Tycon: Type Constructor Classes and Monad Transformers

Brian Huffman

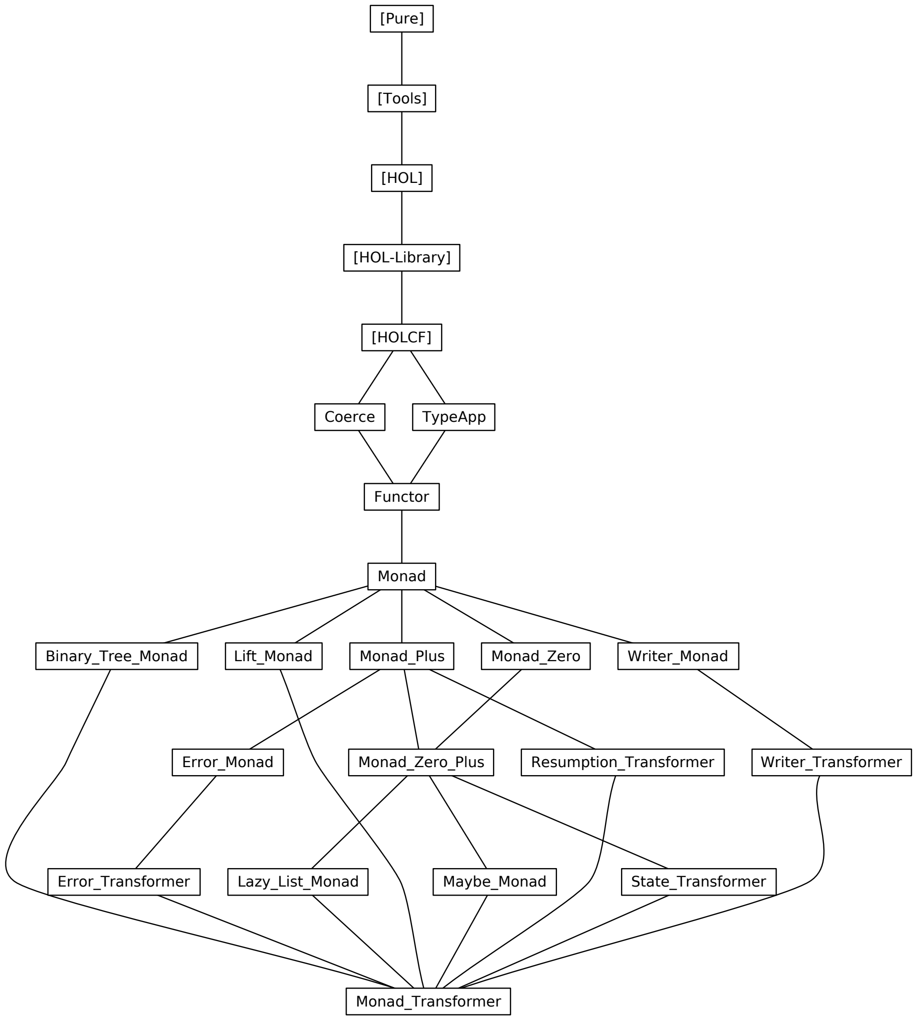
December 14, 2021

Abstract

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

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

Contents


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

1 Type Application

theory *TypeApp*
imports *HOLCF*
begin

1.1 Class of type constructors

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

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

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

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

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

translations *TC('a)* $\rightleftharpoons$ *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)*$\cdot$*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* ((-·-) [*999*,*1000*] *999*)

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

lemma *TC-DEFL*: *TC*(*'t*::*tycon*)·*DEFL*(*'a*) = *DEFL*(*'a*·*'t*)
by (*rule DEFL-app* [*symmetric*])

lemma *DEFL-app-mono* [*simp*, *intro*]:
DEFL(*'a*) ⊑ *DEFL*(*'b*) ⟹ *DEFL*(*'a*·*'t*::*tycon*) ⊑ *DEFL*(*'b*·*'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* → *udom* and *prj*::*udom* → *'a*. By composing the *prj* and *emb* functions together, we can coerce values between any two types in class *domain*.

definition *coerce* :: *'a* → *'b*
where *coerce* ≡ *prj oo emb*

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

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

translations
EMB(*'a*) ⇀ *CONST emb* :: *'a* → *udom*
PRJ(*'a*) ⇀ *CONST prj* :: *udom* → *'a*
COERCE(*'a*,*'b*) ⇀ *CONST coerce* :: *'a* → *'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*: $coerce{\cdot}x = prj{\cdot}(emb{\cdot}x)$
by (*simp add*: *coerce-def*)

lemma *prj-emb*: $prj{\cdot}(emb{\cdot}x) = coerce{\cdot}x$
by (*simp add*: *coerce-def*)

lemma *coerce-strict* [*simp*]: $coerce{\cdot}\bot = \bot$
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*:
  $DEFL('a) \sqsubseteq DEFL('b)$
  $\Longrightarrow EMB('b){\cdot}(COERCE('a,'b){\cdot}x) = EMB('a){\cdot}x$
by (*simp add*: *beta-coerce emb-prj-emb*)

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

lemma *coerce-idem* [*simp*]:
  $DEFL('a) \sqsubseteq DEFL('b)$
  $\Longrightarrow COERCE('b,'c){\cdot}(COERCE('a,'b){\cdot}x) = COERCE('a,'c){\cdot}x$

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

lemma *coerce-u*: $coerce = u\text{-}map\cdot coerce$
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'*: $coerce = cfun\text{-}map\cdot coerce\cdot coerce$
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*: $coerce = ssum\text{-}map\cdot coerce\cdot coerce$
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*]: $coerce \cdot f \cdot x = coerce \cdot (f \cdot (coerce \cdot x))$
by (*simp add*: *coerce-cfun'*)

lemma *coerce-cfun*: $coerce \cdot f = coerce\ oo\ f\ oo\ coerce$
by (*simp add*: *cfun-eqI*)

lemma *coerce-cfun-app* [*coerce-simp*]:
  $coerce \cdot f = (\Lambda\ x.\ coerce \cdot (f \cdot (coerce \cdot x)))$
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 $f :: 'a \to 'a$ with a deflation $t :: udom\ defl$: $isodefl\ f\ t = (cast \cdot t = EMB('a)\ oo\ f\ oo\ PRJ('a))$.

class *prefunctor* = *tycon* +
  fixes $fmapU :: (udom \to udom) \to udom \cdot 'a \to udom \cdot 'a{::}tycon$
  assumes *isodefl-fmapU*:
    $isodefl\ (fmapU \cdot (cast \cdot t))\ (TC('a{::}tycon) \cdot t)$

