

Tycon: Type Constructor Classes and Monad Transformers

Brian Huffman

March 17, 2025

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].

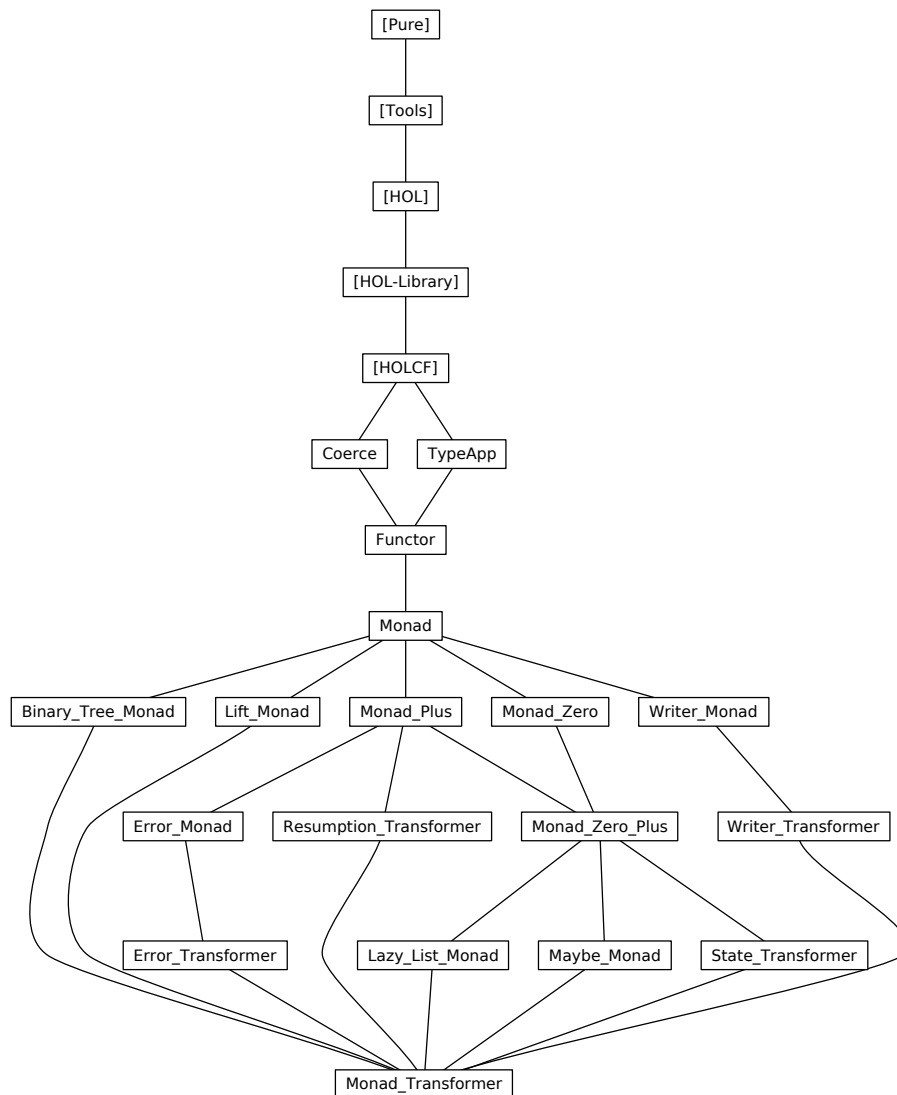
Based on the hierarchy of type classes in Haskell, we define classes for functors, monads, monad-plus, etc. Each one includes all the standard laws as axioms. We also provide a new user command, *tycondef*, for defining new type constructors in HOLCF. Using *tycondef*, we instantiate the type class hierarchy 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>	8
2.3	Coercing various datatypes	8
2.4	Simplifying coercions	9
3	Functor Class	10
3.1	Class definition	10
3.2	Polymorphic functor laws	11
3.3	Derived properties of <i>fmap</i>	12
3.4	Proving that <i>fmap.coerce</i> = <i>coerce</i>	13
3.5	Lemmas for reasoning about coercion	14
3.6	Configuration of Domain package	14
3.7	Configuration of the Tycon package	14

4	Monad Class	14
4.1	Class definition	15
4.2	Naturality of bind and return	16
4.3	Polymorphic versions of class assumptions	16
4.4	Derived rules	16
4.5	Laws for join	17
4.6	Equivalence of monad laws and fmap/join laws	17
4.7	Simplification of coercions	18
5	Monad-Zero Class	18
6	Monad-Plus Class	19
7	Monad-Zero-Plus Class	20
8	Lazy list monad	21
8.1	Type definition	22
8.2	Class instances	22
8.3	Transfer properties to polymorphic versions	24
9	Maybe monad	25
9.1	Type definition	25
9.2	Class instance proofs	26
9.3	Transfer properties to polymorphic versions	27
9.4	Maybe is not in <i>monad-plus</i>	28
10	Error monad	28
10.1	Type definition	28
10.2	Monad class instance	29
10.3	Transfer properties to polymorphic versions	30
11	Writer monad	31
11.1	Monoid class	31
11.2	Writer monad type	31
11.3	Class instance proofs	32
11.4	Transfer properties to polymorphic versions	32
11.5	Extra operations	33
12	Binary tree monad	33
12.1	Type definition	33
12.2	Class instance proofs	34
12.3	Transfer properties to polymorphic versions	35

13 Lift monad	35
13.1 Type definition	35
13.2 Class instance proofs	36
13.3 Transfer properties to polymorphic versions	36
14 Resumption monad transformer	37
14.1 Type definition	37
14.2 Class instance proofs	38
14.3 Transfer properties to polymorphic versions	39
14.4 Nondeterministic interleaving	39
15 State monad transformer	41
15.1 Functor class instance	42
15.2 Monad class instance	42
15.3 Monad zero instance	43
15.4 Monad plus instance	44
15.5 Monad zero plus instance	44
15.6 Transfer properties to polymorphic versions	44
16 Error monad transformer	45
16.1 Type definition	46
16.2 Functor class instance	47
16.3 Transfer properties to polymorphic versions	47
16.4 Monad operations	48
16.5 Laws	49
16.6 Error monad transformer invariant	50
16.7 Invariant expressed as a deflation	51
17 Writer monad transformer	53
17.1 Type definition	53
17.2 Functor class instance	54
17.3 Monad operations	55
17.4 Laws	56
17.5 Writer monad transformer invariant	58
17.6 Invariant expressed as a deflation	60



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 ( $\langle (1TC/(1'(-))) \rangle$ )
```

```
syntax-consts -TC  $\equiv$  tc
```

```
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 $'a \cdot 't$ 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* ($\langle \langle \dots \rangle \rangle$ [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*: $TC('t::tycon) \cdot DEFL('a) = DEFL('a \cdot 't)$
by (*rule DEFL-app [symmetric]*)

lemma *DEFL-app-mono* [*simp, intro*]:
 $DEFL('a) \sqsubseteq DEFL('b) \implies DEFL('a \cdot 't::tycon) \sqsubseteq DEFL('b \cdot 't)$
apply (*simp add: DEFL-app*)
apply (*erule monofun-cfun-arg*)
done
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
 $-emb :: type \Rightarrow logic \ (\langle \langle 1EMB / (1'(-)) \rangle \rangle)$
 $-prj :: type \Rightarrow logic \ (\langle \langle 1PRJ / (1'(-)) \rangle \rangle)$
 $-coerce :: type \Rightarrow type \Rightarrow logic \ (\langle \langle 1COERCE / (1'(-, / -)) \rangle \rangle)$

syntax-consts
 $-emb \equiv emb$ **and**
 $-prj \equiv prj$ **and**

$-coerce \equiv coerce$

translations

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

typed-print-translation

```
let
  fun emb-tr' (ctxt : Proof.context) (Type(-, [T, -])) [] =
    Syntax.const @{\syntax-const -emb} $ Syntax-Phases.term-of-typ ctxt T
  fun prj-tr' ctxt (Type(-, [-, T])) [] =
    Syntax.const @{\syntax-const -prj} $ Syntax-Phases.term-of-typ ctxt T
  fun coerce-tr' ctxt (Type(-, [T, U])) [] =
    Syntax.const @{\syntax-const -coerce} $
      Syntax-Phases.term-of-typ ctxt T $ Syntax-Phases.term-of-typ ctxt U
in
  [(@{\const-syntax emb}, emb-tr'),
   (@{\const-syntax prj}, prj-tr'),
   (@{\const-syntax coerce}, coerce-tr')]
end
>
```

lemma *beta-coerce*: $coerce.x = prj.(emb.x)$
by (*simp add: coerce-def*)

lemma *prj-emb*: $prj.(emb.x) = coerce.x$
by (*simp add: coerce-def*)

lemma *coerce-strict* [*simp*]: $coerce.\perp = \perp$
by (*simp add: coerce-def*)

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

lemma *coerce-eq-ID* [*simp*]: $COERCE('a, 'a) = ID$
by (*rule cfun-eqI, simp add: beta-coerce*)

lemma *coerce-eq-emb* [*simp*]: $COERCE('a, udom) = EMB('a)$
by (*rule cfun-eqI, simp add: beta-coerce*)

lemma *coerce-eq-prj* [*simp*]: $COERCE(udom, 'a) = PRJ('a)$
by (*rule cfun-eqI, simp add: beta-coerce*)

Cancellation rules

lemma *emb-coerce*:
 $DEFL('a) \sqsubseteq DEFL('b)$
 $\implies EMB('b).(COERCE('a, 'b).x) = EMB('a).x$
by (*simp add: beta-coerce emb-prj-emb*)

lemma *coerce-prj*:
 $DEFL('a) \sqsubseteq DEFL('b)$
 $\implies COERCE('b, 'a) \cdot (PRJ('b) \cdot x) = PRJ('a) \cdot x$
by (*simp add: beta-coerce prj-emb-prj*)

lemma *coerce-idem* [*simp*]:
 $DEFL('a) \sqsubseteq DEFL('b)$
 $\implies COERCE('b, 'c) \cdot (COERCE('a, 'b) \cdot x) = COERCE('a, 'c) \cdot x$
by (*simp add: beta-coerce emb-prj-emb*)

2.2 More lemmas about emb and prj

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

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

$DEFL(udom)$

lemma *below-DEFL-udom* [*simp*]: $A \sqsubseteq DEFL(udom)$
apply (*rule cast-below-imp-below*)
apply (*rule cast.belowI*)
apply (*simp add: cast-DEFL*)
done

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* :: $('a \rightarrow 'c) \rightarrow ('b \rightarrow 'd) \rightarrow 'a \otimes 'b \rightarrow 'c \otimes 'd$. Each of the several type constructors defined in HOLCF satisfies a similar property, with respect to its own map combinator.

lemma *coerce-u*: $coerce = u\text{-map} \cdot coerce$
apply (*rule cfun-eqI, simp add: coerce-def*)
apply (*simp add: emb-u-def prj-u-def liftemb-eq liftprj-eq*)
apply (*subst ep-pair.e-inverse [OF ep-pair-u]*)
apply (*simp add: u-map-map cfcomp1*)
done

lemma *coerce-sfun*: $coerce = sfun\text{-map} \cdot coerce \cdot coerce$
apply (*rule cfun-eqI, simp add: coerce-def*)
apply (*simp add: emb-sfun-def prj-sfun-def*)
apply (*subst ep-pair.e-inverse [OF ep-pair-sfun]*)
apply (*simp add: sfun-map-map cfcomp1*)
done


```

lemma coerce-cfun': coerce = cfun-map.coerce.coerce
apply (rule cfun-eqI, simp add: prj-emb [symmetric])
apply (simp add: emb-cfun-def prj-cfun-def)
apply (simp add: prj-emb coerce-sfun coerce-u)
apply (simp add: encode-cfun-map [symmetric])
done

```

```

lemma coerce-ssum: coerce = ssum-map.coerce.coerce
apply (rule cfun-eqI, simp add: coerce-def)
apply (simp add: emb-ssum-def prj-ssum-def)
apply (subst ep-pair.e-inverse [OF ep-pair-ssum])
apply (simp add: ssum-map-map cfcomp1)
done

```

```

lemma coerce-sprod: coerce = sprod-map.coerce.coerce
apply (rule cfun-eqI, simp add: coerce-def)
apply (simp add: emb-sprod-def prj-sprod-def)
apply (subst ep-pair.e-inverse [OF ep-pair-sprod])
apply (simp add: sprod-map-map cfcomp1)
done

```

```

lemma coerce-prod: coerce = prod-map.coerce.coerce
apply (rule cfun-eqI, simp add: coerce-def)
apply (simp add: emb-prod-def prj-prod-def)
apply (subst ep-pair.e-inverse [OF ep-pair-prod])
apply (simp add: prod-map-map cfcomp1)
done

