where *plusU-llist·LNil·ys* = *ys*
  | *plusU-llist·*(*LCons·x·xs*)*·ys* = *LCons·x·*(*plusU-llist·xs·ys*)

lemma *plusU-llist-strict* [*simp*]: *plusU·*⊥*·ys* = (⊥::*udom·llist*)
⟨*proof*⟩

fixrec *bindU-llist* :: *udom·llist* → (*udom* → *udom·llist*) → *udom·llist*
  where *bindU-llist·LNil·k* = *LNil*
  | *bindU-llist·*(*LCons·x·xs*)*·k* = *plusU·*(*k·x*)*·*(*bindU-llist·xs·k*)

lemma *bindU-llist-strict* [*simp*]: *bindU·*⊥*·k* = (⊥::*udom·llist*)
⟨*proof*⟩

definition *zeroU-llist-def*:
  zeroU = *LNil*

definition *returnU-llist-def*:
  returnU = (Λ *x*. *LCons·x·LNil*)

lemma *plusU-LNil-right*: *plusU·xs·LNil* = *xs*
⟨*proof*⟩

lemma *plusU-llist-assoc*:
  fixes *xs ys zs* :: *udom·llist*
  shows *plusU·*(*plusU·xs·ys*)*·zs* = *plusU·xs·*(*plusU·ys·zs*)
⟨*proof*⟩

lemma *bindU-plusU-llist*:
  fixes *xs ys* :: *udom·llist* **shows**
  bindU·(*plusU·xs·ys*)*·f* = *plusU·*(*bindU·xs·f*)*·*(*bindU·ys·f*)
⟨*proof*⟩

instance ⟨*proof*⟩

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*]:
  $fmap{\cdot}f{\cdot}(\bot::'a{\cdot}llist) = \bot$
  $fmap{\cdot}f{\cdot}LNil = LNil$
  $fmap{\cdot}f{\cdot}(LCons{\cdot}x{\cdot}xs) = LCons{\cdot}(f{\cdot}x){\cdot}(fmap{\cdot}f{\cdot}xs)$
⟨*proof*⟩

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

lemma *bind-llist-simps* [*simp*]:
  $bind{\cdot}(\bot::'a{\cdot}llist){\cdot}f = \bot$
  $bind{\cdot}LNil{\cdot}f = LNil$
  $bind{\cdot}(LCons{\cdot}x{\cdot}xs){\cdot}f = mplus{\cdot}(f{\cdot}x){\cdot}(bind{\cdot}xs{\cdot}f)$
⟨*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*]:
  $join{\cdot}(\bot::'a{\cdot}llist{\cdot}llist) = \bot$
  $join{\cdot}LNil = LNil$
  $join{\cdot}(LCons{\cdot}xs{\cdot}xss) = mplus{\cdot}xs{\cdot}(join{\cdot}xss)$
⟨*proof*⟩

end

9 Maybe monad

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

9.1 Type definition

tycondef *'a·maybe* = *Nothing* | *Just* (**lazy** *'a*)

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

lemma *coerce-Nothing* [*simp*]: *coerce·Nothing* = *Nothing*
⟨*proof*⟩

lemma *coerce-Just* [*simp*]: *coerce·*(*Just·x*) = *Just·*(*coerce·x*)
⟨*proof*⟩

lemma *fmapU-maybe-simps* [*simp*]:
fmapU·f·(⊥::*udom·maybe*) = ⊥
fmapU·f·Nothing = *Nothing*
fmapU·f·(*Just·x*) = *Just·*(*f·x*)
⟨*proof*⟩

9.2 Class instance proofs

instance *maybe* :: *functor*
⟨*proof*⟩

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

fixrec *plusU-maybe* :: *udom·maybe* → *udom·maybe* → *udom·maybe*
where *plusU-maybe·Nothing·ys* = *ys*
| *plusU-maybe·*(*Just·x*)*·ys* = *Just·x*

lemma *plusU-maybe-strict* [*simp*]: *plusU·*⊥*·ys* = (⊥::*udom·maybe*)
⟨*proof*⟩

fixrec *bindU-maybe* :: *udom·maybe* → (*udom* → *udom·maybe*) → *udom·maybe*
where *bindU-maybe·Nothing·k* = *Nothing*
| *bindU-maybe·*(*Just·x*)*·k* = *k·x*

lemma *bindU-maybe-strict* [*simp*]: *bindU·*⊥*·k* = (⊥::*udom·maybe*)
⟨*proof*⟩

definition *zeroU-maybe-def*:
zeroU = *Nothing*

definition *returnU-maybe-def*:
returnU = *Just*

lemma *plusU-Nothing-right*: *plusU·xs·Nothing* = *xs*
⟨*proof*⟩

lemma *bindU-plusU-maybe*:
fixes *xs ys* :: *udom·maybe* **shows**
bindU·(plusU·xs·ys)·f = *plusU·(bindU·xs·f)·(bindU·ys·f)*
⟨*proof*⟩