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

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

$fmapU \cdot f \cdot (fmapU \cdot g \cdot xs) = fmapU \cdot (\Lambda\ x.\ f \cdot (g \cdot x)) \cdot xs$

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

definition $fmap :: ('a \to 'b) \to 'a \cdot 'f \to 'b \cdot 'f{::}functor$
where $fmap = coerce \cdot (fmapU :: \text{-} \to udom \cdot 'f \to udom \cdot 'f)$

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

lemma *isodefl-cast*: *isodefl* $(cast \cdot t)$ t
by (*simp add*: *isodefl-def*)

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

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

lemma *isodefl-fmap*:
assumes *isodefl d t*
shows *isodefl* $(fmap \cdot d :: 'a \cdot 'f \to \text{-})$ $(TC('f{::}functor) \cdot t)$
proof −
have *deflation-d*: *deflation d*
using *assms* **by** (*rule isodefl-imp-deflation*)
have *cast-t*: $cast \cdot t = emb\ oo\ d\ oo\ prj$
using *assms* **unfolding** *isodefl-def* **.**
have *t-below*: $t \sqsubseteq DEFL('a)$
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*: $fmap \cdot d = PRJ('a \cdot 'f)\ oo\ cast \cdot (TC('f) \cdot t)\ oo\ emb$
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*]: $fmap\cdot ID = ID$
apply (*rule isodefl-DEFL-imp-ID*)
apply (*subst DEFL-app*)
apply (*rule isodefl-fmap*)
apply (*rule isodefl-ID-DEFL*)
done

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

lemma *coerce-coerce-eq-fmapU-cast* [*coerce-simp*]:
  fixes xs :: $udom\cdot'f$::*functor*
  shows $COERCE('a\cdot'f,\ udom\cdot'f)\cdot(COERCE(udom\cdot'f,\ 'a\cdot'f)\cdot xs) =$
    $fmapU\cdot(cast\cdot DEFL('a))\cdot xs$
by (*simp add*: *coerce-def emb-prj DEFL-app cast-TC*)

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

lemma *fmap-cfcomp*: $fmap\cdot(f\ oo\ g) = fmap\cdot f\ oo\ fmap\cdot g$
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*:
  $deflation\ d \Longrightarrow deflation\ (fmap\cdot d)$
 apply (*rule deflation.intro*)
  apply (*simp add*: *fmap-fmap deflation.idem eta-cfun*)
 apply (*subgoal-tac* $fmap\cdot d\cdot x \sqsubseteq fmap\cdot ID\cdot x$, *simp*)
 apply (*rule monofun-cfun-fun*, *rule monofun-cfun-arg*)
 apply (*erule deflation.below-ID*)
done

lemma *ep-pair-fmap*:
  $ep\text{-}pair\ e\ p \Longrightarrow ep\text{-}pair\ (fmap\cdot e)\ (fmap\cdot p)$
 apply (*rule ep-pair.intro*)
  apply (*simp add*: *fmap-fmap ep-pair.e-inverse*)
 apply (*simp add*: *fmap-fmap*)
 apply (*rule-tac* $y{=}fmap\cdot ID\cdot y$ **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 $f :: {'a} \to {'b}$
assumes $f\cdot\bot = \bot$ **shows** $fmap\cdot f\cdot\bot = (\bot::{'b}\cdot{'f}::functor)$
proof (*rule bottomI*)
have $fmap\cdot f\cdot(\bot::{'a}\cdot{'f}) \sqsubseteq fmap\cdot f\cdot(fmap\cdot\bot\cdot(\bot::{'b}\cdot{'f}))$
by (*simp add*: *monofun-cfun*)
also have ... $= fmap\cdot(\Lambda\ x.\ f\cdot(\bot\cdot x))\cdot(\bot::{'b}\cdot{'f})$
by (*simp add*: *fmap-fmap*)
also have ... $\sqsubseteq fmap\cdot ID\cdot\bot$
by (*simp add*: *monofun-cfun assms del*: *fmap-ID*)
also have ... $= \bot$
by *simp*
finally show $fmap\cdot f\cdot\bot \sqsubseteq (\bot::{'b}\cdot{'f}::functor)$ **.**
qed

3.4 Proving that $fmap\cdot coerce = coerce$

lemma *fmapU-cast-eq*:
$fmapU\cdot(cast\cdot A) =$
$PRJ(udom\cdot{'f})\ oo\ cast\cdot(TC({'f}::functor)\cdot A)\ oo\ emb$
by (*subst cast-TC*, *rule cfun-eqI*, *simp*)

lemma *fmapU-cast-DEFL*:
$fmapU\cdot(cast\cdot DEFL({'a})) =$
$PRJ(udom\cdot{'f})\ oo\ cast\cdot DEFL({'a}\cdot{'f}::functor)\ oo\ emb$
by (*simp add*: *fmapU-cast-eq DEFL-app*)

lemma *coerce-functor*: $COERCE({'a}\cdot{'f},\ {'b}\cdot{'f}::functor) = fmap\cdot coerce$
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 $m :: {'a}\cdot{'f}::functor$
shows $fmapU\cdot(cast\cdot DEFL({'a}))\cdot(COERCE({'a}\cdot{'f},\ udom\cdot{'f})\cdot m) =$
$COERCE({'a}\cdot{'f},\ udom\cdot{'f})\cdot m$

by (*simp add*: *coerce-functor cast-DEFL fmapU-eq-fmap fmap-fmap eta-cfun*)

lemma *coerce-fmap* [*coerce-simp*]:
  fixes *xs* :: $'a{\cdot}'f$::*functor* **and** *f* :: $'a \to 'b$
  shows $COERCE('b{\cdot}'f,\ 'c{\cdot}'f){\cdot}(fmap{\cdot}f{\cdot}xs) = fmap{\cdot}(\Lambda\ x.\ COERCE('b,'c){\cdot}(f{\cdot}x)){\cdot}xs$
by (*simp add*: *coerce-functor fmap-fmap*)