```

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)
by (simp add: coerce-u)

```

```

lemma coerce-sinl [simp]: coerce·(sinl·x) = sinl·(coerce·x)
by (simp add: coerce-ssum ssum-map-sinl)

```

```

lemma coerce-sinr [simp]: coerce·(sinr·x) = sinr·(coerce·x)
by (simp add: coerce-ssum ssum-map-sinr')

lemma coerce-spair [simp]: coerce·(:x, y) = (:coerce·x, coerce·y)
by (simp add: coerce-sprod sprod-map-spair')

lemma coerce-Pair [simp]: coerce·(x, y) = (coerce·x, coerce·y)
by (simp add: coerce-prod)

lemma beta-coerce-cfun [simp]: coerce·f·x = coerce·(f·(coerce·x))
by (simp add: coerce-cfun')

lemma coerce-cfun: coerce·f = coerce oo f oo coerce
by (simp add: cfun-eqI)

lemma coerce-cfun-app [coerce-simp]:
  coerce·f = (λ x. coerce·(f·(coerce·x)))
by (simp add: cfun-eqI)

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 $f :: 'a \rightarrow 'a$ with a deflation $t :: \text{udom} \text{ defl}$: $\text{isodefl } f \ t = (\text{cast} \cdot t = \text{EMB}('a) \text{ oo } f \text{ oo } \text{PRJ}('a))$.

```

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*]:
 $\bigwedge g \ (xs::\text{udom} \cdot 'a::\text{tycon}).$
fmapU·*f*·(*fmapU*·*g*·*xs*) = *fmapU*·($\Lambda \ x. \ f \cdot (g \cdot x)$)·*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*
by (*simp add: fmap-def eta-cfun*)

lemma *fmap-eq-fmapU*: *fmap* = *fmapU*
by (*simp only: fmapU-eq-fmap*)

lemma *cast-TC*:
 $\text{cast} \cdot (\text{TC}('f) \cdot t) = \text{emb} \circ \text{fmapU} \cdot (\text{cast} \cdot t) \circ \text{PRJ}(\text{udom} \cdot 'f::\text{prefunctor})$
by (*rule isodefl-fmapU [unfolded isodefl-def]*)

lemma *isodefl-cast*: *isodefl* (*cast*·*t*) *t*
by (*simp add: isodefl-def*)

lemma *cast-cast-below1*: $A \sqsubseteq B \implies \text{cast} \cdot A \cdot (\text{cast} \cdot B \cdot x) = \text{cast} \cdot A \cdot x$
by (*intro deflation-below-comp1 deflation-cast monofun-cfun-arg*)

lemma *cast-cast-below2*: $A \sqsubseteq B \implies \text{cast} \cdot B \cdot (\text{cast} \cdot A \cdot x) = \text{cast} \cdot A \cdot x$
by (*intro deflation-below-comp2 deflation-cast monofun-cfun-arg*)

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

proof –

have *deflation-d*: *deflation d*
using *assms* **by** (*rule isodefl-imp-deflation*)
have *cast-t*: *cast*·*t* = *emb* oo *d* oo *prj*
using *assms* **unfolding** *isodefl-def* .
have *t-below*: $t \sqsubseteq \text{DEFL}('a)$
apply (*rule cast-below-imp-below*)
apply (*simp only: cast-t cast-DEFL*)
apply (*simp add: cfun-below-iff deflation.below [OF deflation-d]*)
done

```

have fmap-eq: fmap.d = PRJ('a.'f) oo cast.(TC('f).t) oo emb
  by (simp add: fmap-def coerce-cfun cast-TC cast-t prj-emb cfcomp1)
show ?thesis
  apply (simp add: fmap-eq isodefl-def cfun-eq-iff emb-prj)
  apply (simp add: DEFL-app)
  apply (simp add: cast-cast-below1 monofun-cfun t-below)
  apply (simp add: cast-cast-below2 monofun-cfun t-below)
  done
qed

```

```

lemma fmap-ID [simp]: fmap.ID = ID
apply (rule isodefl-DEFL-imp-ID)
apply (subst DEFL-app)
apply (rule isodefl-fmap)
apply (rule isodefl-ID-DEFL)
done

```

```

lemma fmap-ident [simp]: fmap.( $\Lambda$  x. x) = ID
by (simp add: ID-def [symmetric])

```

```

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.(cast.DEFL('a)).xs
by (simp add: coerce-def emb-prj DEFL-app cast-TC)

```

```

lemma fmap-fmap:
  fixes xs :: 'a.'f::functor and g :: 'a  $\rightarrow$  'b and f :: 'b  $\rightarrow$  'c
  shows fmap.f.(fmap.g.xs) = fmap.( $\Lambda$  x. f.(g.x)).xs
unfolding fmap-def
by (simp add: coerce-simp)

```

```

lemma fmap-cfcomp: fmap.(f oo g) = fmap.f oo fmap.g
by (simp add: cfcomp1 fmap-fmap eta-cfun)

```

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.d)
  apply (rule deflation.intro)
  apply (simp add: fmap-fmap deflation.idem eta-cfun)
  apply (subgoal-tac fmap.d.x  $\sqsubseteq$  fmap.ID.x, simp)
  apply (rule monofun-cfun-fun, rule monofun-cfun-arg)
  apply (erule deflation.below-ID)
  done

```

```

lemma ep-pair-fmap:
  ep-pair e p  $\implies$  ep-pair (fmap·e) (fmap·p)
apply (rule ep-pair.intro)
apply (simp add: fmap-fmap ep-pair.e-inverse)
apply (simp add: fmap-fmap)
apply (rule-tac y=fmap.ID·y in below-trans)
apply (rule monofun-cfun-fun)
apply (rule monofun-cfun-arg)
apply (rule cfun-belowI, simp)
apply (erule ep-pair.e-p-below)
apply simp
done

lemma fmap-strict:
  fixes f :: 'a  $\rightarrow$  'b
  assumes f· $\perp$  =  $\perp$  shows fmap·f· $\perp$  = ( $\perp$ ::'b.'f::functor)
proof (rule bottomI)
  have fmap·f·( $\perp$ ::'a.'f)  $\sqsubseteq$  fmap·f·(fmap· $\perp$ ·( $\perp$ ::'b.'f))
    by (simp add: monofun-cfun)
  also have ... = fmap·( $\Lambda$  x. f·( $\perp$ ·x))·( $\perp$ ::'b.'f)
    by (simp add: fmap-fmap)
  also have ...  $\sqsubseteq$  fmap.ID· $\perp$ 
    by (simp add: monofun-cfun assms del: fmap-ID)
  also have ... =  $\perp$ 
    by simp
  finally show fmap·f· $\perp$   $\sqsubseteq$  ( $\perp$ ::'b.'f::functor) .
qed

```

3.4 Proving that *fmap*·*coerce* = *coerce*

```

lemma fmapU-cast-eq:
  fmapU·(cast·A) =
    PRJ(udom·'f) oo cast·(TC('f::functor)·A) oo emb
by (subst cast-TC, rule cfun-eqI, simp)

lemma fmapU-cast-DEFL:
  fmapU·(cast·DEFL('a)) =
    PRJ(udom·'f) oo cast·DEFL('a.'f::functor) oo emb
by (simp add: fmapU-cast-eq DEFL-app)

lemma coerce-functor: COERCE('a.'f, 'b.'f::functor) = fmap·coerce
apply (rule cfun-eqI, rename-tac xs)
apply (simp add: fmap-def coerce-cfun)
apply (simp add: coerce-def)
apply (simp add: cfcomp1)
apply (simp only: emb-prj)
apply (subst fmapU-fmapU [symmetric])
apply (simp add: fmapU-cast-DEFL)
apply (simp add: emb-prj)

```

```

apply (simp add: cast-cast-below1 cast-cast-below2)
done

```

3.5 Lemmas for reasoning about coercion

```

lemma fmapU-cast-coerce [coerce-simp]:
  fixes m :: 'a.'f::functor
  shows fmapU·(cast·DEFL('a))·(COERCE('a.'f, udom.'f)·m) =
    COERCE('a.'f, udom.'f)·m
by (simp add: coerce-functor cast-DEFL fmapU-eq-fmap fmap-fmap eta-cfun)

```

```

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
by (simp add: coerce-functor fmap-fmap)

```

```

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
by (simp add: coerce-functor fmap-fmap)

```

3.6 Configuration of Domain package

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

```

setup <Domain-Take-Proofs.add-rec-type (@{type-name app}, [true, false])>

```

```

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-file <tycondef.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:
     $\bigwedge f\ xs. \text{fmapU}\cdot f\cdot xs = \text{bindU}\cdot xs\cdot (\lambda x. \text{returnU}\cdot (f\cdot x))$ 
  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
  fix f g :: udom -> udom and xs :: udom.'a
  show fmapU·f·(fmapU·g·xs) = fmapU·( $\lambda x. f\cdot(g\cdot x)$ )·xs
  by (simp add: fmapU-eq-bindU bindU-bindU bindU-returnU)
qed
```

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  $\langle \gg \rangle$  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*]:
 $fmapU \cdot f \cdot (returnU \cdot x) = returnU \cdot (f \cdot x)$
by (*simp add: fmapU-eq-bindU bindU-returnU*)

lemma *fmapU-bindU* [*coerce-simp*]:
 $fmapU \cdot f \cdot (bindU \cdot m \cdot k) = bindU \cdot m \cdot (\Lambda x. fmapU \cdot f \cdot (k \cdot x))$
by (*simp add: fmapU-eq-bindU bindU-bindU*)

lemma *bindU-fmapU*:
 $bindU \cdot (fmapU \cdot f \cdot xs) \cdot k = bindU \cdot xs \cdot (\Lambda x. k \cdot (f \cdot x))$
by (*simp add: fmapU-eq-bindU bindU-returnU bindU-bindU*)

4.3 Polymorphic versions of class assumptions

lemma *monad-fmap*:
fixes $xs :: 'a \cdot 'm :: monad$ **and** $f :: 'a \rightarrow 'b$
shows $fmap \cdot f \cdot xs = xs \gg= (\Lambda x. return \cdot (f \cdot x))$
unfolding *bind-def return-def fmap-def*
by (*simp add: coerce-simp fmapU-eq-bindU bindU-returnU*)

lemma *monad-left-unit* [*simp*]: $(return \cdot x \gg= f) = (f \cdot x)$
unfolding *bind-def return-def*
by (*simp add: coerce-simp bindU-returnU*)

lemma *bind-bind*:
fixes $m :: 'a \cdot 'm :: monad$
shows $((m \gg= f) \gg= g) = (m \gg= (\Lambda x. f \cdot x \gg= g))$
unfolding *bind-def*
by (*simp add: coerce-simp bindU-bindU*)

4.4 Derived rules

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

lemma *monad-right-unit* [*simp*]: $(m \gg= return) = m$
apply (*subgoal-tac fmap.ID \cdot m = m*)
apply (*simp only: monad-fmap*)
apply (*simp add: eta-cfun*)
apply *simp*
done

lemma *fmap-return*: $fmap \cdot f \cdot (return \cdot x) = return \cdot (f \cdot x)$
by (*simp add: monad-fmap*)

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

by (*simp add: monad-fmap bind-bind*)

lemma *bind-fmap*: $\text{bind} \cdot (\text{fmap} \cdot f \cdot xs) \cdot k = \text{bind} \cdot xs \cdot (\lambda x. k \cdot (f \cdot x))$
by (*simp add: monad-fmap bind-bind*)

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

lemma *bind-strict*:
assumes $k \cdot \perp = \perp$ **shows** $\perp \gg k = \perp$
proof –
have $\perp \gg k \sqsubseteq \text{return} \cdot \perp \gg k$
by (*intro monofun-cfun below-refl minimal*)
thus $\perp \gg k = \perp$
by (*simp add: assms*)
qed

lemma *congruent-bind*:
 $(\forall m. m \gg k1 = m \gg k2) = (k1 = k2)$
apply (*safe, rule cfun-eqI*)
apply (*drule-tac x=return.x in spec, simp*)
done