instance ⟨*proof*⟩

end

9.3 Transfer properties to polymorphic versions

lemma *fmap-maybe-simps* [*simp*]:
fmap·f·(⊥::'a·maybe) = ⊥
fmap·f·Nothing = *Nothing*
fmap·f·(Just·x) = *Just·(f·x)*
⟨*proof*⟩

lemma *fplus-maybe-simps* [*simp*]:
fplus·(⊥::'a·maybe)·ys = ⊥
fplus·Nothing·ys = *ys*
fplus·(Just·x)·ys = *Just·x*
⟨*proof*⟩

lemma *fplus-Nothing-right* [*simp*]:
fplus·m·Nothing = *m*
⟨*proof*⟩

lemma *bind-maybe-simps* [*simp*]:
bind·(⊥::'a·maybe)·f = ⊥
bind·Nothing·f = *Nothing*
bind·(Just·x)·f = *f·x*
⟨*proof*⟩

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

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

lemma *join-maybe-simps* [*simp*]:
join·(⊥::'a·maybe·maybe) = ⊥
join·Nothing = *Nothing*
join·(Just·xs) = *xs*
⟨*proof*⟩

9.4 Maybe is not in *monad-plus*

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

lemma *maybe-counterexample1*:

$[\![a = Just{\cdot}x;\ b = \bot;\ k{\cdot}x = Nothing]\!]$
$\Longrightarrow fplus{\cdot}a{\cdot}b \gg\!= k \neq fplus{\cdot}(a \gg\!= k){\cdot}(b \gg\!= k)$
⟨*proof*⟩

lemma *maybe-counterexample2*:
$[\![a = Just{\cdot}x;\ b = Just{\cdot}y;\ k{\cdot}x = Nothing;\ k{\cdot}y = Just{\cdot}z]\!]$
$\Longrightarrow fplus{\cdot}a{\cdot}b \gg\!= k \neq fplus{\cdot}(a \gg\!= k){\cdot}(b \gg\!= k)$
⟨*proof*⟩

end

10 Error monad

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

10.1 Type definition

tycondef $'a{\cdot}'e\ error = Err\ (\textbf{lazy}\ 'e) \mid Ok\ (\textbf{lazy}\ 'a)$

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

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

lemma *coerce-Ok* [*simp*]: $coerce{\cdot}(Ok{\cdot}m) = Ok{\cdot}(coerce{\cdot}m)$
⟨*proof*⟩

lemma *fmapU-error-simps* [*simp*]:
$fmapU{\cdot}f{\cdot}(\bot::udom{\cdot}'a\ error) = \bot$
$fmapU{\cdot}f{\cdot}(Err{\cdot}e) = Err{\cdot}e$
$fmapU{\cdot}f{\cdot}(Ok{\cdot}x) = Ok{\cdot}(f{\cdot}x)$
⟨*proof*⟩

10.2 Monad class instance

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

definition
$returnU = Ok$

fixrec $bindU\text{-}error :: udom{\cdot}'a\ error \to (udom \to udom{\cdot}'a\ error) \to udom{\cdot}'a\ error$
where $bindU\text{-}error{\cdot}(Err{\cdot}e){\cdot}f = Err{\cdot}e$
| $bindU\text{-}error{\cdot}(Ok{\cdot}x){\cdot}f = f{\cdot}x$

lemma *bindU-error-strict* [*simp*]: $bindU{\cdot}\bot{\cdot}k = (\bot::udom{\cdot}'a\ error)$

⟨*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*: ⋀*ys*. *mappend*·*mempty*·*ys* = *ys*
  assumes *mempty-right*: ⋀*xs*. *mappend*·*xs*·*mempty* = *xs*
  assumes *mappend-assoc*:
    ⋀*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-writer*·(*Writer*·*w*·*x*)·*f* =
(*case f*·*x of Writer*·*w'*·*y* ⇒ *Writer*·(*mappend*·*w*·*w'*)·*y*)

lemma *bindU-writer-strict* [*simp*]: *bindU*·⊥·*k* = (⊥::*udom*·'*a writer*)
⟨*proof*⟩

definition
returnU = *Writer*·*mempty*

instance ⟨*proof*⟩

end

11.4 Transfer properties to polymorphic versions

lemma *fmap-writer-simps* [*simp*]:
fmap·*f*·(⊥::'*a*·'*w writer*) = ⊥
fmap·*f*·(*Writer*·*w*·*x* :: '*a*·'*w writer*) = *Writer*·*w*·(*f*·*x*)
⟨*proof*⟩

lemma *return-writer-def*: *return* = *Writer*·*mempty*
⟨*proof*⟩

lemma *bind-writer-simps* [*simp*]:
bind·(⊥ :: '*a*·'*w*::*monoid writer*)·*f* = ⊥
bind·(*Writer*·*w*·*x* :: '*a*·'*w*::*monoid writer*)·*k* =
(*case k*·*x of Writer*·*w'*·*y* ⇒ *Writer*·(*mappend*·*w*·*w'*)·*y*)
⟨*proof*⟩

lemma *join-writer-simps* [*simp*]:
join·⊥ = (⊥ :: '*a*·'*w*::*monoid writer*)
join·(*Writer*·*w*·(*Writer*·*w'*·*x*)) = *Writer*·(*mappend*·*w*·*w'*)·*x*
⟨*proof*⟩