lemma *fmap-coerce* [*coerce-simp*]:
  fixes *xs* :: $'a{\cdot}'f$::*functor* **and** *f* :: $'b \to 'c$
  shows $fmap{\cdot}f{\cdot}(COERCE('a{\cdot}'f,\ 'b{\cdot}'f){\cdot}xs) = fmap{\cdot}(\Lambda\ x.\ f{\cdot}(COERCE('a,'b){\cdot}x)){\cdot}xs$
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*:
    $\bigwedge f\ xs.\ fmapU{\cdot}f{\cdot}xs = bindU{\cdot}xs{\cdot}(\Lambda\ x.\ returnU{\cdot}(f{\cdot}x))$
  assumes *bindU-returnU*:
    $\bigwedge f\ x.\ bindU{\cdot}(returnU{\cdot}x){\cdot}f = f{\cdot}x$
  assumes *bindU-bindU*:
    $\bigwedge xs\ f\ g.\ bindU{\cdot}(bindU{\cdot}xs{\cdot}f){\cdot}g = bindU{\cdot}xs{\cdot}(\Lambda\ x.\ bindU{\cdot}(f{\cdot}x){\cdot}g)$

instance *monad* ⊆ *functor*
proof
  fix *f g* :: *udom* → *udom* **and** *xs* :: *udom*·*'a*
  show $fmapU{\cdot}f{\cdot}(fmapU{\cdot}g{\cdot}xs) = fmapU{\cdot}(\Lambda\ x.\ f{\cdot}(g{\cdot}x)){\cdot}xs$
    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* ⇒ (*'a* → *'b*·*'m*) ⇒ *'b*·*'m* (**infixl** ⪢= *55*)
  where *m* ⪢= *f* ≡ *bind*·*m*·*f*

4.2 Naturality of bind and return

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

lemma *fmapU-returnU* [*coerce-simp*]:
  $fmapU{\cdot}f{\cdot}(returnU{\cdot}x) = returnU{\cdot}(f{\cdot}x)$
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** $f :: 'a \to 'b$
shows $fmap \cdot f \cdot xs = xs \ggg= (\Lambda\ x.\ return \cdot (f \cdot x))$
unfolding *bind-def return-def fmap-def*
by (*simp add*: *coerce-simp fmapU-eq-bindU bindU-returnU*)

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

lemma *bind-bind*:
fixes $m :: 'a \cdot 'm::monad$
shows $((m \ggg= f) \ggg= g) = (m \ggg= (\Lambda\ x.\ f \cdot x \ggg= g))$
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*]: $(m \ggg= return) = m$
apply (*subgoal-tac* $fmap \cdot ID \cdot m = m$)
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*: $bind \cdot (fmap \cdot f \cdot xs) \cdot k = bind \cdot xs \cdot (\Lambda\ x.\ k \cdot (f \cdot x))$
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 $k \cdot \bot = \bot$ **shows** $\bot \ggg= k = \bot$

proof −
  have $\bot \gg\!= k \sqsubseteq return\cdot\bot \gg\!= k$
    by (*intro monofun-cfun below-refl minimal*)
  thus $\bot \gg\!= k = \bot$
    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 $join :: ('a\cdot'm)\cdot'm \rightarrow 'a\cdot'm{::}monad$
  where $join \equiv \Lambda\ m.\ m \gg\!= (\Lambda\ x.\ x)$

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

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

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

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

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

4.6 Equivalence of monad laws and fmap/join laws

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

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

lemma $((m \gg\!= f) \gg\!= g) = (m \gg\!= (\Lambda\ x.\ f\cdot x \gg\!= g))$
 apply (*simp only*: *bind-def2*)
 apply (*subgoal-tac* $join\cdot(fmap\cdot g\cdot(join\cdot(fmap\cdot f\cdot m))) =$
    $join\cdot(fmap\cdot join\cdot(fmap\cdot(fmap\cdot g)\cdot(fmap\cdot f\cdot m))))$
  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*]:
  $COERCE('a \cdot 'm,'b \cdot 'm::monad) \cdot (return \cdot x) = return \cdot (COERCE('a,'b) \cdot x)$
by (*simp add*: *coerce-functor fmap-return*)

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

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

end

5 Monad-Zero Class

theory *Monad-Zero*
imports *Monad*
begin

class *zeroU* = *tycon* +
  fixes *zeroU* :: $udom \cdot 'a::tycon$

class *functor-zero* = *zeroU* + *functor* +
  assumes *fmapU-zeroU* [*coerce-simp*]:
    $fmapU \cdot f \cdot zeroU = zeroU$

class *monad-zero* = *zeroU* + *monad* +
  assumes *bindU-zeroU*:
    $bindU \cdot zeroU \cdot f = zeroU$

instance *monad-zero* $\subseteq$ *functor-zero*
proof
  fix f **show** $fmapU \cdot f \cdot zeroU = (zeroU :: udom \cdot 'a)$
    unfolding *fmapU-eq-bindU*
    by (*rule bindU-zeroU*)
qed

definition *fzero* :: $'a \cdot 'f::functor\text{-}zero$
  where $fzero = coerce \cdot (zeroU :: udom \cdot 'f)$

lemma *fmap-fzero*:

$fmap{\cdot}f{\cdot}(fzero :: 'a{\cdot}'f::functor\text{-}zero) = (fzero :: 'b{\cdot}'f)$
unfolding *fmap-def fzero-def*
by (*simp add*: *coerce-simp*)

abbreviation *mzero* :: $'a{\cdot}'m::monad\text{-}zero$
where *mzero* ≡ *fzero*

lemmas *mzero-def* = *fzero-def* [**where** $'f='m::monad\text{-}zero$] **for** *f*
lemmas *fmap-mzero* = *fmap-fzero* [**where** $'f='m::monad\text{-}zero$] **for** *f*

lemma *bindU-eq-bind*: *bindU* = *bind*
unfolding *bind-def* **by** *simp*

lemma *bind-mzero*:
$bind{\cdot}(fzero :: 'a{\cdot}'m::monad\text{-}zero){\cdot}k = (mzero :: 'b{\cdot}'m)$
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* :: $udom{\cdot}'a \to udom{\cdot}'a \to udom{\cdot}'a::tycon$

class *functor-plus* = *plusU* + *functor* +
assumes *fmapU-plusU* [*coerce-simp*]:
$fmapU{\cdot}f{\cdot}(plusU{\cdot}a{\cdot}b) = plusU{\cdot}(fmapU{\cdot}f{\cdot}a){\cdot}(fmapU{\cdot}f{\cdot}b)$
assumes *plusU-assoc*:
$plusU{\cdot}(plusU{\cdot}a{\cdot}b){\cdot}c = plusU{\cdot}a{\cdot}(plusU{\cdot}b{\cdot}c)$

class *monad-plus* = *plusU* + *monad* +
assumes *bindU-plusU*:
$bindU{\cdot}(plusU{\cdot}xs{\cdot}ys){\cdot}k = plusU{\cdot}(bindU{\cdot}xs{\cdot}k){\cdot}(bindU{\cdot}ys{\cdot}k)$
assumes *plusU-assoc'*:
$plusU{\cdot}(plusU{\cdot}a{\cdot}b){\cdot}c = plusU{\cdot}a{\cdot}(plusU{\cdot}b{\cdot}c)$