4.5 Laws for join

definition *join* :: $('a \cdot 'm) \cdot 'm \rightarrow 'a \cdot 'm :: \text{monad}$
where $\text{join} \equiv \lambda m. m \gg (\lambda x. x)$

lemma *join-fmap-fmap*: $\text{join} \cdot (\text{fmap} \cdot (\text{fmap} \cdot f) \cdot xss) = \text{fmap} \cdot f \cdot (\text{join} \cdot xss)$
by (*simp add: join-def monad-fmap bind-bind*)

lemma *join-return*: $\text{join} \cdot (\text{return} \cdot xs) = xs$
by (*simp add: join-def*)

lemma *join-fmap-return*: $\text{join} \cdot (\text{fmap} \cdot \text{return} \cdot xs) = xs$
by (*simp add: join-def monad-fmap eta-cfun bind-bind*)

lemma *join-fmap-join*: $\text{join} \cdot (\text{fmap} \cdot \text{join} \cdot xsss) = \text{join} \cdot (\text{join} \cdot xsss)$
by (*simp add: join-def monad-fmap bind-bind*)

lemma *bind-def2*: $m \gg k = \text{join} \cdot (\text{fmap} \cdot k \cdot m)$
by (*simp add: join-def monad-fmap eta-cfun bind-bind*)

4.6 Equivalence of monad laws and fmap/join laws

lemma $(\text{return} \cdot x \gg f) = (f \cdot x)$
by (*simp only: bind-def2 fmap-return join-return*)

lemma $(m \gg \text{return}) = m$
by (*simp only: bind-def2 join-fmap-return*)

```

lemma (( $m \ggg f$ )  $\ggg g$ ) = ( $m \ggg (\Lambda x. f \cdot x \ggg g)$ )
apply (simp only: bind-def2)
apply (subgoal-tac join.(fmap.g.(join.(fmap.f.m))) =
  join.(fmap.join.(fmap.(fmap.g).(fmap.f.m))))
apply (simp add: fmap-fmap)
apply (simp add: join-fmap-join join-fmap-fmap)
done

```

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).(return.x) = return.(COERCE('a,'b).x)
by (simp add: coerce-functor fmap-return)

lemma coerce-bind [coerce-simp]:
  fixes m :: 'a.'m::monad and k :: 'a → 'b.'m
  shows COERCE('b.'m, 'c.'m).(m  $\ggg$  k) = m  $\ggg$  ( $\Lambda x. COERCE('b.'m, 'c.'m).(k.x)$ )
by (simp add: coerce-functor fmap-bind)

lemma bind-coerce [coerce-simp]:
  fixes m :: 'a.'m::monad and k :: 'b → 'c.'m
  shows COERCE('a.'m, 'b.'m).m  $\ggg$  k = m  $\ggg$  ( $\Lambda x. k.(COERCE('a,'b).x)$ )
by (simp add: coerce-functor bind-fmap)

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
  fix f show fmapU.f.zeroU = (zeroU :: udom.'a)

```

unfolding *fmapU-eq-bindU*
by (*rule bindU-zeroU*)
qed

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)
unfolding *fmap-def fzero-def*
by (*simp add: coerce-simp*)

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*
unfolding *bind-def* **by** *simp*

lemma *bind-mzero*:
bind.(*fzero* :: 'a.'m::monad-zero).*k* = (*mzero* :: 'b.'m)
unfolding *bind-def mzero-def*
by (*simp add: coerce-simp bindU-zeroU*)

end

6 Monad-Plus Class

theory *Monad-Plus*
imports *Monad*
begin

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

class *plusU* = *tycon* +
fixes *plusU* :: *udom*.'a \rightarrow *udom*.'a \rightarrow *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'*:

$$plusU \cdot (plusU \cdot a \cdot b) \cdot c = plusU \cdot a \cdot (plusU \cdot b \cdot c)$$

instance *monad-plus* \subseteq *functor-plus*

by *standard* (*simp-all only*: *fmapU-eq-bindU bindU-plusU plusU-assoc*)

definition *fplus* :: '*a*'*f*::*functor-plus* \rightarrow '*a*'*f* \rightarrow '*a*'*f*

where *fplus* = *coerce*·(*plusU* :: *udom*'*f* \rightarrow -)

lemma *fmap-fplus*:

fixes *f* :: '*a* \rightarrow '*b* **and** *a b* :: '*a*'*f*::*functor-plus*

shows *fmap*·*f*·(*fplus*·*a*·*b*) = *fplus*·(*fmap*·*f*·*a*)·(*fmap*·*f*·*b*)

unfolding *fmap-def fplus-def*

by (*simp add*: *coerce-simp*)

lemma *fplus-assoc*:

fixes *a b c* :: '*a*'*f*::*functor-plus*

shows *fplus*·(*fplus*·*a*·*b*)·*c* = *fplus*·*a*·(*fplus*·*b*·*c*)

unfolding *fplus-def*

by (*simp add*: *coerce-simp plusU-assoc*)

abbreviation *mplus* :: '*a*'*m*::*monad-plus* \rightarrow '*a*'*m* \rightarrow '*a*'*m*

where *mplus* \equiv *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*)

unfolding *bind-def mplus-def*

by (*simp add*: *coerce-simp bindU-plusU*)

lemma *join-mplus*:

fixes *xss yss* :: ('*a*'*m*)'·*m*::*monad-plus*

shows *join*·(*mplus*·*xss*·*yss*) = *mplus*·(*join*·*xss*)·(*join*·*yss*)

by (*simp add*: *join-def bind-mplus*)

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
unfolding fplus-def fzero-def
by (simp add: coerce-simp plusU-zeroU-left)

lemma fplus-fzero-right:
  fixes m :: 'a.'f::functor-zero-plus
  shows fplus.m.fzero = m
unfolding fplus-def fzero-def
by (simp add: coerce-simp plusU-zeroU-right)

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)
apply (simp add: llist-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-llist)
done

```

```

lemma coerce-LNil [simp]: coerce·LNil = LNil
unfolding LNil-def by simp

```

```

lemma coerce-LCons [simp]: coerce·(LCons·x·xs) = LCons·(coerce·x)·(coerce·xs)
unfolding LCons-def by simp

```

```

lemma fmapU-llist-simps [simp]:
  fmapU·f·(⊥::udom·llist) = ⊥
  fmapU·f·LNil = LNil
  fmapU·f·(LCons·x·xs) = LCons·(f·x)·(fmapU·f·xs)
unfolding fmapU-llist-def llist-map-def
apply (subst fix-eq, simp)
apply (subst fix-eq, simp add: LNil-def)
apply (subst fix-eq, simp add: LCons-def)
done

```

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
  fix f g and xs :: udom·llist
  show fmapU·f·(fmapU·g·xs) = fmapU·(λ x. f·(g·x))·xs
    by (induct xs rule: llist.induct) simp-all
qed

```

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*)
by *fixrec-simp*

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*)
by *fixrec-simp*

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

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

lemma *plusU-LNil-right*: *plusU*·*xs*·*LNil* = *xs*
by (*induct xs* rule: *llist.induct*) *simp-all*

lemma *plusU-llist-assoc*:
fixes *xs ys zs* :: *udom·llist*
shows *plusU*·(*plusU*·*xs*·*ys*)·*zs* = *plusU*·*xs*·(*plusU*·*ys*·*zs*)
by (*induct xs* rule: *llist.induct*) *simp-all*

lemma *bindU-plusU-llist*:
fixes *xs ys* :: *udom·llist* **shows**
bindU·(*plusU*·*xs*·*ys*)·*f* = *plusU*·(*bindU*·*xs*·*f*)·(*bindU*·*ys*·*f*)
by (*induct xs* rule: *llist.induct*) (*simp-all* add: *plusU-llist-assoc*)

instance proof
fix *x* :: *udom*
fix *f* :: *udom* → *udom*
fix *h k* :: *udom* → *udom·llist*
fix *xs ys zs* :: *udom·llist*
show *fmapU*·*f*·*xs* = *bindU*·*xs*·(λ *x*. *returnU*·(*f*·*x*))
by (*induct xs* rule: *llist.induct*, *simp-all* add: *returnU-llist-def*)

```

show bindU.(returnU.x).k = k.x
  by (simp add: returnU-llist-def plusU-LNil-right)
show bindU.(bindU.xs.h).k = bindU.xs.( $\Lambda$  x. bindU.(h.x).k)
  by (induct xs rule: llist.induct)
    (simp-all add: bindU-plusU-llist)
show bindU.(plusU.xs.ys).k = plusU.(bindU.xs.k).(bindU.ys.k)
  by (induct xs rule: llist.induct)
    (simp-all add: plusU-llist-assoc)
show plusU.(plusU.xs.ys).zs = plusU.xs.(plusU.ys.zs)
  by (rule plusU-llist-assoc)
show bindU.zeroU.k = zeroU
  by (simp add: zeroU-llist-def)
show fmapU.f.(plusU.xs.ys) = plusU.(fmapU.f.xs).(fmapU.f.ys)
  by (induct xs rule: llist.induct) simp-all
show fmapU.f.zeroU = (zeroU :: udom.llist)
  by (simp add: zeroU-llist-def)
show plusU.zeroU.xs = xs
  by (simp add: zeroU-llist-def)
show plusU.xs.zeroU = xs
  by (simp add: zeroU-llist-def plusU-LNil-right)
qed

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.f.( $\perp :: 'a$ .llist) =  $\perp$ 
  fmap.f.LNil = LNil
  fmap.f.(LCons.x.xs) = LCons.(f.x).(fmap.f.xs)
unfolding fmap-def by simp-all

```

```

lemma mplus-llist-simps [simp]:
  mplus.( $\perp :: 'a$ .llist).ys =  $\perp$ 
  mplus.LNil.ys = ys
  mplus.(LCons.x.xs).ys = LCons.x.(mplus.xs.ys)
unfolding mplus-def by simp-all

```

```

lemma bind-llist-simps [simp]:
  bind.( $\perp :: 'a$ .llist).f =  $\perp$ 
  bind.LNil.f = LNil

```



```

    bind.(LCons.x.xs).f = mplus.(f.x).(bind.xs.f)
unfolding bind-def mplus-def
by (simp-all add: coerce-simp)

```

```

lemma return-llist-def:
  return = (λ x. LCons.x.LNil)
unfolding return-def returnU-llist-def
by (simp add: coerce-simp)

```

```

lemma mzero-llist-def:
  mzero = LNil
unfolding mzero-def zeroU-llist-def
by simp

```

```

lemma join-llist-simps [simp]:
  join.(⊥::'a.llist.llist) = ⊥
  join.LNil = LNil
  join.(LCons.xs.xss) = mplus.xs.(join.xss)
unfolding join-def by simp-all

```

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)
apply (simp add: maybe-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-maybe)
done

```

```

lemma coerce-Nothing [simp]: coerce.Nothing = Nothing
unfolding Nothing-def by simp

```

```

lemma coerce-Just [simp]: coerce.(Just.x) = Just.(coerce.x)
unfolding Just-def by simp

```

```

lemma fmapU-maybe-simps [simp]:
  fmapU.f.(⊥::udom.maybe) = ⊥
  fmapU.f.Nothing = Nothing
  fmapU.f.(Just.x) = Just.(f.x)
unfolding fmapU-maybe-def maybe-map-def fix-const
apply simp

```

```

apply (simp add: Nothing-def)
apply (simp add: Just-def)
done

```

9.2 Class instance proofs

```

instance maybe :: functor
apply standard
apply (induct-tac xs rule: maybe.induct, simp-all)
done

```

```

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)
by fixrec-simp

```

```

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)
by fixrec-simp

```

```

definition zeroU-maybe-def:
  zeroU = Nothing

```

```

definition returnU-maybe-def:
  returnU = Just

```

```

lemma plusU-Nothing-right: plusU.xs.Nothing = xs
by (induct xs rule: maybe.induct) simp-all

```

```

lemma bindU-plusU-maybe:
  fixes xs ys :: udom-maybe shows
    bindU.(plusU.xs.ys)·f = plusU.(bindU.xs.f)·(bindU.ys.f)
apply (induct xs rule: maybe.induct)
apply simp-all
oops

```

```

instance proof
  fix x :: udom
  fix f :: udom → udom
  fix h k :: udom → udom-maybe
  fix xs ys zs :: udom-maybe

```

```

show fmapU.f.xs = bindU.xs.( $\Lambda$  x. returnU.(f.x))
  by (induct xs rule: maybe.induct, simp-all add: returnU-maybe-def)
show bindU.(returnU.x).k = k.x
  by (simp add: returnU-maybe-def plusU-Nothing-right)
show bindU.(bindU.xs.h).k = bindU.xs.( $\Lambda$  x. bindU.(h.x).k)
  by (induct xs rule: maybe.induct) simp-all
show plusU.(plusU.xs.ys).zs = plusU.xs.(plusU.ys.zs)
  by (induct xs rule: maybe.induct) simp-all
show bindU.zeroU.k = zeroU
  by (simp add: zeroU-maybe-def)
show fmapU.f.(plusU.xs.ys) = plusU.(fmapU.f.xs).(fmapU.f.ys)
  by (induct xs rule: maybe.induct) simp-all
show fmapU.f.zeroU = (zeroU :: udom.maybe)
  by (simp add: zeroU-maybe-def)
show plusU.zeroU.xs = xs
  by (simp add: zeroU-maybe-def)
show plusU.xs.zeroU = xs
  by (simp add: zeroU-maybe-def plusU-Nothing-right)
qed

end