11.5 Extra operations

definition *tell* :: '*w* → *unit*·('*w*::*monoid writer*)
where *tell* = (Λ *w*. *Writer*·*w*·())

end

12 Binary tree monad

theory *Binary-Tree-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 → (udom → udom·btree) → 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*]:
  $bind{\cdot}(\bot::'a{\cdot}btree){\cdot}k = \bot$
  $bind{\cdot}(Leaf{\cdot}x){\cdot}k = k{\cdot}x$
  $bind{\cdot}(Node{\cdot}xs{\cdot}ys){\cdot}k = Node{\cdot}(bind{\cdot}xs{\cdot}k){\cdot}(bind{\cdot}ys{\cdot}k)$
⟨*proof*⟩

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

lemma *join-btree-simps* [*simp*]:
  $join{\cdot}(\bot::'a{\cdot}btree{\cdot}btree) = \bot$
  $join{\cdot}(Leaf{\cdot}xs) = xs$
  $join{\cdot}(Node{\cdot}xss{\cdot}yss) = Node{\cdot}(join{\cdot}xss){\cdot}(join{\cdot}yss)$
⟨*proof*⟩

end

13 Lift monad

theory *Lift-Monad*
imports *Monad*
begin

13.1 Type definition

tycondef $'a{\cdot}lifted = Lifted$ (**lazy** $'a$)

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

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

lemma *fmapU-lifted-simps* [*simp*]:
  $fmapU{\cdot}f{\cdot}(\bot::udom{\cdot}lifted) = \bot$
  $fmapU{\cdot}f{\cdot}(Lifted{\cdot}x) = Lifted{\cdot}(f{\cdot}x)$
⟨*proof*⟩

13.2 Class instance proofs

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

instantiation *lifted* :: *monad*
begin

fixrec $bindU\text{-}lifted :: udom{\cdot}lifted \to (udom \to udom{\cdot}lifted) \to udom{\cdot}lifted$

where *bindU-lifted*·(*Lifted*·*x*)·*k* = *k*·*x*

lemma *bindU-lifted-strict* [*simp*]: *bindU*·⊥·*k* = (⊥::*udom*·*lifted*)
⟨*proof*⟩

definition *returnU-lifted-def*:
returnU = *Lifted*

instance ⟨*proof*⟩

end

13.3 Transfer properties to polymorphic versions

lemma *fmap-lifted-simps* [*simp*]:
fmap·*f*·(⊥::'*a*·*lifted*) = ⊥
fmap·*f*·(*Lifted*·*x*) = *Lifted*·(*f*·*x*)
⟨*proof*⟩

lemma *bind-lifted-simps* [*simp*]:
bind·(⊥::'*a*·*lifted*)·*f* = ⊥
bind·(*Lifted*·*x*)·*f* = *f*·*x*
⟨*proof*⟩

lemma *return-lifted-def*: *return* = *Lifted*
⟨*proof*⟩

lemma *join-lifted-simps* [*simp*]:
join·(⊥::'*a*·*lifted*·*lifted*) = ⊥
join·(*Lifted*·*xs*) = *xs*
⟨*proof*⟩

end

14 Resumption monad transformer

theory *Resumption-Transformer*
imports *Monad-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*]:
fmap·f·(⊥::*'a·'f*::*functor resT*) = ⊥
fmap·f·(*Done·x* :: *'a·'f*::*functor resT*) = *Done·*(*f·x*)
fmap·f·(*More·m* :: *'a·'f*::*functor resT*) = *More·*(*fmap·*(*fmap·f*)·*m*)
⟨*proof*⟩

lemma *return-resT-def*: *return* = *Done*
⟨*proof*⟩

lemma *bind-resT-simps* [*simp*]:
bind·(⊥ :: *'a·'f*::*functor resT*)·*f* = ⊥
bind·(*Done·x* :: *'a·'f*::*functor resT*)·*f* = *f·x*
bind·(*More·m* :: *'a·'f*::*functor resT*)·*f* = *More·*(*fmap·*(Λ *r*. *bind·r·f*)·*m*)
⟨*proof*⟩