instance *monad-plus* ⊆ *functor-plus*
by *standard* (*simp-all only*: *fmapU-eq-bindU bindU-plusU plusU-assoc'*)

definition *fplus* :: $'a{\cdot}'f::functor\text{-}plus \to 'a{\cdot}'f \to 'a{\cdot}'f$
where $fplus = coerce{\cdot}(plusU :: udom{\cdot}'f \to \text{-})$

lemma *fmap-fplus*:
  fixes f :: $'a \to 'b$ **and** a b :: $'a \cdot 'f$::*functor-plus*
  shows $fmap \cdot f \cdot (fplus \cdot a \cdot b) = fplus \cdot (fmap \cdot f \cdot a) \cdot (fmap \cdot f \cdot b)$
unfolding *fmap-def fplus-def*
by (*simp add*: *coerce-simp*)

lemma *fplus-assoc*:
  fixes a b c :: $'a \cdot 'f$::*functor-plus*
  shows $fplus \cdot (fplus \cdot a \cdot b) \cdot c = fplus \cdot a \cdot (fplus \cdot b \cdot c)$
unfolding *fplus-def*
by (*simp add*: *coerce-simp plusU-assoc*)

abbreviation *mplus* :: $'a \cdot 'm$::*monad-plus* $\to$ $'a \cdot 'm \to 'a \cdot 'm$
  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 :: $'a \cdot 'm$::*monad-plus*
  shows $bind \cdot (mplus \cdot a \cdot b) \cdot k = mplus \cdot (bind \cdot a \cdot k) \cdot (bind \cdot b \cdot k)$
unfolding *bind-def mplus-def*
by (*simp add*: *coerce-simp bindU-plusU*)

lemma *join-mplus*:
  fixes *xss yss* :: $('a \cdot 'm) \cdot 'm$::*monad-plus*
  shows $join \cdot (mplus \cdot xss \cdot yss) = mplus \cdot (join \cdot xss) \cdot (join \cdot yss)$
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*:
    $plusU \cdot zeroU \cdot m = m$
  assumes *plusU-zeroU-right*:
    $plusU \cdot m \cdot zeroU = m$

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 $'a \cdot llist = LNil \mid LCons$ (**lazy** $'a$) (**lazy** $'a \cdot llist$)

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*]: $coerce \cdot (llist\text{-}abs \cdot x) = llist\text{-}abs \cdot (coerce \cdot x)$
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*]: $coerce \cdot LNil = LNil$
unfolding *LNil-def* **by** *simp*

lemma *coerce-LCons* [*simp*]: $coerce \cdot (LCons \cdot x \cdot xs) = LCons \cdot (coerce \cdot x) \cdot (coerce \cdot xs)$
unfolding *LCons-def* **by** *simp*

lemma *fmapU-llist-simps* [*simp*]:
  $fmapU \cdot f \cdot (\bot :: udom \cdot llist) = \bot$
  $fmapU \cdot f \cdot LNil = LNil$
  $fmapU \cdot f \cdot (LCons \cdot x \cdot xs) = LCons \cdot (f \cdot x) \cdot (fmapU \cdot f \cdot xs)$
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** $xs :: udom \cdot llist$
  show $fmapU \cdot f \cdot (fmapU \cdot g \cdot xs) = fmapU \cdot (\Lambda\ x.\ f \cdot (g \cdot x)) \cdot xs$
    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·*(Λ *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: $\mathit{llist.induct}$)
($\mathit{simp\text{-}all}$ add: $\mathit{plusU\text{-}llist\text{-}assoc}$)
show $\mathit{plusU}\cdot(\mathit{plusU}\cdot\mathit{xs}\cdot\mathit{ys})\cdot\mathit{zs} = \mathit{plusU}\cdot\mathit{xs}\cdot(\mathit{plusU}\cdot\mathit{ys}\cdot\mathit{zs})$
by (rule $\mathit{plusU\text{-}llist\text{-}assoc}$)
show $\mathit{bindU}\cdot\mathit{zeroU}\cdot k = \mathit{zeroU}$
by (simp add: $\mathit{zeroU\text{-}llist\text{-}def}$)
show $\mathit{fmapU}\cdot f\cdot(\mathit{plusU}\cdot\mathit{xs}\cdot\mathit{ys}) = \mathit{plusU}\cdot(\mathit{fmapU}\cdot f\cdot\mathit{xs})\cdot(\mathit{fmapU}\cdot f\cdot\mathit{ys})$
by (induct xs rule: $\mathit{llist.induct}$) $\mathit{simp\text{-}all}$
show $\mathit{fmapU}\cdot f\cdot\mathit{zeroU} = (\mathit{zeroU} :: \mathit{udom}\cdot\mathit{llist})$
by (simp add: $\mathit{zeroU\text{-}llist\text{-}def}$)
show $\mathit{plusU}\cdot\mathit{zeroU}\cdot\mathit{xs} = \mathit{xs}$
by (simp add: $\mathit{zeroU\text{-}llist\text{-}def}$)
show $\mathit{plusU}\cdot\mathit{xs}\cdot\mathit{zeroU} = \mathit{xs}$
by (simp add: $\mathit{zeroU\text{-}llist\text{-}def}$ $\mathit{plusU\text{-}LNil\text{-}right}$)
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 $\mathit{fmap\text{-}llist\text{-}simps}$ [simp]:
$\mathit{fmap}\cdot f\cdot(\bot{::}'a\cdot\mathit{llist}) = \bot$
$\mathit{fmap}\cdot f\cdot\mathit{LNil} = \mathit{LNil}$
$\mathit{fmap}\cdot f\cdot(\mathit{LCons}\cdot x\cdot\mathit{xs}) = \mathit{LCons}\cdot(f\cdot x)\cdot(\mathit{fmap}\cdot f\cdot\mathit{xs})$
unfolding $\mathit{fmap\text{-}def}$ **by** $\mathit{simp\text{-}all}$

lemma $\mathit{mplus\text{-}llist\text{-}simps}$ [simp]:
$\mathit{mplus}\cdot(\bot{::}'a\cdot\mathit{llist})\cdot\mathit{ys} = \bot$
$\mathit{mplus}\cdot\mathit{LNil}\cdot\mathit{ys} = \mathit{ys}$
$\mathit{mplus}\cdot(\mathit{LCons}\cdot x\cdot\mathit{xs})\cdot\mathit{ys} = \mathit{LCons}\cdot x\cdot(\mathit{mplus}\cdot\mathit{xs}\cdot\mathit{ys})$
unfolding $\mathit{mplus\text{-}def}$ **by** $\mathit{simp\text{-}all}$