```

9.3 Transfer properties to polymorphic versions

```

lemma fmap-maybe-simps [simp]:
  fmap.f.( $\perp$ ::'a.maybe) =  $\perp$ 
  fmap.f.Nothing = Nothing
  fmap.f.(Just.x) = Just.(f.x)
unfolding fmap-def by simp-all

```

```

lemma fplus-maybe-simps [simp]:
  fplus.( $\perp$ ::'a.maybe).ys =  $\perp$ 
  fplus.Nothing.ys = ys
  fplus.(Just.x).ys = Just.x
unfolding fplus-def by simp-all

```

```

lemma fplus-Nothing-right [simp]:
  fplus.m.Nothing = m
by (simp add: fplus-def plusU-Nothing-right)

```

```

lemma bind-maybe-simps [simp]:
  bind.( $\perp$ ::'a.maybe).f =  $\perp$ 
  bind.Nothing.f = Nothing
  bind.(Just.x).f = f.x
unfolding bind-def fplus-def by simp-all

```

```

lemma return-maybe-def: return = Just
unfolding return-def returnU-maybe-def
by (simp add: coerce-cfun ccomp1 eta-cfun)

```

```

lemma mzero-maybe-def: mzero = Nothing
unfolding mzero-def zeroU-maybe-def
by simp

```

```

lemma join-maybe-simps [simp]:
  join.( $\perp :: 'a$ .maybe.maybe) =  $\perp$ 
  join.Nothing = Nothing
  join.(Just.xs) = xs
unfolding join-def by simp-all

```

9.4 Maybe is not in monad-plus

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

```

lemma maybe-counterexample1:
   $\llbracket a = \text{Just}\cdot x; b = \perp; k\cdot x = \text{Nothing} \rrbracket$ 
   $\implies \text{fplus}\cdot a\cdot b \ggg k \neq \text{fplus}\cdot (a \ggg k)\cdot (b \ggg k)$ 
by simp

```

```

lemma maybe-counterexample2:
   $\llbracket a = \text{Just}\cdot x; b = \text{Just}\cdot y; k\cdot x = \text{Nothing}; k\cdot y = \text{Just}\cdot z \rrbracket$ 
   $\implies \text{fplus}\cdot a\cdot b \ggg k \neq \text{fplus}\cdot (a \ggg k)\cdot (b \ggg k)$ 
by simp

```

end

10 Error monad

```

theory Error-Monad
imports Monad-Plus
begin

```

10.1 Type definition

```

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

```

```

lemma coerce-error-abs [simp]: coerce.(error-abs.x) = error-abs.(coerce.x)
apply (simp add: error-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-error)
done

```

```

lemma coerce-Err [simp]: coerce.(Err.x) = Err.(coerce.x)
unfolding Err-def by simp

```

```

lemma coerce-Ok [simp]: coerce.(Ok.m) = Ok.(coerce.m)
unfolding Ok-def by simp

```

```

lemma fmapU-error-simps [simp]:

```

```

  fmapU.f.(⊥::udom.'a error) = ⊥
  fmapU.f.(Err.e) = Err.e
  fmapU.f.(Ok.x) = Ok.(f.x)
unfolding fmapU-error-def error-map-def fix-const
apply simp
apply (simp add: Err-def)
apply (simp add: Ok-def)
done

```

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)
by fixrec-simp

```

```

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) = ⊥
by fixrec-simp

```

instance proof

```

  fix f g :: udom → udom and r :: udom.'a error
  show fmapU.f.(fmapU.g.r) = fmapU.(λ x. f.(g.x)).r
  by (induct r rule: error.induct) simp-all
next
  fix f :: udom → udom and r :: udom.'a error
  show fmapU.f.r = bindU.r.(λ x. returnU.(f.x))
  by (induct r rule: error.induct)
  (simp-all add: returnU-error-def)
next
  fix f g :: udom → udom.'a error and x :: udom
  show bindU.(returnU.x).f = f.x
  by (simp add: returnU-error-def)
next
  fix r :: udom.'a error and f g :: udom → udom.'a error
  show bindU.(bindU.r.f).g = bindU.r.(λ x. bindU.(f.x).g)
  by (induct r rule: error.induct)
  simp-all

```

```

next
  fix f :: udom → udom and a b :: udom.'a error
  show fmapU.f.(plusU.a.b) = plusU.(fmapU.f.a).(fmapU.f.b)
    by (induct a rule: error.induct) simp-all
next
  fix a b c :: udom.'a error
  show plusU.(plusU.a.b).c = plusU.a.(plusU.b.c)
    by (induct a rule: error.induct) simp-all
qed

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)
unfolding fmap-def [where 'f='e error]
by (simp-all add: coerce-simp)

```

```

lemma return-error-def: return = Ok
unfolding return-def returnU-error-def
by (simp add: coerce-simp eta-cfun)

```

```

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
unfolding bind-def
by (simp-all add: coerce-simp)

```

```

lemma join-error-simps [simp]:
  join.⊥ = (⊥ :: 'a.'e error)
  join.(Err.e) = Err.e
  join.(Ok.x) = x
unfolding join-def by simp-all

```

```

lemma fplus-error-simps [simp]:
  fplus.⊥.r = (⊥ :: 'a.'e error)
  fplus.(Err.e).r = r
  fplus.(Ok.x).r = Ok.x
unfolding fplus-def
by (simp-all add: coerce-simp)

```

```

end

```

11 Writer monad

```
theory Writer-Monad
imports Monad
begin
```

11.1 Monoid class

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

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)
apply (simp add: writer-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-writer)
done
```

```
lemma coerce-Writer [simp]:
  coerce.(Writer.w.x) = Writer.(coerce.w).(coerce.x)
unfolding Writer-def by simp
```

```
lemma fmapU-writer-simps [simp]:
  fmapU.f.( $\perp :: \text{udom}.'w\ writer$ ) =  $\perp$ 
  fmapU.f.(Writer.w.x) = Writer.w.(f.x)
unfolding fmapU-writer-def writer-map-def fix-const
apply simp
apply (simp add: Writer-def)
done
```

11.3 Class instance proofs

instance *writer* :: (domain) functor

proof

fix $f\ g :: \text{udom} \rightarrow \text{udom}$ **and** $xs :: \text{udom}.'a\ \text{writer}$
 show $\text{fmap}\ U.f \cdot (\text{fmap}\ U.g \cdot xs) = \text{fmap}\ U.(\lambda\ x.\ f \cdot (g \cdot x)) \cdot xs$
 by (induct *xs* rule: *writer.induct*) simp-all

qed

instantiation *writer* :: (monoid) monad

begin

fixrec *bindU-writer* ::

$\text{udom}.'a\ \text{writer} \rightarrow (\text{udom} \rightarrow \text{udom}.'a\ \text{writer}) \rightarrow \text{udom}.'a\ \text{writer}$

where *bindU-writer* · (*Writer* · *w* · *x*) · *f* =
 (case *f* · *x* of *Writer* · *w'* · *y* \Rightarrow *Writer* · (*mappend* · *w* · *w'*) · *y*)

lemma *bindU-writer-strict* [*simp*]: $\text{bind}\ U.\perp \cdot k = (\perp :: \text{udom}.'a\ \text{writer})$

by *fixrec-simp*

definition

returnU = *Writer* · *mempty*

instance proof

fix $f :: \text{udom} \rightarrow \text{udom}$ **and** $m :: \text{udom}.'a\ \text{writer}$
 show $\text{fmap}\ U.f \cdot m = \text{bind}\ U.m \cdot (\lambda\ x.\ \text{return}\ U.(f \cdot x))$
 by (induct *m* rule: *writer.induct*)
 (simp-all add: *returnU-writer-def* *mempty-right*)

next

fix $f :: \text{udom} \rightarrow \text{udom}.'a\ \text{writer}$ **and** $x :: \text{udom}$
 show $\text{bind}\ U.(\text{return}\ U.x) \cdot f = f \cdot x$
 by (cases *f* · *x* rule: *writer.exhaust*)
 (simp-all add: *returnU-writer-def* *mempty-left*)

next

fix $m :: \text{udom}.'a\ \text{writer}$ **and** $f\ g :: \text{udom} \rightarrow \text{udom}.'a\ \text{writer}$
 show $\text{bind}\ U.(\text{bind}\ U.m \cdot f) \cdot g = \text{bind}\ U.m \cdot (\lambda\ x.\ \text{bind}\ U.(f \cdot x) \cdot g)$
 apply (induct *m* rule: *writer.induct*, *simp*)
 apply (case-tac *f* · *a* rule: *writer.exhaust*, *simp*)
 apply (case-tac *g* · *aa* rule: *writer.exhaust*, *simp*)
 apply (simp add: *mappend-assoc*)
 done

qed

end

11.4 Transfer properties to polymorphic versions

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

$\text{fmap}\ f \cdot (\perp :: 'a.'a\ \text{writer}) = \perp$
 $\text{fmap}\ f \cdot (\text{Writer} \cdot w \cdot x :: 'a.'a\ \text{writer}) = \text{Writer} \cdot w \cdot (f \cdot x)$

unfolding *fmap-def* [**where** $'f = 'w \text{ writer}$]
by (*simp-all add: coerce-simp*)

lemma *return-writer-def*: $\text{return} = \text{Writer} \cdot \text{mempty}$
unfolding *return-def returnU-writer-def*
by (*simp add: coerce-simp eta-cfun*)

lemma *bind-writer-simps* [*simp*]:
 $\text{bind} \cdot (\perp :: 'a \cdot 'w :: \text{monoid writer}) \cdot f = \perp$
 $\text{bind} \cdot (\text{Writer} \cdot w \cdot x :: 'a \cdot 'w :: \text{monoid writer}) \cdot k =$
 $(\text{case } k \cdot x \text{ of } \text{Writer} \cdot w' \cdot y \Rightarrow \text{Writer} \cdot (\text{mappend} \cdot w \cdot w') \cdot y)$
unfolding *bind-def*
apply (*simp add: coerce-simp*)
apply (*cases k x rule: writer.exhaust*)
apply (*simp-all add: coerce-simp*)
done

lemma *join-writer-simps* [*simp*]:
 $\text{join} \cdot \perp = (\perp :: 'a \cdot 'w :: \text{monoid writer})$
 $\text{join} \cdot (\text{Writer} \cdot w \cdot (\text{Writer} \cdot w' \cdot x)) = \text{Writer} \cdot (\text{mappend} \cdot w \cdot w') \cdot x$
unfolding *join-def* **by** *simp-all*

11.5 Extra operations

definition *tell* :: $'w \rightarrow \text{unit} \cdot ('w :: \text{monoid writer})$
where $\text{tell} = (\Lambda w. \text{Writer} \cdot w \cdot ())$

end

12 Binary tree monad

theory *Binary-Tree-Monad*
imports *Monad*
begin

12.1 Type definition

tycondef $'a \cdot \text{btree} =$
 $\text{Leaf} \ (\text{lazy } 'a) \mid \text{Node} \ (\text{lazy } 'a \cdot \text{btree}) \ (\text{lazy } 'a \cdot \text{btree})$

lemma *coerce-btree-abs* [*simp*]: $\text{coerce} \cdot (\text{btree-abs} \cdot x) = \text{btree-abs} \cdot (\text{coerce} \cdot x)$
apply (*simp add: btree-abs-def coerce-def*)
apply (*simp add: emb-prj-emb prj-emb-prj DEFL-eq-btree*)
done

lemma *coerce-Leaf* [*simp*]: $\text{coerce} \cdot (\text{Leaf} \cdot x) = \text{Leaf} \cdot (\text{coerce} \cdot x)$
unfolding *Leaf-def* **by** *simp*

lemma *coerce-Node* [*simp*]: $\text{coerce} \cdot (\text{Node} \cdot xs \cdot ys) = \text{Node} \cdot (\text{coerce} \cdot xs) \cdot (\text{coerce} \cdot ys)$

unfolding *Node-def* **by** *simp*

lemma *fmapU-btree-simps* [*simp*]:
 $fmapU \cdot f \cdot (\perp :: udom \cdot btree) = \perp$
 $fmapU \cdot f \cdot (Leaf \cdot x) = Leaf \cdot (f \cdot x)$
 $fmapU \cdot f \cdot (Node \cdot xs \cdot ys) = Node \cdot (fmapU \cdot f \cdot xs) \cdot (fmapU \cdot f \cdot ys)$
unfolding *fmapU-btree-def* *btree-map-def*
apply (*subst fix-eq*, *simp*)
apply (*subst fix-eq*, *simp* *add*: *Leaf-def*)
apply (*subst fix-eq*, *simp* *add*: *Node-def*)
done