lemma *join-resT-simps* [*simp*]:
join·⊥ = (⊥ :: *'a·'f*::*functor resT*)
join·(*Done·x*) = *x*
join·(*More·m*) = *More·*(*fmap·join·m*)
⟨*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* → *'b* → *'c*) → *'a·*(*'m*::*functor-plus*) *resT* → *'b·'m resT* → *'c·'m resT*
where *zipRT-Done-Done*:
zipRT·f·(*Done·x*)·(*Done·y*) = *Done·*(*f·x·y*)
| *zipRT-Done-More*:
zipRT·f·(*Done·x*)·(*More·b*) =
More·(*fmap·*(Λ *r*. *zipRT·f·*(*Done·x*)·*r*)·*b*)
| *zipRT-More-Done*:
zipRT·f·(*More·a*)·(*Done·y*) =
More·(*fmap·*(Λ *r*. *zipRT·f·r·*(*Done·y*))·*a*)
| *zipRT-More-More*:
zipRT·f·(*More·a*)·(*More·b*) =
More·(*fplus·*(*fmap·*(Λ *r*. *zipRT·f·*(*More·a*)·*r*)·*b*)
·(*fmap·*(Λ *r*. *zipRT·f·r·*(*More·b*))·*a*))

lemma *zipRT-strict1* [*simp*]: *zipRT·f·*⊥·*r* = ⊥
⟨*proof*⟩

lemma *zipRT-strict2* [*simp*]: $zipRT{\cdot}f{\cdot}r{\cdot}\bot = \bot$
⟨*proof*⟩

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

Proofs that *zipRT* satisfies the applicative functor laws:

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

lemma *zipRT-identity*: $Done{\cdot}ID \diamond r = r$
  ⟨*proof*⟩

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

The associativity rule is the hard one!

lemma *zipRT-associativity*: $Done{\cdot}cfcomp \diamond r1 \diamond r2 \diamond r3 = r1 \diamond (r2 \diamond r3)$
⟨*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 $'a{\cdot}('f{::}functor,\ 's)\ stateT =$
  $StateT\ (runStateT :: 's \to ('a \times 's){\cdot}'f)$

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

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

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

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

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

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

15.1 Functor class instance

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

lemma *runStateT-fmapU* [*simp*]:
$runStateT{\cdot}(fmapU{\cdot}f{\cdot}m){\cdot}s =$
$fmap{\cdot}(\Lambda(x,\ s').\ (f{\cdot}x,\ s')){\cdot}(runStateT{\cdot}m{\cdot}s)$
⟨*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*:
$returnU = (\Lambda\ x.\ StateT{\cdot}(\Lambda\ s.\ return{\cdot}(x,\ s)))$

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

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

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

instance ⟨*proof*⟩

end

15.3 Monad zero instance

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

definition *zeroU-stateT-def*:
zeroU = *StateT*·(Λ *s*. *mzero*)

lemma *runStateT-zeroU* [*simp*]:
runStateT·*zeroU*·*s* = *mzero*
⟨*proof*⟩

instance ⟨*proof*⟩

end

15.4 Monad plus instance

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

definition *plusU-stateT-def*:
plusU = (Λ *a b*. *StateT*·(Λ *s*. *mplus*·(*runStateT*·*a*·*s*)·(*runStateT*·*b*·*s*)))

lemma *runStateT-plusU* [*simp*]:
runStateT·(*plusU*·*a*·*b*)·*s* =
mplus·(*runStateT*·*a*·*s*)·(*runStateT*·*b*·*s*)
⟨*proof*⟩

instance ⟨*proof*⟩

end

15.5 Monad zero plus instance

instance *stateT* :: (*monad-zero-plus*, *domain*) *monad-zero-plus*
⟨*proof*⟩

15.6 Transfer properties to polymorphic versions

lemma *coerce-csplit* [*coerce-simp*]:
shows *coerce*·(*csplit*·*f*·*p*) = *csplit*·(Λ *x y*. *coerce*·(*f*·*x*·*y*))·*p*
⟨*proof*⟩

lemma *csplit-coerce* [*coerce-simp*]:
fixes *p* :: *'a* × *'b*
shows *csplit*·*f*·(*COERCE*(*'a* × *'b*, *'c* × *'d*)·*p*) =

$csplit{\cdot}(\Lambda\ x\ y.\ f{\cdot}(COERCE('a,\ 'c){\cdot}x){\cdot}(COERCE('b,\ 'd){\cdot}y)){\cdot}p$
⟨*proof*⟩

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

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

lemma *return-stateT-def*:
$(return :: - \to 'a{\cdot}('m{::}monad,\ 's)\ stateT) =$
$(\Lambda\ x.\ StateT{\cdot}(\Lambda\ s.\ return{\cdot}(x,\ s)))$
⟨*proof*⟩

lemma *bind-stateT-def*:
$bind = (\Lambda\ m\ k.\ StateT{\cdot}(\Lambda\ s.\ runStateT{\cdot}m{\cdot}s \gg\!= (\Lambda\ (x,\ s').\ runStateT{\cdot}(k{\cdot}x){\cdot}s')))$
⟨*proof*⟩

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

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

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

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 $'a{\cdot}('f{::}functor,'e{::}domain)\ errorT =$
  $ErrorT\ (runErrorT :: ('a{\cdot}'e\ error){\cdot}'f)$