lemma $\mathit{bind\text{-}llist\text{-}simps}$ [simp]:
$\mathit{bind}\cdot(\bot{::}'a\cdot\mathit{llist})\cdot f = \bot$
$\mathit{bind}\cdot\mathit{LNil}\cdot f = \mathit{LNil}$
$\mathit{bind}\cdot(\mathit{LCons}\cdot x\cdot\mathit{xs})\cdot f = \mathit{mplus}\cdot(f\cdot x)\cdot(\mathit{bind}\cdot\mathit{xs}\cdot f)$
unfolding $\mathit{bind\text{-}def}$ $\mathit{mplus\text{-}def}$
by ($\mathit{simp\text{-}all}$ add: $\mathit{coerce\text{-}simp}$)

lemma $\mathit{return\text{-}llist\text{-}def}$:
$\mathit{return} = (\Lambda\ x.\ \mathit{LCons}\cdot x\cdot\mathit{LNil})$

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*]:
  $join{\cdot}(\bot{::}'a{\cdot}llist{\cdot}llist) = \bot$
  $join{\cdot}LNil = LNil$
  $join{\cdot}(LCons{\cdot}xs{\cdot}xss) = mplus{\cdot}xs{\cdot}(join{\cdot}xss)$
unfolding *join-def* **by** *simp-all*

end

9 Maybe monad

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

9.1 Type definition

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

lemma *coerce-maybe-abs* [*simp*]: $coerce{\cdot}(maybe\text{-}abs{\cdot}x) = maybe\text{-}abs{\cdot}(coerce{\cdot}x)$
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*]: $coerce{\cdot}Nothing = Nothing$
unfolding *Nothing-def* **by** *simp*

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

lemma *fmapU-maybe-simps* [*simp*]:
  $fmapU{\cdot}f{\cdot}(\bot{::}udom{\cdot}maybe) = \bot$
  $fmapU{\cdot}f{\cdot}Nothing = Nothing$
  $fmapU{\cdot}f{\cdot}(Just{\cdot}x) = Just{\cdot}(f{\cdot}x)$
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·(Λ 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·(Λ x. bindU·(h·x)·k)*

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

end

9.3 Transfer properties to polymorphic versions

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

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

lemma *fplus-Nothing-right* [*simp*]:
$fplus \cdot m \cdot Nothing = m$
by (*simp add*: *fplus-def plusU-Nothing-right*)

lemma *bind-maybe-simps* [*simp*]:
$bind \cdot (\bot :: 'a \cdot maybe) \cdot f = \bot$
$bind \cdot Nothing \cdot f = Nothing$
$bind \cdot (Just \cdot x) \cdot f = f \cdot x$
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 cfcomp1 eta-cfun*)

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

lemma *join-maybe-simps* [*simp*]:
  $join\cdot(\bot::'a\cdot maybe\cdot maybe) = \bot$
  $join\cdot Nothing = Nothing$
  $join\cdot(Just\cdot xs) = xs$
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*:
  $[\![a = Just\cdot x;\ b = \bot;\ k\cdot x = Nothing]\!]$
    $\Longrightarrow fplus\cdot a\cdot b \ggg= k \neq fplus\cdot(a \ggg= k)\cdot(b \ggg= k)$
by *simp*

lemma *maybe-counterexample2*:
  $[\![a = Just\cdot x;\ b = Just\cdot y;\ k\cdot x = Nothing;\ k\cdot y = Just\cdot z]\!]$
    $\Longrightarrow fplus\cdot a\cdot b \ggg= k \neq fplus\cdot(a \ggg= k)\cdot(b \ggg= k)$
by *simp*

end

10 Error monad

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

10.1 Type definition

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

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

lemma *coerce-Ok* [*simp*]: $coerce\cdot(Ok\cdot m) = Ok\cdot(coerce\cdot m)$
unfolding *Ok-def* **by** *simp*

lemma *fmapU-error-simps* [*simp*]:
  $fmapU\cdot f\cdot(\bot::udom\cdot'a\ error) = \bot$
  $fmapU\cdot f\cdot(Err\cdot e) = Err\cdot e$
  $fmapU\cdot f\cdot(Ok\cdot x) = Ok\cdot(f\cdot x)$
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* :: *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 :: $udom\cdot'a$ $error$
show $plusU\cdot(plusU\cdot a\cdot b)\cdot c = plusU\cdot a\cdot(plusU\cdot b\cdot c)$
by (*induct a rule*: *error.induct*) *simp-all*
qed

end

10.3 Transfer properties to polymorphic versions

lemma *fmap-error-simps* [*simp*]:
$fmap\cdot f\cdot(\bot::'a\cdot'e\ error) = \bot$
$fmap\cdot f\cdot(Err\cdot e :: 'a\cdot'e\ error) = Err\cdot e$
$fmap\cdot f\cdot(Ok\cdot x :: 'a\cdot'e\ error) = Ok\cdot(f\cdot x)$
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*]:
$bind\cdot(\bot :: 'a\cdot'e\ error)\cdot f = \bot$
$bind\cdot(Err\cdot e :: 'a\cdot'e\ error)\cdot f = Err\cdot e$
$bind\cdot(Ok\cdot x :: 'a\cdot'e\ error)\cdot f = f\cdot x$
unfolding *bind-def*
by (*simp-all add*: *coerce-simp*)

lemma *join-error-simps* [*simp*]:
$join\cdot\bot = (\bot :: 'a\cdot'e\ error)$
$join\cdot(Err\cdot e) = Err\cdot e$
$join\cdot(Ok\cdot x) = x$
unfolding *join-def* **by** *simp-all*

lemma *fplus-error-simps* [*simp*]:
$fplus\cdot\bot\cdot r = (\bot :: 'a\cdot'e\ error)$
$fplus\cdot(Err\cdot e)\cdot r = r$
$fplus\cdot(Ok\cdot x)\cdot r = Ok\cdot x$
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 *mempty* :: *'a*
  fixes *mappend* :: *'a* → *'a* → *'a*
  assumes *mempty-left*: ⋀*ys*. *mappend*·*mempty*·*ys* = *ys*
  assumes *mempty-right*: ⋀*xs*. *mappend*·*xs*·*mempty* = *xs*
  assumes *mappend-assoc*:
    ⋀*xs ys zs*. *mappend*·(*mappend*·*xs*·*ys*)·*zs* = *mappend*·*xs*·(*mappend*·*ys*·*zs*)