12.2 Class instance proofs

instance *btree* :: *functor*
apply *standard*
apply (*induct-tac xs* *rule*: *btree.induct*, *simp-all*)
done

instantiation *btree* :: *monad*
begin

definition
 $returnU = Leaf$

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

lemma *bindU-btree-strict* [*simp*]: $bindU \cdot \perp \cdot k = (\perp :: udom \cdot btree)$
by *fixrec-simp*

instance proof
fix *x* :: *udom*
fix *f* :: *udom* → *udom*
fix *h* *k* :: *udom* → *udom* · *btree*
fix *xs* :: *udom* · *btree*
show $fmapU \cdot f \cdot xs = bindU \cdot xs \cdot (\lambda x. returnU \cdot (f \cdot x))$
by (*induct xs* *rule*: *btree.induct*, *simp-all* *add*: *returnU-btree-def*)
show $bindU \cdot (returnU \cdot x) \cdot k = k \cdot x$
by (*simp* *add*: *returnU-btree-def*)
show $bindU \cdot (bindU \cdot xs \cdot h) \cdot k = bindU \cdot xs \cdot (\lambda x. bindU \cdot (h \cdot x) \cdot k)$
by (*induct xs* *rule*: *btree.induct*) *simp-all*
qed

end

12.3 Transfer properties to polymorphic versions

lemma *fmap-btree-simps* [*simp*]:
 fmap·*f*·(\perp ::'a.btree) = \perp
 fmap·*f*·(*Leaf*·*x*) = *Leaf*·(*f*·*x*)
 fmap·*f*·(*Node*·*xs*·*ys*) = *Node*·(*fmap*·*f*·*xs*)·(*fmap*·*f*·*ys*)
unfolding *fmap-def* **by** *simp-all*

lemma *bind-btree-simps* [*simp*]:
 bind·(\perp ::'a.btree)·*k* = \perp
 bind·(*Leaf*·*x*)·*k* = *k*·*x*
 bind·(*Node*·*xs*·*ys*)·*k* = *Node*·(*bind*·*xs*·*k*)·(*bind*·*ys*·*k*)
unfolding *bind-def*
by (*simp-all* *add*: *coerce-simp*)

lemma *return-btree-def*:
 return = *Leaf*
unfolding *return-def* *returnU-btree-def*
by (*simp* *add*: *coerce-simp* *eta-cfun*)

lemma *join-btree-simps* [*simp*]:
 join·(\perp ::'a.btree.btree) = \perp
 join·(*Leaf*·*xs*) = *xs*
 join·(*Node*·*xss*·*yss*) = *Node*·(*join*·*xss*)·(*join*·*yss*)
unfolding *join-def* **by** *simp-all*

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*)
apply (*simp* *add*: *lifted-abs-def* *coerce-def*)
apply (*simp* *add*: *emb-prj-emb* *prj-emb-prj* *DEFL-eq-lifted*)
done

lemma *coerce-Lifted* [*simp*]: *coerce*·(*Lifted*·*x*) = *Lifted*·(*coerce*·*x*)
unfolding *Lifted-def* **by** *simp*

lemma *fmapU-lifted-simps* [*simp*]:
 fmapU·*f*·(\perp ::*udom*·lifted) = \perp
 fmapU·*f*·(*Lifted*·*x*) = *Lifted*·(*f*·*x*)

```

unfolding fmapU-lifted-def lifted-map-def fix-const
apply simp
apply (simp add: Lifted-def)
done

```

13.2 Class instance proofs

```

instance lifted :: functor
  by standard (induct-tac xs rule: lifted.induct, simp-all)

```

```

instantiation lifted :: monad
begin

```

```

fixrec bindU-lifted :: udom·lifted → (udom → udom·lifted) → udom·lifted
  where bindU-lifted·(Lifted·x)·k = k·x

```

```

lemma bindU-lifted-strict [simp]: bindU·⊥·k = (⊥::udom·lifted)
by fixrec-simp

```

```

definition returnU-lifted-def:
  returnU = Lifted

```

```

instance proof
  fix x :: udom
  fix f :: udom → udom
  fix h k :: udom → udom·lifted
  fix xs :: udom·lifted
  show fmapU·f·xs = bindU·xs·(λ x. returnU·(f·x))
    by (induct xs rule: lifted.induct, simp-all add: returnU-lifted-def)
  show bindU·(returnU·x)·k = k·x
    by (simp add: returnU-lifted-def)
  show bindU·(bindU·xs·h)·k = bindU·xs·(λ x. bindU·(h·x)·k)
    by (induct xs rule: lifted.induct) simp-all
qed

```

```

end

```

13.3 Transfer properties to polymorphic versions

```

lemma fmap-lifted-simps [simp]:
  fmap·f·(⊥::'a·lifted) = ⊥
  fmap·f·(Lifted·x) = Lifted·(f·x)
unfolding fmap-def by simp-all

```

```

lemma bind-lifted-simps [simp]:
  bind·(⊥::'a·lifted)·f = ⊥
  bind·(Lifted·x)·f = f·x
unfolding bind-def by simp-all

```

```

lemma return-lifted-def: return = Lifted

```

```

unfolding return-def returnU-lifted-def
by (simp add: coerce-cfun cfcomp1 eta-cfun)

```

```

lemma join-lifted-simps [simp]:
  join.( $\perp :: 'a$ .lifted.lifted) =  $\perp$ 
  join.(Lifted.xs) = xs
unfolding join-def by simp-all

```

```

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)
apply (simp add: resT-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-resT)
done

```

```

lemma coerce-Done [simp]: coerce.(Done.x) = Done.(coerce.x)
unfolding Done-def by simp

```

```

lemma coerce-More [simp]: coerce.(More.m) = More.(coerce.m)
unfolding More-def by simp

```

```

lemma resT-induct [case-names adm bottom Done More]:
  fixes P :: 'a.'f::functor resT  $\Rightarrow$  bool
  assumes adm: adm P
  assumes bottom: P  $\perp$ 
  assumes Done:  $\bigwedge x. P (Done.x)$ 
  assumes More:  $\bigwedge m f. (\bigwedge (r :: 'a.'f \text{ resT}). P (f.r)) \Longrightarrow P (More.(fmap.f.m))$ 
  shows P r
proof (induct r rule: resT.take-induct [OF adm])

```

```

fix n show P (resT-take n·r)
  apply (induct n arbitrary: r)
  apply (simp add: bottom)
  apply (case-tac r rule: resT.exhaust)
  apply (simp add: bottom)
  apply (simp add: Done)
  apply (simp add: More)
  done
qed

```

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)
unfolding fmapU-resT-def resT-map-def
apply (subst fix-eq, simp)
apply (subst fix-eq, simp add: Done-def)
apply (subst fix-eq, simp add: More-def)
done

```

```

instance resT :: (functor) functor
proof
  fix f g :: udom → udom and xs :: udom·'a resT
  show fmapU·f·(fmapU·g·xs) = fmapU·(λ x. f·(g·x))·xs
  by (induct xs rule: resT-induct, simp-all add: fmap-fmap)
qed

```

```

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)
by fixrec-simp

```

```

definition
  returnU = Done

```

```

instance proof
  fix f :: udom → udom and xs :: udom·'a resT
  show fmapU·f·xs = bindU·xs·(λ x. returnU·(f·x))
  by (induct xs rule: resT-induct)
  (simp-all add: fmap-fmap returnU-resT-def)
next
  fix f :: udom → udom·'a resT and x :: udom

```

```

show bindU·(returnU·x)·f = f·x
  by (simp add: returnU-resT-def)
next
  fix xs :: udom·'a resT and h k :: udom → udom·'a resT
  show bindU·(bindU·xs·h)·k = bindU·xs·(λ x. bindU·(h·x)·k)
    by (induct xs rule: resT-induct)
      (simp-all add: fmap-fmap)
qed

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)
unfolding fmap-def [where 'f='f resT]
by (simp-all add: coerce-simp)

```

```

lemma return-resT-def: return = Done
unfolding return-def returnU-resT-def
by (simp add: coerce-simp eta-cfun)

```

```

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)
unfolding bind-def
by (simp-all add: coerce-simp)

```

```

lemma join-resT-simps [simp]:
  join·⊥ = (⊥ :: 'a.'f::functor resT)
  join·(Done·x) = x
  join·(More·m) = More·(fmap·join·m)
unfolding join-def by simp-all

```

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]: $\text{zipRT} \cdot f \cdot \perp \cdot r = \perp$
by *fixrec-simp*

lemma *zipRT-strict2* [simp]: $\text{zipRT} \cdot f \cdot r \cdot \perp = \perp$
by (*fixrec-simp, cases r, simp-all*)

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*: $\text{Done} \cdot f \diamond \text{Done} \cdot x = \text{Done} \cdot (f \cdot x)$
by *simp*

lemma *zipRT-identity*: $\text{Done} \cdot \text{ID} \diamond r = r$
by (*induct r rule: resT-induct, simp-all add: fmap-fmap eta-cfun*)

lemma *zipRT-interchange*: $r \diamond \text{Done} \cdot x = \text{Done} \cdot (\lambda f. f \cdot x) \diamond r$
by (*induct r rule: resT-induct, simp-all add: fmap-fmap*)

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 (*induct r1 arbitrary: r2 r3 rule: resT-induct*)

case (*Done x1*) **thus** ?case

proof (*induct r2 arbitrary: r3 rule: resT-induct*)

case (*Done x2*) **thus** ?case

proof (*induct r3 rule: resT-induct*)

case (*More p3 c3*) **thus** ?case

by (*simp add: fmap-fmap*)

qed *simp-all*

next

case (*More p2 c2*) **thus** ?case

proof (*induct r3 rule: resT-induct*)

case (*Done x3*) **thus** ?case

by (*simp add: fmap-fmap*)

next

case (*More p3 c3*) **thus** ?case

by (*simp add: fmap-fmap fmap-fplus*)


```

      qed simp-all
    qed simp-all
  next
    case (More p1 c1) thus ?case
  proof (induct r2 arbitrary: r3 rule: resT-induct)
    case (Done y) thus ?case
  proof (induct r3 rule: resT-induct)
    case (Done x3) thus ?case
    by (simp add: fmap-fmap)
  next
    case (More p3 c3) thus ?case
    by (simp add: fmap-fmap)
  qed simp-all
  next
    case (More p2 c2) thus ?case
  proof (induct r3 rule: resT-induct)
    case (Done x3) thus ?case
    by (simp add: fmap-fmap fmap-fplus)
  next
    case (More p3 c3) thus ?case
    by (simp add: fmap-fmap fmap-fplus fplus-assoc)
  qed simp-all
  qed simp-all
qed simp-all

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.('f::functor, 's) stateT =
  StateT (runStateT :: 's → ('a × 's)·'f)

```

```

lemma coerce-stateT-abs [simp]: coerce.(stateT-abs·x) = stateT-abs.(coerce·x)
apply (simp add: stateT-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-stateT)
done

```

```

lemma coerce-StateT [simp]: coerce.(StateT·k) = StateT.(coerce·k)
unfolding StateT-def by simp

```