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

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

lemma *errorT-cases* [*case-names ErrorT*]:
  obtains k **where** $y = ErrorT{\cdot}k$
⟨*proof*⟩

lemma *ErrorT-runErrorT* [*simp*]: $ErrorT{\cdot}(runErrorT{\cdot}m) = m$
⟨*proof*⟩

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

lemma *errorT-eq-iff*:
  $a = b \longleftrightarrow runErrorT{\cdot}a = runErrorT{\cdot}b$
⟨*proof*⟩

lemma *errorT-eqI*:
  $runErrorT{\cdot}a = runErrorT{\cdot}b \Longrightarrow a = b$
⟨*proof*⟩

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

16.2 Functor class instance

lemma *fmap-error-def*: $fmap = error\text{-}map{\cdot}ID$
⟨*proof*⟩

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

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

instance $errorT :: (functor,\ domain)\ functor$

⟨*proof*⟩

16.3 Transfer properties to polymorphic versions

lemma *fmap-ErrorT* [*simp*]:
  fixes $f :: 'a \to 'b$ **and** $m :: 'a\cdot'e\ error\cdot('m::functor)$
  shows $fmap\cdot f\cdot(ErrorT\cdot m) = ErrorT\cdot(fmap\cdot(fmap\cdot f)\cdot m)$
⟨*proof*⟩

lemma *runErrorT-fmap* [*simp*]:
  fixes $f :: 'a \to 'b$ **and** $m :: 'a\cdot('m::functor,'e)\ errorT$
  shows $runErrorT\cdot(fmap\cdot f\cdot m) = fmap\cdot(fmap\cdot f)\cdot(runErrorT\cdot m)$
⟨*proof*⟩

lemma *errorT-fmap-strict* [*simp*]:
  shows $fmap\cdot f\cdot(\bot::'a\cdot('m::monad,'e)\ errorT) = \bot$
⟨*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 $'a\cdot('m,'e)\ errorT$ 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 $unitET :: 'a \to 'a\cdot('m::monad,'e)\ errorT$
  where $unitET = (\Lambda\ x.\ ErrorT\cdot(return\cdot(Ok\cdot x)))$

definition $bindET :: 'a\cdot('m::monad,'e)\ errorT \to$
    $('a \to 'b\cdot('m,'e)\ errorT) \to 'b\cdot('m,'e)\ errorT$
  where $bindET = (\Lambda\ m\ k.\ ErrorT\cdot(bind\cdot(runErrorT\cdot m)\cdot$
    $(\Lambda\ n.\ case\ n\ of\ Err\cdot e \Rightarrow return\cdot(Err\cdot e) \mid Ok\cdot x \Rightarrow runErrorT\cdot(k\cdot x))))$

definition $liftET :: 'a\cdot'm::monad \to 'a\cdot('m,'e)\ errorT$
  where $liftET = (\Lambda\ m.\ ErrorT\cdot(fmap\cdot Ok\cdot m))$

definition $throwET :: 'e \to 'a\cdot('m::monad,'e)\ errorT$
  where $throwET = (\Lambda\ e.\ ErrorT\cdot(return\cdot(Err\cdot e)))$

definition $catchET :: 'a\cdot('m::monad,'e)\ errorT \to$
    $('e \to 'a\cdot('m,'e)\ errorT) \to 'a\cdot('m,'e)\ errorT$
  where $catchET = (\Lambda\ m\ h.\ ErrorT\cdot(bind\cdot(runErrorT\cdot m)\cdot(\Lambda\ n.\ case\ n\ of$
    $Err\cdot e \Rightarrow runErrorT\cdot(h\cdot e) \mid Ok\cdot x \Rightarrow return\cdot(Ok\cdot x))))$

definition $fmapET :: ('a \to 'b) \to$

$'a{\cdot}('m{::}monad,'e)\ errorT \to 'b{\cdot}('m,'e)\ errorT$
where $fmapET = (\Lambda\ f\ m.\ bindET{\cdot}m{\cdot}(\Lambda\ x.\ unitET{\cdot}(f{\cdot}x)))$

lemma *runErrorT-unitET* [*simp*]:
$runErrorT{\cdot}(unitET{\cdot}x) = return{\cdot}(Ok{\cdot}x)$
⟨*proof*⟩

lemma *runErrorT-bindET* [*simp*]:
$runErrorT{\cdot}(bindET{\cdot}m{\cdot}k) = bind{\cdot}(runErrorT{\cdot}m){\cdot}$
$(\Lambda\ n.\ case\ n\ of\ Err{\cdot}e \Rightarrow return{\cdot}(Err{\cdot}e) \mid Ok{\cdot}x \Rightarrow runErrorT{\cdot}(k{\cdot}x))$
⟨*proof*⟩