11.2 Writer monad type

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

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

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

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

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

tycondef *'a*·*'w writer* = *Writer* (**lazy** *'w*) (**lazy** *'a*)

lemma *coerce-writer-abs* [*simp*]: *coerce*·(*writer-abs*·*x*) = *writer-abs*·(*coerce*·*x*)
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*·(⊥::*udom*·*'w writer*) = ⊥
  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* :: *udom* → *udom* **and** *xs* :: *udom*·*'a writer*

show *fmapU·f·(fmapU·g·xs) = fmapU·(Λ x. f·(g·x))·xs*
by (*induct xs rule*: *writer.induct*) *simp-all*
qed

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

fixrec *bindU-writer* ::
udom·'a writer → (udom → udom·'a writer) → udom·'a writer
where *bindU-writer·(Writer·w·x)·f =*
(*case f·x of Writer·w'·y ⇒ Writer·(mappend·w·w')·y*)

lemma *bindU-writer-strict* [*simp*]: *bindU·⊥·k = (⊥::udom·'a writer)*
by *fixrec-simp*

definition
returnU = Writer·mempty

instance proof
fix *f* :: *udom → udom* **and** *m* :: *udom·'a writer*
show *fmapU·f·m = bindU·m·(Λ x. returnU·(f·x))*
by (*induct m rule*: *writer.induct*)
(*simp-all add*: *returnU-writer-def mempty-right*)
next
fix *f* :: *udom → udom·'a writer* **and** *x* :: *udom*
show *bindU·(returnU·x)·f = f·x*
by (*cases f·x rule*: *writer.exhaust*)
(*simp-all add*: *returnU-writer-def mempty-left*)
next
fix *m* :: *udom·'a writer* **and** *f g* :: *udom → udom·'a writer*
show *bindU·(bindU·m·f)·g = bindU·m·(Λ x. bindU·(f·x)·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*]:
fmap·f·(⊥::'a·'w writer) = ⊥
fmap·f·(Writer·w·x :: 'a·'w writer) = Writer·w·(f·x)
unfolding *fmap-def* [**where** *'f='w writer*]
by (*simp-all add*: *coerce-simp*)

lemma *return-writer-def*: *return = Writer·mempty*

unfolding *return-def returnU-writer-def*
by (*simp add*: *coerce-simp eta-cfun*)

lemma *bind-writer-simps* [*simp*]:
  $bind\cdot(\bot :: {'a}\cdot{'w}::monoid\ writer)\cdot f = \bot$
  $bind\cdot(Writer\cdot w\cdot x :: {'a}\cdot{'w}::monoid\ writer)\cdot k =$
    $(case\ k\cdot x\ of\ Writer\cdot w'\cdot y \Rightarrow Writer\cdot(mappend\cdot w\cdot w')\cdot y)$
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*]:
  $join\cdot\bot = (\bot :: {'a}\cdot{'w}::monoid\ writer)$
  $join\cdot(Writer\cdot w\cdot(Writer\cdot w'\cdot x)) = Writer\cdot(mappend\cdot w\cdot w')\cdot x$
unfolding *join-def* **by** *simp-all*

11.5 Extra operations

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

end

12 Binary tree monad

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

12.1 Type definition

tycondef $'a\cdot btree =$
  Leaf (**lazy** $'a$) | *Node* (**lazy** $'a\cdot btree$) (**lazy** $'a\cdot btree$)

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

lemma *coerce-Node* [*simp*]: $coerce\cdot(Node\cdot xs\cdot ys) = Node\cdot(coerce\cdot xs)\cdot(coerce\cdot ys)$
unfolding *Node-def* **by** *simp*

lemma *fmapU-btree-simps* [*simp*]:
  $fmapU\cdot f\cdot(\bot::udom\cdot btree) = \bot$

$fmapU \cdot f \cdot (Leaf \cdot x) = Leaf \cdot (f \cdot x)$
$fmapU \cdot f \cdot (Node \cdot xs \cdot ys) = Node \cdot (fmapU \cdot f \cdot xs) \cdot (fmapU \cdot f \cdot ys)$
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* :: $udom \cdot btree \to (udom \to udom \cdot btree) \to udom \cdot btree$
where $bindU\text{-}btree \cdot (Leaf \cdot x) \cdot k = k \cdot x$
| $bindU\text{-}btree \cdot (Node \cdot xs \cdot ys) \cdot k =$
$Node \cdot (bindU\text{-}btree \cdot xs \cdot k) \cdot (bindU\text{-}btree \cdot ys \cdot k)$

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

instance proof
fix $x :: udom$
fix $f :: udom \to udom$
fix $h\ k :: udom \to udom \cdot btree$
fix $xs :: udom \cdot btree$
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*]:
$fmap \cdot f \cdot (\bot :: 'a \cdot btree) = \bot$
$fmap \cdot f \cdot (Leaf \cdot x) = Leaf \cdot (f \cdot x)$
$fmap \cdot f \cdot (Node \cdot xs \cdot ys) = Node \cdot (fmap \cdot f \cdot xs) \cdot (fmap \cdot f \cdot ys)$

unfolding *fmap-def* **by** *simp-all*

lemma *bind-btree-simps* [*simp*]:
  $bind{\cdot}(\bot{::}'a{\cdot}btree){\cdot}k = \bot$
  $bind{\cdot}(Leaf{\cdot}x){\cdot}k = k{\cdot}x$
  $bind{\cdot}(Node{\cdot}xs{\cdot}ys){\cdot}k = Node{\cdot}(bind{\cdot}xs{\cdot}k){\cdot}(bind{\cdot}ys{\cdot}k)$
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*]:
  $join{\cdot}(\bot{::}'a{\cdot}btree{\cdot}btree) = \bot$
  $join{\cdot}(Leaf{\cdot}xs) = xs$
  $join{\cdot}(Node{\cdot}xss{\cdot}yss) = Node{\cdot}(join{\cdot}xss){\cdot}(join{\cdot}yss)$
unfolding *join-def* **by** *simp-all*

end

13 Lift monad

theory *Lift-Monad*
imports *Monad*
begin

13.1 Type definition

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

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

lemma *fmapU-lifted-simps* [*simp*]:
  $fmapU{\cdot}f{\cdot}(\bot{::}udom{\cdot}lifted) = \bot$
  $fmapU{\cdot}f{\cdot}(Lifted{\cdot}x) = Lifted{\cdot}(f{\cdot}x)$
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·(⊥::'*a·lifted·lifted*) = ⊥