```

lemma stateT-cases [case-names StateT]:
  obtains k where y = StateT·k
proof
  show y = StateT.(runStateT·y)

```

by (cases y, simp-all)
qed

lemma stateT-induct [case-names StateT]:
fixes P :: 'a.('f::functor,'s) stateT \Rightarrow bool
assumes $\bigwedge k. P (StateT.k)$
shows P y
by (cases y rule: stateT-cases, simp add: assms)

lemma stateT-eqI:
($\bigwedge s. runStateT.a.s = runStateT.b.s \implies a = b$)
apply (cases a rule: stateT-cases)
apply (cases b rule: stateT-cases)
apply (simp add: cfun-eq-iff)
done

lemma runStateT-coerce [simp]:
runStateT.(coerce.k).s = coerce.(runStateT.k.s)
by (induct k rule: stateT-induct, simp)

15.1 Functor class instance

lemma fmapU-StateT [simp]:
fmapU.f.(StateT.k) =
StateT.($\Lambda s. fmap.(\Lambda(x, s'). (f.x, s')).(k.s)$)
unfolding fmapU-stateT-def stateT-map-def StateT-def
by (subst fix-eq, simp add: cfun-map-def csplit-def prod-map-def)

lemma runStateT-fmapU [simp]:
runStateT.(fmapU.f.m).s =
fmap.($\Lambda(x, s'). (f.x, s').(runStateT.m.s)$)
by (cases m rule: stateT-cases, simp)

instantiation stateT :: (functor, domain) functor
begin

instance
apply standard
apply (induct-tac xs rule: stateT-induct)
apply (simp-all add: fmap-fmap ID-def csplit-def)
done

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]*:

$bindU \cdot (StateT \cdot f) \cdot k =$

$StateT \cdot (\Lambda s. f \cdot s \gg= (\Lambda (x, s'). runStateT \cdot (k \cdot x) \cdot s'))$

unfolding *bindU-stateT-def* **by** *simp*

lemma *runStateT-bindU [simp]*:

$runStateT \cdot (bindU \cdot m \cdot k) \cdot s = runStateT \cdot m \cdot s \gg= (\Lambda (x, s'). runStateT \cdot (k \cdot x) \cdot s')$

unfolding *bindU-stateT-def* **by** *simp*

instance proof

fix $f :: udom \rightarrow udom$ **and** $r :: udom \cdot ('a, 'b) stateT$

show $fmapU \cdot f \cdot r = bindU \cdot r \cdot (\Lambda x. returnU \cdot (f \cdot x))$

by (*rule stateT-eqI*)

(*simp add: returnU-stateT-def monad-fmap prod-map-def csplit-def*)

next

fix $f :: udom \rightarrow udom \cdot ('a, 'b) stateT$ **and** $x :: udom$

show $bindU \cdot (returnU \cdot x) \cdot f = f \cdot x$

by (*rule stateT-eqI*)

(*simp add: returnU-stateT-def eta-cfun*)

next

fix $r :: udom \cdot ('a, 'b) stateT$ **and** $f g :: udom \rightarrow udom \cdot ('a, 'b) stateT$

show $bindU \cdot (bindU \cdot r \cdot f) \cdot g = bindU \cdot r \cdot (\Lambda x. bindU \cdot (f \cdot x) \cdot g)$

by (*rule stateT-eqI*)

(*simp add: bind-bind csplit-def*)

qed

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]*:

$runStateT \cdot zeroU \cdot s = mzero$

unfolding *zeroU-stateT-def* **by** *simp*

instance proof

fix $k :: udom \rightarrow udom \cdot ('a, 'b) stateT$

show $bindU \cdot zeroU \cdot k = zeroU$

by (*rule stateT-eqI, simp add: bind-mzero*)

qed

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]:

$runStateT \cdot (plusU \cdot a \cdot b) \cdot s =$
 $mplus \cdot (runStateT \cdot a \cdot s) \cdot (runStateT \cdot b \cdot s)$

unfolding $plusU-stateT-def$ **by** $simp$

instance proof

fix $a b :: udom \cdot ('a, 'b) stateT$ **and** $k :: udom \rightarrow udom \cdot ('a, 'b) stateT$
show $bindU \cdot (plusU \cdot a \cdot b) \cdot k = plusU \cdot (bindU \cdot a \cdot k) \cdot (bindU \cdot b \cdot k)$
by ($rule stateT-eqI$, $simp$ add : $bind-mplus$)

next

fix $a b c :: udom \cdot ('a, 'b) stateT$
show $plusU \cdot (plusU \cdot a \cdot b) \cdot c = plusU \cdot a \cdot (plusU \cdot b \cdot c)$
by ($rule stateT-eqI$, $simp$ add : $mplus-assoc$)

qed

end

15.5 Monad zero plus instance

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

fix $m :: udom \cdot ('a, 'b) stateT$
show $plusU \cdot zeroU \cdot m = m$
by ($rule stateT-eqI$, $simp$ add : $mplus-mzero-left$)

next

fix $m :: udom \cdot ('a, 'b) stateT$
show $plusU \cdot m \cdot zeroU = m$
by ($rule stateT-eqI$, $simp$ add : $mplus-mzero-right$)

qed

15.6 Transfer properties to polymorphic versions

lemma $coerce-csplit$ [coerce-simp]:

shows $coerce \cdot (csplit \cdot f \cdot p) = csplit \cdot (\Lambda x y. coerce \cdot (f \cdot x \cdot y)) \cdot p$

unfolding $csplit-def$ **by** $simp$

lemma $csplit-coerce$ [coerce-simp]:

fixes $p :: 'a \times 'b$

shows $csplit \cdot f \cdot (COERCE('a \times 'b, 'c \times 'd) \cdot p) =$
 $csplit \cdot (\Lambda x y. f \cdot (COERCE('a, 'c) \cdot x) \cdot (COERCE('b, 'd) \cdot y)) \cdot p$
unfolding *coerce-prod csplit-def prod-map-def by simp*

lemma *fmap-stateT-simps [simp]*:
 $fmap \cdot f \cdot (StateT \cdot m :: 'a \cdot ('f :: functor, 's) \text{ stateT}) =$
 $StateT \cdot (\Lambda s. fmap \cdot (\Lambda (x, s'). (f \cdot x, s')) \cdot (m \cdot s))$
unfolding *fmap-def [where 'f=(f, 's) stateT]*
by (*simp add: coerce-simp eta-cfun*)

lemma *runStateT-fmap [simp]*:
 $runStateT \cdot (fmap \cdot f \cdot m) \cdot s = fmap \cdot (\Lambda (x, s'). (f \cdot x, s')) \cdot (runStateT \cdot m \cdot s)$
by (*induct m rule: stateT-induct, simp*)

lemma *return-stateT-def*:
 $(return :: - \rightarrow 'a \cdot ('m :: monad, 's) \text{ stateT}) =$
 $(\Lambda x. StateT \cdot (\Lambda s. return \cdot (x, s)))$
unfolding *return-def [where 'm=(m, 's) stateT] returnU-stateT-def*
by (*simp add: coerce-simp*)

lemma *bind-stateT-def*:
 $bind = (\Lambda m k. StateT \cdot (\Lambda s. runStateT \cdot m \cdot s \gg (\Lambda (x, s'). runStateT \cdot (k \cdot x) \cdot s')))$
apply (*subst bind-def, subst bindU-stateT-def*)
apply (*simp add: coerce-simp*)
apply (*simp add: coerce-idem domain-defl-simps monofun-cfun*)
apply (*simp add: eta-cfun*)
done

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

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

lemma *runStateT-bind [simp]*:
 $runStateT \cdot (m \gg k) \cdot s = runStateT \cdot m \cdot s \gg (\Lambda (x, s'). runStateT \cdot (k \cdot x) \cdot s')$
unfolding *bind-stateT-def by simp*

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)
apply (simp add: errorT-abs-def coerce-def)
apply (simp add: emb-prj-emb prj-emb-prj DEFL-eq-errorT)
done
```

```
lemma coerce-ErrorT [simp]: coerce·(ErrorT·k) = ErrorT·(coerce·k)
unfolding ErrorT-def by simp
```

```
lemma errorT-cases [case-names ErrorT]:
  obtains k where y = ErrorT·k
proof
  show y = ErrorT·(runErrorT·y)
    by (cases y, simp-all)
qed
```

```
lemma ErrorT-runErrorT [simp]: ErrorT·(runErrorT·m) = m
by (cases m rule: errorT-cases, simp)
```

```
lemma errorT-induct [case-names ErrorT]:
  fixes P :: 'a'('f::functor,'e) errorT ⇒ bool
  assumes ∧k. P (ErrorT·k)
  shows P y
by (cases y rule: errorT-cases, simp add: assms)
```

```
lemma errorT-eq-iff:
  a = b ⟷ runErrorT·a = runErrorT·b
apply (cases a rule: errorT-cases)
apply (cases b rule: errorT-cases)
apply simp
done
```

```
lemma errorT-eqI:
  runErrorT·a = runErrorT·b ⟹ a = b
by (simp add: errorT-eq-iff)
```

```
lemma runErrorT-coerce [simp]:
```

```

  runErrorT.(coerce.k) = coerce.(runErrorT.k)
by (induct k rule: errorT-induct, simp)

```

16.2 Functor class instance

```

lemma fmap-error-def: fmap = error-map.ID
apply (rule cfun-eqI, rename-tac f)
apply (rule cfun-eqI, rename-tac x)
apply (case-tac x rule: error.exhaust, simp-all)
apply (simp add: error-map-def fix-const)
apply (simp add: error-map-def fix-const Err-def)
apply (simp add: error-map-def fix-const Ok-def)
done

```

```

lemma fmapU-ErrorT [simp]:
  fmapU.f.(ErrorT.m) = ErrorT.(fmap.(fmap.f).m)
unfolding fmapU-errorT-def errorT-map-def fmap-error-def fix-const ErrorT-def
by simp

```

```

lemma runErrorT-fmapU [simp]:
  runErrorT.(fmapU.f.m) = fmap.(fmap.f).(runErrorT.m)
by (induct m rule: errorT-induct) simp

```

```

instance errorT :: (functor, domain) functor
proof
  fix f g and xs :: udom.('a, 'b) errorT
  show fmapU.f.(fmapU.g.xs) = fmapU.( $\Lambda$  x. f.(g.x)).xs
  apply (induct xs rule: errorT-induct)
  apply (simp add: fmap-fmap eta-cfun)
  done
qed

```

16.3 Transfer properties to polymorphic versions

```

lemma fmap-ErrorT [simp]:
  fixes f :: 'a  $\rightarrow$  'b and m :: 'a.('m::functor)
  shows fmap.f.(ErrorT.m) = ErrorT.(fmap.(fmap.f).m)
unfolding fmap-def [where 'f=('m,'e) errorT]
by (simp-all add: coerce-simp eta-cfun)

```

```

lemma runErrorT-fmap [simp]:
  fixes f :: 'a  $\rightarrow$  'b and m :: 'a.('m::functor,'e) errorT
  shows runErrorT.(fmap.f.m) = fmap.(fmap.f).(runErrorT.m)
using fmap-ErrorT [of f runErrorT.m]
by simp

```

```

lemma errorT-fmap-strict [simp]:
  shows fmap.f.( $\perp$ ::'a.('m::monad,'e) errorT) =  $\perp$ 
by (simp add: errorT-eq-iff fmap-strict)

```

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 $\text{unitET} :: 'a \rightarrow 'a \cdot ('m :: \text{monad}, 'e) \text{ errorT}$
where $\text{unitET} = (\lambda x. \text{ErrorT} \cdot (\text{return} \cdot (\text{Ok} \cdot x)))$

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

definition $\text{liftET} :: 'a \cdot 'm :: \text{monad} \rightarrow 'a \cdot ('m, 'e) \text{ errorT}$
where $\text{liftET} = (\lambda m. \text{ErrorT} \cdot (\text{fmap} \cdot \text{Ok} \cdot m))$

definition $\text{throwET} :: 'e \rightarrow 'a \cdot ('m :: \text{monad}, 'e) \text{ errorT}$
where $\text{throwET} = (\lambda e. \text{ErrorT} \cdot (\text{return} \cdot (\text{Err} \cdot e)))$

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

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

lemma $\text{runErrorT} \cdot \text{unitET} \text{ [simp]}:$
 $\text{runErrorT} \cdot (\text{unitET} \cdot x) = \text{return} \cdot (\text{Ok} \cdot x)$
unfolding unitET-def **by** simp

lemma $\text{runErrorT} \cdot \text{bindET} \text{ [simp]}:$
 $\text{runErrorT} \cdot (\text{bindET} \cdot m \cdot k) = \text{bind} \cdot (\text{runErrorT} \cdot m) \cdot$
 $(\lambda n. \text{case } n \text{ of } \text{Err} \cdot e \Rightarrow \text{return} \cdot (\text{Err} \cdot e) \mid \text{Ok} \cdot x \Rightarrow \text{runErrorT} \cdot (k \cdot x))$
unfolding bindET-def **by** simp

lemma $\text{runErrorT} \cdot \text{liftET} \text{ [simp]}:$
 $\text{runErrorT} \cdot (\text{liftET} \cdot m) = \text{fmap} \cdot \text{Ok} \cdot m$
unfolding liftET-def **by** simp

lemma $\text{runErrorT} \cdot \text{throwET} \text{ [simp]}:$
 $\text{runErrorT} \cdot (\text{throwET} \cdot e) = \text{return} \cdot (\text{Err} \cdot e)$

unfolding *throwET-def* **by** *simp*

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

$$\begin{aligned} \text{runErrorT} \cdot (\text{catchET} \cdot m \cdot h) = \\ \text{bind} \cdot (\text{runErrorT} \cdot m) \cdot (\lambda n. \text{case } n \text{ of} \\ \text{Err} \cdot e \Rightarrow \text{runErrorT} \cdot (h \cdot e) \mid \text{Ok} \cdot x \Rightarrow \text{return} \cdot (\text{Ok} \cdot x)) \end{aligned}$$

unfolding *catchET-def* **by** *simp*

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

$$\begin{aligned} \text{runErrorT} \cdot (\text{fmapET} \cdot f \cdot m) = \\ \text{bind} \cdot (\text{runErrorT} \cdot m) \cdot (\lambda n. \text{case } n \text{ of} \\ \text{Err} \cdot e \Rightarrow \text{return} \cdot (\text{Err} \cdot e) \mid \text{Ok} \cdot x \Rightarrow \text{return} \cdot (\text{Ok} \cdot (f \cdot x))) \end{aligned}$$

unfolding *fmapET-def* **by** *simp*

16.5 Laws

lemma *bindET-unitET* [*simp*]:

$$\text{bindET} \cdot (\text{unitET} \cdot x) \cdot k = k \cdot x$$

by (*rule errorT-eqI*, *simp*)

lemma *catchET-unitET* [*simp*]:

$$\text{catchET} \cdot (\text{unitET} \cdot x) \cdot h = \text{unitET} \cdot x$$

by (*rule errorT-eqI*, *simp*)

lemma *catchET-throwET* [*simp*]:

$$\text{catchET} \cdot (\text{throwET} \cdot e) \cdot h = h \cdot e$$

by (*rule errorT-eqI*, *simp*)

lemma *liftET-return*:

$$\text{liftET} \cdot (\text{return} \cdot x) = \text{unitET} \cdot x$$

by (*rule errorT-eqI*, *simp* *add*: *fmap-return*)

lemma *liftET-bind*:

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

by (*rule errorT-eqI*, *simp* *add*: *fmap-bind* *bind-fmap*)

lemma *bindET-throwET*:

$$\text{bindET} \cdot (\text{throwET} \cdot e) \cdot k = \text{throwET} \cdot e$$

by (*rule errorT-eqI*, *simp*)

lemma *bindET-bindET*:

$$\text{bindET} \cdot (\text{bindET} \cdot m \cdot h) \cdot k = \text{bindET} \cdot m \cdot (\lambda x. \text{bindET} \cdot (h \cdot x) \cdot k)$$

apply (*rule errorT-eqI*)

apply *simp*

apply (*simp* *add*: *bind-bind*)

apply (*rule cfun-arg-cong*)

apply (*rule cfun-eqI*, *simp*)

apply (*case-tac* *x*)

apply (*simp* *add*: *bind-strict*)

```

apply simp
apply simp
done

```

```

lemma fmapET-fmapET:
  fmapET·f·(fmapET·g·m) = fmapET·(λ x. f·(g·x))·m
by (simp add: fmapET-def bindET-bindET)

```

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
by (simp add: errorT-eq-iff assms)

```

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
unfolding errorT-eq-iff
apply simp
apply (rule trans [OF - monad-right-unit])
apply (rule cfun-arg-cong)
apply (rule cfun-eqI)
apply (case-tac x, simp-all add: assms)
done

```

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
using assms

```

```

apply (induct set: invar)
apply (rule errorT-eqI, simp add: bind-strict)
apply (rule admD, simp, assumption, assumption)
apply (rule errorT-eqI, simp)
apply (simp add: errorT-eq-iff bind-bind)
apply (rule cfun-arg-cong, rule cfun-eqI, simp)
apply (case-tac x, simp add: bind-strict, simp, simp)
apply (rule errorT-eqI, simp)
apply (simp add: errorT-eq-iff bind-bind)
apply (rule cfun-arg-cong, rule cfun-eqI, simp)
apply (case-tac x, simp add: bind-strict, simp, simp)
apply (rule errorT-eqI, simp add: monad-fmap bind-bind)
done

```

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))
using assms
apply (induct set: invar)
apply (rule errorT-eqI, simp add: bind-strict fmap-strict)
apply (rule admD, simp, assumption, assumption)
apply (rule errorT-eqI, simp add: fmap-return)
apply (simp add: errorT-eq-iff bind-bind fmap-bind)
apply (rule cfun-arg-cong, rule cfun-eqI, simp)
apply (case-tac x)
apply (simp add: bind-strict fmap-strict)
apply (simp add: fmap-return)
apply simp
apply (rule errorT-eqI, simp add: fmap-return)
apply (simp add: errorT-eq-iff bind-bind fmap-bind)
apply (rule cfun-arg-cong, rule cfun-eqI, simp)
apply (case-tac x)
apply (simp add: bind-strict fmap-strict)
apply simp
apply (simp add: fmap-return)
apply (rule errorT-eqI, simp add: monad-fmap bind-bind return-error-def)
done

```

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)
  unfolding invar'-def by (simp add: fmapET-def)

lemma invar'-fmapET: invar' m  $\implies$  invar' (fmapET.f.m)
  unfolding invar'-def
  by (erule subst, simp add: fmapET-def bindET-bindET eta-cfun)

lemma invar'-bindET:  $\llbracket \text{invar}' m; \bigwedge x. \text{invar}' (k.x) \rrbracket \implies \text{invar}' (\text{bindET}.m.k)$ 
  unfolding invar'-def
  by (simp add: fmapET-def bindET-bindET eta-cfun)

lemma invar'-throwET: invar' (throwET.e)
  unfolding invar'-def by (simp add: fmapET-def bindET-throwET eta-cfun)

lemma invar'-catchET:  $\llbracket \text{invar}' m; \bigwedge e. \text{invar}' (h.e) \rrbracket \implies \text{invar}' (\text{catchET}.m.h)$ 
  unfolding invar'-def
  apply (simp add: fmapET-def eta-cfun)
  apply (rule errorT-eqI)
  apply (simp add: bind-bind eta-cfun)
  apply (rule cfun-arg-cong)
  apply (rule cfun-eqI)
  apply (case-tac x)
  apply (simp add: bind-strict)
  apply simp
  apply (drule-tac x=e in meta-spec)
  apply (erule-tac t=h.e in subst) back
  apply (simp add: eta-cfun)
  apply simp
  done

lemma invar'-liftET: invar' (liftET.m)
  unfolding invar'-def
  apply (simp add: fmapET-def errorT-eq-iff)
  apply (simp add: monad-fmap bind-bind)
  done

lemma invar'-bottom: invar'  $\perp$ 
  unfolding invar'-def fmapET-def
  by (simp add: errorT-eq-iff bind-strict)

lemma adm-invar': adm invar'
  unfolding invar'-def [abs-def] by simp

```

All monad laws are preserved by values satisfying the invariant.

```

lemma bindET-fmapET-unitET:
  shows bindET.(fmapET.f.m).unitET = fmapET.f.m
by (simp add: fmapET-def bindET-bindET)

```

```

lemma invar'-right-unit: invar' m  $\implies$  bindET.m.unitET = m

```

unfolding *invar'*-def **by** (*erule subst*, *rule bindET-fmapET-unitET*)

lemma *invar'*-monad-fmap:

invar' *m* \implies *fmapET*·*f*·*m* = *bindET*·*m*·(Λ *x*. *unitET*·(*f*·*x*))

unfolding *invar'*-def **by** (*erule subst*, *simp add: errorT-eq-iff*)

lemma *invar'*-bind-assoc:

$\llbracket \text{invar}'\ m; \bigwedge x. \text{invar}'\ (f \cdot x); \bigwedge y. \text{invar}'\ (g \cdot y) \rrbracket$

$\implies \text{bindET} \cdot (\text{bindET} \cdot m \cdot f) \cdot g = \text{bindET} \cdot m \cdot (\Lambda\ x. \text{bindET} \cdot (f \cdot x) \cdot g)$

by (*rule bindET-bindET*)

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·(*writerT-abs*·*x*) = *writerT-abs*·(*coerce*·*x*)

apply (*simp add: writerT-abs-def coerce-def*)

apply (*simp add: emb-prj-emb prj-emb-prj DEFL-eq-writerT*)

done

lemma *coerce-WriterT* [*simp*]: *coerce*·(*WriterT*·*k*) = *WriterT*·(*coerce*·*k*)

unfolding *WriterT-def* **by** *simp*

lemma *writerT-cases* [*case-names WriterT*]:

obtains *k* **where** *y* = *WriterT*·*k*

```

proof
  show  $y = \text{WriterT} \cdot (\text{runWriterT} \cdot y)$ 
  by (cases y, simp-all)
qed

lemma WriterT-runWriterT [simp]:  $\text{WriterT} \cdot (\text{runWriterT} \cdot m) = m$ 
by (cases m rule: writerT-cases, simp)

lemma writerT-induct [case-names WriterT]:
  fixes  $P :: 'a \rightarrow ('f :: \text{functor}, 'e) \text{writerT} \Rightarrow \text{bool}$ 
  assumes  $\bigwedge k. P (\text{WriterT} \cdot k)$ 
  shows  $P y$ 
by (cases y rule: writerT-cases, simp add: assms)

lemma writerT-eq-iff:
   $a = b \iff \text{runWriterT} \cdot a = \text{runWriterT} \cdot b$ 
apply (cases a rule: writerT-cases)
apply (cases b rule: writerT-cases)
apply simp
done

lemma writerT-below-iff:
   $a \sqsubseteq b \iff \text{runWriterT} \cdot a \sqsubseteq \text{runWriterT} \cdot b$ 
apply (cases a rule: writerT-cases)
apply (cases b rule: writerT-cases)
apply simp
done

lemma writerT-eqI:
   $\text{runWriterT} \cdot a = \text{runWriterT} \cdot b \implies a = b$ 
by (simp add: writerT-eq-iff)

lemma writerT-belowI:
   $\text{runWriterT} \cdot a \sqsubseteq \text{runWriterT} \cdot b \implies a \sqsubseteq b$ 
by (simp add: writerT-below-iff)

lemma runWriterT-coerce [simp]:
   $\text{runWriterT} \cdot (\text{coerce} \cdot k) = \text{coerce} \cdot (\text{runWriterT} \cdot k)$ 
by (induct k rule: writerT-induct, simp)

```

17.2 Functor class instance