lemma *runErrorT-liftET* [*simp*]:
$runErrorT{\cdot}(liftET{\cdot}m) = fmap{\cdot}Ok{\cdot}m$
⟨*proof*⟩

lemma *runErrorT-throwET* [*simp*]:
$runErrorT{\cdot}(throwET{\cdot}e) = return{\cdot}(Err{\cdot}e)$
⟨*proof*⟩

lemma *runErrorT-catchET* [*simp*]:
$runErrorT{\cdot}(catchET{\cdot}m{\cdot}h) =$
$bind{\cdot}(runErrorT{\cdot}m){\cdot}(\Lambda\ n.\ case\ n\ of$
$Err{\cdot}e \Rightarrow runErrorT{\cdot}(h{\cdot}e) \mid Ok{\cdot}x \Rightarrow return{\cdot}(Ok{\cdot}x))$
⟨*proof*⟩

lemma *runErrorT-fmapET* [*simp*]:
$runErrorT{\cdot}(fmapET{\cdot}f{\cdot}m) =$
$bind{\cdot}(runErrorT{\cdot}m){\cdot}(\Lambda\ n.\ case\ n\ of$
$Err{\cdot}e \Rightarrow return{\cdot}(Err{\cdot}e) \mid Ok{\cdot}x \Rightarrow return{\cdot}(Ok{\cdot}(f{\cdot}x)))$
⟨*proof*⟩

16.5 Laws

lemma *bindET-unitET* [*simp*]:
$bindET{\cdot}(unitET{\cdot}x){\cdot}k = k{\cdot}x$
⟨*proof*⟩

lemma *catchET-unitET* [*simp*]:
$catchET{\cdot}(unitET{\cdot}x){\cdot}h = unitET{\cdot}x$
⟨*proof*⟩

lemma *catchET-throwET* [*simp*]:
$catchET{\cdot}(throwET{\cdot}e){\cdot}h = h{\cdot}e$
⟨*proof*⟩

lemma *liftET-return*:
$liftET{\cdot}(return{\cdot}x) = unitET{\cdot}x$
⟨*proof*⟩

lemma *liftET-bind*:
liftET·(*bind*·*m*·*k*) = *bindET*·(*liftET*·*m*)·(*liftET oo k*)
⟨*proof*⟩

lemma *bindET-throwET*:
bindET·(*throwET*·*e*)·*k* = *throwET*·*e*
⟨*proof*⟩

lemma *bindET-bindET*:
bindET·(*bindET*·*m*·*h*)·*k* = *bindET*·*m*·(Λ *x*. *bindET*·(*h*·*x*)·*k*)
⟨*proof*⟩

lemma *fmapET-fmapET*:
fmapET·*f*·(*fmapET*·*g*·*m*) = *fmapET*·(Λ *x*. *f*·(*g*·*x*))·*m*
⟨*proof*⟩

Right unit monad law is not satisfied in general.

lemma *bindET-unitET-right-counterexample*:
fixes *m* :: *'a*·(*'m*::*monad*,*'e*) *errorT*
assumes *m* = *ErrorT*·(*return*·⊥)
assumes *return*·⊥ ≠ (⊥ :: (*'a*·*'e error*)·*'m*)
shows *bindET*·*m*·*unitET* ≠ *m*
⟨*proof*⟩

Right unit is satisfied for inner monads with strict return.

lemma *bindET-unitET-right-restricted*:
fixes *m* :: *'a*·(*'m*::*monad*,*'e*) *errorT*
assumes *return*·⊥ = (⊥ :: (*'a*·*'e error*)·*'m*)
shows *bindET*·*m*·*unitET* = *m*
⟨*proof*⟩

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*·(*'m*::*monad*, *'e*) *errorT* ⇒ *bool*
where *invar-bottom*: *invar* ⊥
| *invar-lub*: ⋀*Y*. ⟦*chain Y*; ⋀*i*. *invar* (*Y i*)⟧ ⟹ *invar* (⨆ *i*. *Y i*)
| *invar-unitET*: ⋀*x*. *invar* (*unitET*·*x*)
| *invar-bindET*: ⋀*m k*. ⟦*invar m*; ⋀*x*. *invar* (*k*·*x*)⟧ ⟹ *invar* (*bindET*·*m*·*k*)
| *invar-throwET*: ⋀*e*. *invar* (*throwET*·*e*)
| *invar-catchET*: ⋀*m h*. ⟦*invar m*; ⋀*e*. *invar* (*h*·*e*)⟧ ⟹ *invar* (*catchET*·*m*·*h*)
| *invar-liftET*: ⋀*m*. *invar* (*liftET*·*m*)

Right unit is satisfied for arguments built from standard functions.

lemma *bindET-unitET-right-invar*:

assumes *invar m*
shows *bindET·m·unitET = m*
⟨*proof*⟩

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

lemma *errorT-monad-fmap-invar*:
fixes *f* :: *'a → 'b* **and** *m* :: *'a·('m::monad,'e) errorT*
assumes *invar m*
shows *fmap·f·m = bindET·m·(Λ x. unitET·(f·x))*
⟨*proof*⟩