$join \cdot (Lifted \cdot xs) = xs$
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 $'a \cdot ('f::functor)\ resT =$
 $Done$ (**lazy** $'a$) | $More$ (**lazy** $('a \cdot 'f\ resT) \cdot 'f$)

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

lemma *coerce-More* [*simp*]: $coerce \cdot (More \cdot m) = More \cdot (coerce \cdot m)$
unfolding *More-def* **by** *simp*

lemma *resT-induct* [*case-names adm bottom Done More*]:
 fixes $P :: 'a \cdot 'f::functor\ resT \Rightarrow bool$
 assumes *adm*: $adm\ P$
 assumes *bottom*: $P\ \bot$
 assumes *Done*: $\bigwedge x.\ P\ (Done \cdot x)$
 assumes *More*: $\bigwedge m\ f.\ (\bigwedge (r::'a \cdot 'f\ resT).\ P\ (f \cdot r)) \Longrightarrow P\ (More \cdot (fmap \cdot f \cdot m))$
 shows $P\ r$
proof (*induct r rule*: *resT.take-induct* [*OF adm*])
 fix n **show** $P\ (resT\text{-}take\ n \cdot r)$
  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 ($\mathit{induct}\ \mathit{xs}\ \mathit{rule}$: $\mathit{resT\text{-}induct}$)
($\mathit{simp\text{-}all}\ \mathit{add}$: $\mathit{fmap\text{-}fmap}$)
qed

end

14.3 Transfer properties to polymorphic versions

lemma *fmap-resT-simps* [*simp*]:
$\mathit{fmap}\cdot f\cdot(\bot::'a\cdot'f::\mathit{functor}\ \mathit{resT}) = \bot$
$\mathit{fmap}\cdot f\cdot(\mathit{Done}\cdot x :: 'a\cdot'f::\mathit{functor}\ \mathit{resT}) = \mathit{Done}\cdot(f\cdot x)$
$\mathit{fmap}\cdot f\cdot(\mathit{More}\cdot m :: 'a\cdot'f::\mathit{functor}\ \mathit{resT}) = \mathit{More}\cdot(\mathit{fmap}\cdot(\mathit{fmap}\cdot f)\cdot m)$
unfolding *fmap-def* [**where** $'f='f\ \mathit{resT}$]
by (*simp-all add*: *coerce-simp*)

lemma *return-resT-def*: $\mathit{return} = \mathit{Done}$
unfolding *return-def returnU-resT-def*
by (*simp add*: *coerce-simp eta-cfun*)

lemma *bind-resT-simps* [*simp*]:
$\mathit{bind}\cdot(\bot :: 'a\cdot'f::\mathit{functor}\ \mathit{resT})\cdot f = \bot$
$\mathit{bind}\cdot(\mathit{Done}\cdot x :: 'a\cdot'f::\mathit{functor}\ \mathit{resT})\cdot f = f\cdot x$
$\mathit{bind}\cdot(\mathit{More}\cdot m :: 'a\cdot'f::\mathit{functor}\ \mathit{resT})\cdot f = \mathit{More}\cdot(\mathit{fmap}\cdot(\Lambda\ r.\ \mathit{bind}\cdot r\cdot f)\cdot m)$
unfolding *bind-def*
by (*simp-all add*: *coerce-simp*)

lemma *join-resT-simps* [*simp*]:
$\mathit{join}\cdot\bot = (\bot :: 'a\cdot'f::\mathit{functor}\ \mathit{resT})$
$\mathit{join}\cdot(\mathit{Done}\cdot x) = x$
$\mathit{join}\cdot(\mathit{More}\cdot m) = \mathit{More}\cdot(\mathit{fmap}\cdot\mathit{join}\cdot m)$
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* ::
$('a \to 'b \to 'c) \to 'a\cdot('m::\mathit{functor\text{-}plus})\ \mathit{resT} \to 'b\cdot'm\ \mathit{resT} \to 'c\cdot'm\ \mathit{resT}$
where *zipRT-Done-Done*:
$\mathit{zipRT}\cdot f\cdot(\mathit{Done}\cdot x)\cdot(\mathit{Done}\cdot y) = \mathit{Done}\cdot(f\cdot x\cdot y)$
| *zipRT-Done-More*:
$\mathit{zipRT}\cdot f\cdot(\mathit{Done}\cdot x)\cdot(\mathit{More}\cdot b) =$
$\mathit{More}\cdot(\mathit{fmap}\cdot(\Lambda\ r.\ \mathit{zipRT}\cdot f\cdot(\mathit{Done}\cdot x)\cdot r)\cdot b)$
| *zipRT-More-Done*:

$zipRT{\cdot}f{\cdot}(More{\cdot}a){\cdot}(Done{\cdot}y) =$
$More{\cdot}(fmap{\cdot}(\Lambda\ r.\ zipRT{\cdot}f{\cdot}r{\cdot}(Done{\cdot}y)){\cdot}a)$
| *zipRT-More-More*:
$zipRT{\cdot}f{\cdot}(More{\cdot}a){\cdot}(More{\cdot}b) =$
$More{\cdot}(fplus{\cdot}(fmap{\cdot}(\Lambda\ r.\ zipRT{\cdot}f{\cdot}(More{\cdot}a){\cdot}r){\cdot}b)$
$\cdot(fmap{\cdot}(\Lambda\ r.\ zipRT{\cdot}f{\cdot}r{\cdot}(More{\cdot}b)){\cdot}a))$

lemma *zipRT-strict1* [*simp*]: $zipRT{\cdot}f{\cdot}\bot{\cdot}r = \bot$
by *fixrec-simp*

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

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

Proofs that *zipRT* satisfies the applicative functor laws:

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

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

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

The associativity rule is the hard one!