```

lemma fmap-writer-def: fmap = writer-map · ID
apply (rule cfun-eqI, rename-tac f)
apply (rule cfun-eqI, rename-tac x)
apply (case-tac x rule: writer.exhaust, simp-all)
apply (simp add: writer-map-def fix-const)
apply (simp add: writer-map-def fix-const Writer-def)
done

```

lemma *fmapU-WriterT* [simp]:
 $fmapU\ f \cdot (WriterT\ m) = WriterT \cdot (fmap \cdot (fmap\ f) \cdot m)$
unfolding *fmapU-writerT-def writerT-map-def fmap-writer-def fix-const*
WriterT-def **by** *simp*

lemma *runWriterT-fmapU* [simp]:
 $runWriterT \cdot (fmapU\ f \cdot m) = fmap \cdot (fmap\ f) \cdot (runWriterT\ m)$
by (*induct m rule: writerT-induct*) *simp*

instance *writerT* :: (*functor*, *domain*) *functor*

proof

fix $f\ g :: udom \rightarrow udom$ **and** $xs :: udom \cdot ('a, 'b)\ writerT$
show $fmapU\ f \cdot (fmapU\ g \cdot xs) = fmapU \cdot (\Lambda\ x.\ f \cdot (g \cdot x)) \cdot xs$
apply (*induct xs rule: writerT-induct*)
apply (*simp add: fmap-fmap eta-cfun*)
done

qed

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

definition *bindWT* :: $'a \cdot ('m :: monad, 'w :: monoid)\ writerT \rightarrow ('a \rightarrow 'b \cdot ('m, 'w)\ writerT) \rightarrow 'b \cdot ('m, 'w)\ writerT$
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* :: $'a \cdot 'm \rightarrow 'a \cdot ('m :: monad, 'w :: monoid)\ writerT$
where $liftWT = (\Lambda\ m.\ WriterT \cdot (fmap \cdot (Writer \cdot mempty) \cdot m))$

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

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

lemma *runWriterT-fmap* [simp]:
 $\text{runWriterT} \cdot (\text{fmap} \cdot f \cdot m) = \text{fmap} \cdot (\text{fmap} \cdot f) \cdot (\text{runWriterT} \cdot m)$
by (*subst fmap-def, simp add: coerce-simp eta-cfun*)

lemma *runWriterT-unitWT* [simp]:
 $\text{runWriterT} \cdot (\text{unitWT} \cdot x) = \text{return} \cdot (\text{Writer} \cdot \text{mempty} \cdot x)$
unfolding *unitWT-def* **by** *simp*

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

lemma *runWriterT-liftWT* [simp]:
 $\text{runWriterT} \cdot (\text{liftWT} \cdot m) = \text{fmap} \cdot (\text{Writer} \cdot \text{mempty}) \cdot m$
unfolding *liftWT-def* **by** *simp*

lemma *runWriterT-tellWT* [simp]:
 $\text{runWriterT} \cdot (\text{tellWT} \cdot x \cdot w) = \text{return} \cdot (\text{Writer} \cdot w \cdot x)$
unfolding *tellWT-def* **by** *simp*

lemma *runWriterT-fmapWT* [simp]:
 $\text{runWriterT} \cdot (\text{fmapWT} \cdot f \cdot m) =$
 $\text{runWriterT} \cdot m \gg (\Lambda (\text{Writer} \cdot w \cdot x). \text{return} \cdot (\text{Writer} \cdot w \cdot (f \cdot x)))$
by (*simp add: fmapWT-def bindWT-def mempty-right*)

17.4 Laws

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

lemma *liftWT-return*:
 $\text{liftWT} \cdot (\text{return} \cdot x) = \text{unitWT} \cdot x$
by (*rule writerT-eqI, simp add: fmap-return*)

lemma *liftWT-bind*:
 $\text{liftWT} \cdot (\text{bind} \cdot m \cdot k) = \text{bindWT} \cdot (\text{liftWT} \cdot m) \cdot (\text{liftWT} \circ k)$
by (*rule writerT-eqI*)
(simp add: monad-fmap bind-bind mempty-left)

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

lemma *fmapWT-fmapWT*:
 $\text{fmapWT} \cdot f \cdot (\text{fmapWT} \cdot g \cdot m) = \text{fmapWT} \cdot (\Lambda x. f \cdot (g \cdot x)) \cdot m$
apply (*simp add: writerT-eq-iff bind-bind*)
apply (*rule cfun-arg-cong, rule cfun-eqI, simp*)
apply (*case-tac x, simp add: bind-strict, simp add: mempty-right*)

done

lemma *fmapWT-unitWT*:

$fmapWT \cdot f \cdot (unitWT \cdot x) = unitWT \cdot (f \cdot x)$
by (*simp add: writerT-eq-iff mempty-right*)

lemma *fmapWT-bindWT*:

$fmapWT \cdot f \cdot (bindWT \cdot m \cdot k) = bindWT \cdot m \cdot (\Lambda x. fmapWT \cdot f \cdot (k \cdot x))$
apply (*simp add: writerT-eq-iff bind-bind*)
apply (*rule cfun-arg-cong, rule cfun-eqI, rename-tac x, simp*)
apply (*case-tac x, simp add: bind-strict, simp add: bind-bind*)
apply (*rule cfun-arg-cong, rule cfun-eqI, rename-tac y, simp*)
apply (*case-tac y, simp add: bind-strict, simp add: mempty-right*)
done

lemma *bindWT-fmapWT*:

$bindWT \cdot (fmapWT \cdot f \cdot m) \cdot k = bindWT \cdot m \cdot (\Lambda x. k \cdot (f \cdot x))$
apply (*simp add: writerT-eq-iff bind-bind*)
apply (*rule cfun-arg-cong, rule cfun-eqI, rename-tac x, simp*)
apply (*case-tac x, simp add: bind-strict, simp add: mempty-right*)
done

The left unit monad law is not satisfied in general.

lemma *bindWT-unitWT-counterexample*:

fixes $k :: 'a \rightarrow 'b \cdot ('m :: monad, 'w :: monoid) \text{ writerT}$
assumes $1: k \cdot x = \text{WriterT} \cdot (\text{return} \cdot \perp)$
assumes $2: \text{return} \cdot \perp \neq (\perp :: ('b \cdot 'w \text{ writer}) \cdot 'm :: monad)$
shows $bindWT \cdot (unitWT \cdot x) \cdot k \neq k \cdot x$
by (*simp add: writerT-eq-iff mempty-left assms*)

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

lemma *bindWT-unitWT-restricted*:

fixes $k :: 'a \rightarrow 'b \cdot ('m :: monad, 'w :: monoid) \text{ writerT}$
assumes $\text{return} \cdot \perp = (\perp :: ('b \cdot 'w \text{ writer}) \cdot 'm)$
shows $bindWT \cdot (unitWT \cdot x) \cdot k = k \cdot x$
unfolding *writerT-eq-iff*
apply (*simp add: mempty-left*)
apply (*rule trans [OF - monad-right-unit]*)
apply (*rule cfun-arg-cong*)
apply (*rule cfun-eqI*)
apply (*case-tac x, simp-all add: assms*)
done

The associativity of *bindWT* holds unconditionally.

lemma *bindWT-bindWT*:

$bindWT \cdot (bindWT \cdot m \cdot h) \cdot k = bindWT \cdot m \cdot (\Lambda x. bindWT \cdot (h \cdot x) \cdot k)$
apply (*rule writerT-eqI*)

```

apply simp
apply (simp add: bind-bind)
apply (rule cfun-arg-cong)
apply (rule cfun-eqI, simp)
apply (case-tac x)
apply (simp add: bind-strict)
apply (simp add: bind-bind)
apply (rule cfun-arg-cong)
apply (rule cfun-eqI, simp, rename-tac y)
apply (case-tac y)
apply (simp add: bind-strict)
apply (simp add: bind-bind)
apply (rule cfun-arg-cong)
apply (rule cfun-eqI, simp, rename-tac z)
apply (case-tac z)
apply (simp add: bind-strict)
apply (simp add: mappend-assoc)
done

```

The right unit monad law is not satisfied in general.

```

lemma bindWT-unitWT-right-counterexample:
  fixes m :: 'a.('m::monad,'w::monoid) writerT
  assumes m = WriterT.(return. $\perp$ )
  assumes return. $\perp$   $\neq$  ( $\perp$  :: ('a.'w writer).'m)
  shows bindWT.m.unitWT  $\neq$  m
by (simp add: writerT-eq-iff assms)

```

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

```

lemma bindWT-unitWT-right-restricted:
  fixes m :: 'a.('m::monad,'w::monoid) writerT
  assumes return. $\perp$  = ( $\perp$  :: ('a.'w writer).'m)
  shows bindWT.m.unitWT = m
unfolding writerT-eq-iff
apply simp
apply (rule trans [OF - monad-right-unit])
apply (rule cfun-arg-cong)
apply (rule cfun-eqI)
apply (case-tac x, simp-all add: assms mempty-right)
done

```

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.('m::monad, 'w::monoid) writerT  $\Rightarrow$  bool
  where invar-bottom: invar  $\perp$ 

```

$$\begin{array}{l}
| \text{invar-lub}: \bigwedge Y. \llbracket \text{chain } Y; \bigwedge i. \text{invar } (Y \ i) \rrbracket \implies \text{invar } (\bigsqcup i. Y \ i) \\
| \text{invar-unitWT}: \bigwedge x. \text{invar } (\text{unitWT} \cdot x) \\
| \text{invar-bindWT}: \bigwedge m \ k. \llbracket \text{invar } m; \bigwedge x. \text{invar } (k \cdot x) \rrbracket \implies \text{invar } (\text{bindWT} \cdot m \cdot k) \\
| \text{invar-tellWT}: \bigwedge x \ w. \text{invar } (\text{tellWT} \cdot x \cdot w) \\
| \text{invar-liftWT}: \bigwedge m. \text{invar } (\text{liftWT} \cdot m)
\end{array}$$

Right unit is satisfied for arguments built from standard functions.

```

lemma bindWT-unitWT-right-invar:
  fixes  $m :: 'a \rightarrow ('m :: \text{monad}, 'w :: \text{monoid}) \text{ writerT}$ 
  assumes invar m
  shows bindWT · m · unitWT = m
using assms proof (induct set: invar)
  case invar-bottom thus ?case
    by (rule writerT-eqI, simp add: bind-strict)
next
  case invar-lub thus ?case
    by - (rule admD, simp, assumption, assumption)
next
  case invar-unitWT thus ?case
    by (rule writerT-eqI, simp add: bind-bind mempty-left)
next
  case invar-bindWT thus ?case
    apply (simp add: writerT-eq-iff bind-bind)
    apply (rule cfun-arg-cong, rule cfun-eqI, simp)
    apply (case-tac x, simp add: bind-strict, simp add: bind-bind)
    apply (rule cfun-arg-cong, rule cfun-eqI, simp, rename-tac y)
    apply (case-tac y, simp add: bind-strict, simp add: mempty-right)
    done
next
  case invar-tellWT thus ?case
    by (simp add: writerT-eq-iff mempty-right)
next
  case invar-liftWT thus ?case
    by (rule writerT-eqI, simp add: monad-fmap bind-bind mempty-right)
qed

```

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

```

lemma writerT-left-unit-invar-lemma:
  assumes invar m
  shows runWriterT · m  $\gg=$  ( $\Lambda ( \text{Writer} \cdot w \cdot x ). \text{return} \cdot ( \text{Writer} \cdot w \cdot x )$ ) = runWriterT · m
using assms proof (induct m set: invar)
  case invar-bottom thus ?case
    by (simp add: bind-strict)
next
  case invar-lub thus ?case
    by - (rule admD, simp, assumption, assumption)
next
  case invar-unitWT thus ?case

```

```

    by simp
next
case invar-bindWT thus ?case
  apply (simp add: bind-bind)
  apply (rule cfun-arg-cong)
  apply (rule cfun-eqI, simp, rename-tac n)
  apply (case-tac n, simp add: bind-strict)
  apply (simp add: bind-bind)
  apply (rule cfun-arg-cong)
  apply (rule cfun-eqI, simp, rename-tac p)
  apply (case-tac p, simp add: bind-strict)
  apply simp
done
next
case invar-tellWT thus ?case
  by simp
next
case invar-liftWT thus ?case
  by (simp add: monad-fmap bind-bind)
qed

lemma bindWT-unitWT-invar:
  assumes invar (k·x)
  shows bindWT·(unitWT·x)·k = k·x
apply (simp add: writerT-eq-iff mempty-left)
apply (rule writerT-left-unit-invar-lemma [OF assms])
done

```

17.6 Invariant expressed as a deflation

definition $invar' :: 'a \rightarrow ('m::monad, 'w::monoid) \rightarrow writerT \Rightarrow bool$
 where $invar' m \longleftrightarrow fmap WT \cdot ID \cdot m = m$

All standard operations preserve the invariant.

lemma $invar'$ -bottom: $invar' \perp$
 unfolding $invar'$ -def by (simp add: writerT-eq-iff bind-strict)

lemma adm - $invar'$: $adm\ invar'$
 unfolding $invar'$ -def [abs-def] by simp

lemma $invar'$ -unitWT: $invar' (unitWT \cdot x)$
 unfolding $invar'$ -def by (simp add: writerT-eq-iff)

lemma $invar'$ -bindWT: $\llbracket invar' m; \bigwedge x. invar' (k \cdot x) \rrbracket \implies invar' (bindWT \cdot m \cdot k)$
 unfolding $invar'$ -def
 apply (erule subst)
 apply (simp add: writerT-eq-iff)
 apply (simp add: bind-bind)
 apply (rule cfun-arg-cong)

```

apply (rule cfun-eqI, case-tac x)
apply (simp add: bind-strict)
apply simp
apply (simp add: bind-bind)
apply (rule cfun-arg-cong)
apply (rule cfun-eqI, rename-tac x, case-tac x)
apply (simp add: bind-strict)
apply simp
done

lemma invar'-tellWT: invar' (tellWT.x.w)
  unfolding invar'-def by (simp add: writerT-eq-iff)

lemma invar'-liftWT: invar' (liftWT.m)
  unfolding invar'-def by (simp add: writerT-eq-iff monad-fmap bind-bind)

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)
apply (simp add: fmapWT-def writerT-eq-iff bind-bind)
apply (rule cfun-arg-cong, rule cfun-eqI, simp)
apply (case-tac x, simp-all add: bind-strict mempty-left)
done

Right unit is satisfied for arguments built from fmap.

lemma bindWT-fmapWT-unitWT:
  shows bindWT.(fmapWT.f.m).unitWT = fmapWT.f.m
apply (simp add: bindWT-fmapWT)
apply (simp add: fmapWT-def)
done

All monad laws are preserved by values satisfying the invariant.

lemma invar'-right-unit: invar' m ⇒ bindWT.m.unitWT = m
unfolding invar'-def by (erule subst, rule bindWT-fmapWT-unitWT)

lemma invar'-monad-fmap:
  invar' m ⇒ fmapWT.f.m = bindWT.m.(λ x. unitWT.(f.x))
  unfolding invar'-def
  by (erule subst, simp add: writerT-eq-iff mempty-right)

lemma invar'-bind-assoc:
   $\llbracket \text{invar}' m; \bigwedge x. \text{invar}' (f.x); \bigwedge y. \text{invar}' (g.y) \rrbracket$ 
   $\Rightarrow \text{bindWT} . (\text{bindWT} . m . f) . g = \text{bindWT} . m . (\lambda x. \text{bindWT} . (f.x) . g)$ 
  by (rule bindWT-bindWT)

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.