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·('m::monad, 'e) errorT ⇒ bool*
where *invar' m ⟷ fmapET·ID·m = m*

All standard operations preserve the invariant.

lemma *invar'-unitET*: *invar' (unitET·x)*
⟨*proof*⟩

lemma *invar'-fmapET*: *invar' m ⟹ invar' (fmapET·f·m)*
⟨*proof*⟩

lemma *invar'-bindET*: *⟦invar' m; ⋀x. invar' (k·x)⟧ ⟹ invar' (bindET·m·k)*
⟨*proof*⟩

lemma *invar'-throwET*: *invar' (throwET·e)*
⟨*proof*⟩

lemma *invar'-catchET*: *⟦invar' m; ⋀e. invar' (h·e)⟧ ⟹ invar' (catchET·m·h)*
⟨*proof*⟩

lemma *invar'-liftET*: *invar' (liftET·m)*
⟨*proof*⟩

lemma *invar'-bottom*: *invar' ⊥*
⟨*proof*⟩

lemma *adm-invar'*: *adm invar'*
⟨*proof*⟩

All monad laws are preserved by values satisfying the invariant.

lemma *bindET-fmapET-unitET*:
shows *bindET·(fmapET·f·m)·unitET = fmapET·f·m*
⟨*proof*⟩

lemma *invar′-right-unit*: $invar'\ m \Longrightarrow bindET{\cdot}m{\cdot}unitET = m$
⟨*proof*⟩

lemma *invar′-monad-fmap*:
$invar'\ m \Longrightarrow fmapET{\cdot}f{\cdot}m = bindET{\cdot}m{\cdot}(\Lambda\ x.\ unitET{\cdot}(f{\cdot}x))$
⟨*proof*⟩

lemma *invar′-bind-assoc*:
$[\![invar'\ m; \bigwedge x.\ invar'\ (f{\cdot}x); \bigwedge y.\ invar'\ (g{\cdot}y)]\!]$
$\Longrightarrow bindET{\cdot}(bindET{\cdot}m{\cdot}f){\cdot}g = bindET{\cdot}m{\cdot}(\Lambda\ x.\ bindET{\cdot}(f{\cdot}x){\cdot}g)$
⟨*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 $('a{\cdot}('m{::}functor,'w)\ writerT =$
$WriterT\ (runWriterT :: ('a{\cdot}'w\ writer){\cdot}'m)$

lemma *coerce-writerT-abs* [*simp*]:
$coerce{\cdot}(writerT\text{-}abs{\cdot}x) = writerT\text{-}abs{\cdot}(coerce{\cdot}x)$
⟨*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** $y = WriterT{\cdot}k$

⟨*proof*⟩

lemma *WriterT-runWriterT* [*simp*]: *WriterT*·(*runWriterT*·*m*) = *m*
⟨*proof*⟩

lemma *writerT-induct* [*case-names WriterT*]:
  fixes *P* :: '*a*·('*f*::*functor*,'*e*) *writerT* ⇒ *bool*
  assumes ⋀*k*. *P* (*WriterT*·*k*)
  shows *P y*
⟨*proof*⟩

lemma *writerT-eq-iff*:
  a = *b* ⟷ *runWriterT*·*a* = *runWriterT*·*b*
⟨*proof*⟩

lemma *writerT-below-iff*:
  a ⊑ *b* ⟷ *runWriterT*·*a* ⊑ *runWriterT*·*b*
⟨*proof*⟩

lemma *writerT-eqI*:
  runWriterT·*a* = *runWriterT*·*b* ⟹ *a* = *b*
⟨*proof*⟩

lemma *writerT-belowI*:
  runWriterT·*a* ⊑ *runWriterT*·*b* ⟹ *a* ⊑ *b*
⟨*proof*⟩

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

17.2 Functor class instance

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

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

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

instance *writerT* :: (*functor*, *domain*) *functor*
⟨*proof*⟩

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 $unitWT :: 'a \to 'a{\cdot}('m{::}monad,'w{::}monoid)\ writerT$
  where $unitWT = (\Lambda\ x.\ WriterT{\cdot}(return{\cdot}(Writer{\cdot}mempty{\cdot}x)))$

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

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

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

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

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

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

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

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

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

runWriterT·(*tellWT*·*x*·*w*) = *return*·(*Writer*·*w*·*x*)
⟨*proof*⟩

lemma *runWriterT-fmapWT* [*simp*]:
runWriterT·(*fmapWT*·*f*·*m*) =
runWriterT·*m* ⤜ (Λ (*Writer*·*w*·*x*). *return*·(*Writer*·*w*·(*f*·*x*)))
⟨*proof*⟩

17.4 Laws

The *liftWT* function maps *return* and *bind* on the inner monad to *unitWT* and *bindWT*, as expected.

lemma *liftWT-return*:
liftWT·(*return*·*x*) = *unitWT*·*x*
⟨*proof*⟩

lemma *liftWT-bind*:
liftWT·(*bind*·*m*·*k*) = *bindWT*·(*liftWT*·*m*)·(*liftWT oo k*)
⟨*proof*⟩

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

lemma *fmapWT-fmapWT*:
fmapWT·*f*·(*fmapWT*·*g*·*m*) = *fmapWT*·(Λ *x*. *f*·(*g*·*x*))·*m*
⟨*proof*⟩

lemma *fmapWT-unitWT*:
fmapWT·*f*·(*unitWT*·*x*) = *unitWT*·(*f*·*x*)
⟨*proof*⟩

lemma *fmapWT-bindWT*:
fmapWT·*f*·(*bindWT*·*m*·*k*) = *bindWT*·*m*·(Λ *x*. *fmapWT*·*f*·(*k*·*x*))
⟨*proof*⟩

lemma *bindWT-fmapWT*:
bindWT·(*fmapWT*·*f*·*m*)·*k* = *bindWT*·*m*·(Λ *x*. *k*·(*f*·*x*))
⟨*proof*⟩

The left unit monad law is not satisfied in general.

lemma *bindWT-unitWT-counterexample*:
fixes *k* :: *'a* → *'b*·(*'m*::*monad*,*'w*::*monoid*) *writerT*
assumes *1*: *k*·*x* = *WriterT*·(*return*·⊥)
assumes *2*: *return*·⊥ ≠ (⊥ :: (*'b*·*'w writer*)·*'m*::*monad*)
shows *bindWT*·(*unitWT*·*x*)·*k* ≠ *k*·*x*
⟨*proof*⟩

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

lemma *bindWT-unitWT-restricted*:
fixes $k :: 'a \to 'b\cdot('m{::}monad,'w{::}monoid)\ writerT$
assumes $return\cdot\bot = (\bot :: ('b\cdot'w\ writer)\cdot'm)$
shows $bindWT\cdot(unitWT\cdot x)\cdot k = k\cdot x$
⟨*proof*⟩

The associativity of *bindWT* holds unconditionally.

lemma *bindWT-bindWT*:
$bindWT\cdot(bindWT\cdot m\cdot h)\cdot k = bindWT\cdot m\cdot(\Lambda\ x.\ bindWT\cdot(h\cdot x)\cdot k)$
⟨*proof*⟩

The right unit monad law is not satisfied in general.

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

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

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

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

Right unit is satisfied for arguments built from standard functions.

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

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

lemma *writerT-left-unit-invar-lemma*:
  assumes *invar m*
  shows *runWriterT*·*m* ⤜ (Λ (*Writer*·*w*·*x*). *return*·(*Writer*·*w*·*x*)) = *runWriterT*·*m*
⟨*proof*⟩

lemma *bindWT-unitWT-invar*:
  assumes *invar* (*k*·*x*)
  shows *bindWT*·(*unitWT*·*x*)·*k* = *k*·*x*
⟨*proof*⟩

17.6 Invariant expressed as a deflation

definition *invar′* :: 'a·('m::*monad*, 'w::*monoid*) *writerT* ⇒ *bool*
  where *invar′ m* ⟷ *fmapWT*·*ID*·*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*: ⟦*invar′ m*; ⋀*x*. *invar′* (*k*·*x*)⟧ ⟹ *invar′* (*bindWT*·*m*·*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*:
  bindWT·(*unitWT*·*x*)·(Λ *x*. *fmapWT*·*f*·(*k*·*x*))
    = *fmapWT*·*f*·(*k*·*x*)
⟨*proof*⟩

Right unit is satisfied for arguments built from fmap.

lemma *bindWT-fmapWT-unitWT*:
  shows *bindWT*·(*fmapWT*·*f*·*m*)·*unitWT* = *fmapWT*·*f*·*m*
⟨*proof*⟩

All monad laws are preserved by values satisfying the invariant.

lemma *invar′-right-unit*: $invar'\ m \Longrightarrow bindWT{\cdot}m{\cdot}unitWT = m$
⟨*proof*⟩

lemma *invar′-monad-fmap*:
$invar'\ m \Longrightarrow fmapWT{\cdot}f{\cdot}m = bindWT{\cdot}m{\cdot}(\Lambda\ x.\ unitWT{\cdot}(f{\cdot}x))$
⟨*proof*⟩

lemma *invar′-bind-assoc*:
$[\![invar'\ m;\ \bigwedge x.\ invar'\ (f{\cdot}x);\ \bigwedge y.\ invar'\ (g{\cdot}y)]\!]$
$\Longrightarrow bindWT{\cdot}(bindWT{\cdot}m{\cdot}f){\cdot}g = bindWT{\cdot}m{\cdot}(\Lambda\ x.\ bindWT{\cdot}(f{\cdot}x){\cdot}g)$
⟨*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.