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

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

lemma *stateT-cases* [*case-names StateT*]:
  obtains *k* **where** $y = StateT \cdot k$
proof
  show $y = StateT \cdot (runStateT \cdot y)$
    by (*cases y*, *simp-all*)
qed

lemma *stateT-induct* [*case-names StateT*]:
  fixes $P :: 'a \cdot ('f{::}functor, 's)\ stateT \Rightarrow bool$

assumes $\bigwedge k.\ P\ (StateT{\cdot}k)$
shows $P\ y$
by (*cases y rule*: *stateT-cases*, *simp add*: *assms*)

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

lemma *runStateT-coerce* [*simp*]:
$runStateT{\cdot}(coerce{\cdot}k){\cdot}s = coerce{\cdot}(runStateT{\cdot}k{\cdot}s)$
by (*induct k rule*: *stateT-induct*, *simp*)

15.1 Functor class instance

lemma *fmapU-StateT* [*simp*]:
$fmapU{\cdot}f{\cdot}(StateT{\cdot}k) =$
$StateT{\cdot}(\Lambda\ s.\ fmap{\cdot}(\Lambda(x,\ s').\ (f{\cdot}x,\ s')){\cdot}(k{\cdot}s))$
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*]:
$runStateT{\cdot}(fmapU{\cdot}f{\cdot}m){\cdot}s =$
$fmap{\cdot}(\Lambda(x,\ s').\ (f{\cdot}x,\ s')){\cdot}(runStateT{\cdot}m{\cdot}s)$
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*]:
$bindU\cdot(StateT\cdot f)\cdot k =$
$StateT\cdot(\Lambda\ s.\ f\cdot s \ggg= (\Lambda\ (x,\ s').\ runStateT\cdot(k\cdot x)\cdot s'))$
unfolding *bindU-stateT-def* **by** *simp*

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

instance proof
fix $f :: udom \to udom$ **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 $f :: udom \to udom\cdot('a,'b)\ stateT$ **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** $f\ g :: udom \to udom\cdot('a,'b)\ stateT$
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 $k :: udom \to udom\cdot('a,'b)\ stateT$
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 :: $udom \to udom\cdot('a,\ 'b)\ stateT$
  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*]:
  $fmap \cdot f \cdot (StateT \cdot m :: 'a \cdot ('f::functor,'s)\ stateT) =$
    $StateT \cdot (\Lambda\ s.\ fmap \cdot (\Lambda\ (x,\ s').\ (f \cdot x,\ s')) \cdot (m \cdot s))$
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*:
  $(return :: \text{-} \rightarrow 'a \cdot ('m::monad,\ 's)\ stateT) =$
    $(\Lambda\ x.\ StateT \cdot (\Lambda\ s.\ return \cdot (x,\ s)))$
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*]:
  $bind \cdot (StateT \cdot m :: 'a \cdot ('m::monad,'s)\ stateT) \cdot k =$
    $StateT \cdot (\Lambda\ s.\ m \cdot s \gg= (\Lambda\ (x,\ s').\ runStateT \cdot (k \cdot x) \cdot s'))$
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 $'a\cdot('f{::}functor,'e{::}domain)$ *errorT* =
  ErrorT (*runErrorT* :: $('a\cdot'e\ error)\cdot'f$)

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 $P :: {'a}\cdot('f{::}functor,'e)\ errorT \Rightarrow bool$
  assumes $\bigwedge k.\ P\ (ErrorT\cdot k)$
  shows *P y*
by (*cases y rule*: *errorT-cases*, *simp add*: *assms*)

lemma *errorT-eq-iff*:
  $a = b \longleftrightarrow runErrorT\cdot a = runErrorT\cdot b$
apply (*cases a rule*: *errorT-cases*)
apply (*cases b rule*: *errorT-cases*)
apply *simp*
done

lemma *errorT-eqI*:
  $runErrorT\cdot a = runErrorT\cdot b \Longrightarrow a = b$
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·(Λ 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 → 'b* **and** *m* :: *'a·'e error·('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 → '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·(⊥::'a·('m::monad,'e) errorT) = ⊥*
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·('m,'e) errorT* contains values that break the monad laws. However, it turns out that such values are inaccessible: The monad laws are satisfied by all values constructible from the abstract operations.

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

definition *unitET* :: $'a \to 'a\cdot('m::monad,'e)$ *errorT*
  where *unitET* = $(\Lambda\ x.\ ErrorT\cdot(return\cdot(Ok\cdot x)))$

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

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

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

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

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

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

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

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

lemma *runErrorT-throwET* [*simp*]:
  $runErrorT\cdot(throwET\cdot e) = return\cdot(Err\cdot e)$
unfolding *throwET-def* **by** *simp*

lemma *runErrorT-catchET* [*simp*]:
  $runErrorT\cdot(catchET\cdot m\cdot h) =$
    $bind\cdot(runErrorT\cdot m)\cdot(\Lambda\ n.\ case\ n\ of$

$Err \cdot e \Rightarrow runErrorT \cdot (h \cdot e) \mid Ok \cdot x \Rightarrow return \cdot (Ok \cdot x))$
unfolding *catchET-def* **by** *simp*

lemma *runErrorT-fmapET* [*simp*]:
$runErrorT \cdot (fmapET \cdot f \cdot m) =$
$bind \cdot (runErrorT \cdot m) \cdot (\Lambda\ n.\ case\ n\ of$
$Err \cdot e \Rightarrow return \cdot (Err \cdot e) \mid Ok \cdot x \Rightarrow return \cdot (Ok \cdot (f \cdot x)))$
unfolding *fmapET-def* **by** *simp*

16.5 Laws

lemma *bindET-unitET* [*simp*]:
$bindET \cdot (unitET \cdot x) \cdot k = k \cdot x$
by (*rule errorT-eqI*, *simp*)

lemma *catchET-unitET* [*simp*]:
$catchET \cdot (unitET \cdot x) \cdot h = unitET \cdot x$
by (*rule errorT-eqI*, *simp*)

lemma *catchET-throwET* [*simp*]:
$catchET \cdot (throwET \cdot e) \cdot h = h \cdot e$
by (*rule errorT-eqI*, *simp*)

lemma *liftET-return*:
$liftET \cdot (return \cdot x) = unitET \cdot x$
by (*rule errorT-eqI*, *simp add*: *fmap-return*)

lemma *liftET-bind*:
$liftET \cdot (bind \cdot m \cdot k) = bindET \cdot (liftET \cdot m) \cdot (liftET\ oo\ k)$
by (*rule errorT-eqI*, *simp add*: *fmap-bind bind-fmap*)

lemma *bindET-throwET*:
$bindET \cdot (throwET \cdot e) \cdot k = throwET \cdot e$
by (*rule errorT-eqI*, *simp*)

lemma *bindET-bindET*:
$bindET \cdot (bindET \cdot m \cdot h) \cdot k = bindET \cdot m \cdot (\Lambda\ x.\ bindET \cdot (h \cdot x) \cdot k)$
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*:

$fmapET \cdot f \cdot (fmapET \cdot g \cdot m) = fmapET \cdot (\Lambda\ x.\ f \cdot (g \cdot x)) \cdot m$
by (*simp add*: *fmapET-def bindET-bindET*)

Right unit monad law is not satisfied in general.

lemma *bindET-unitET-right-counterexample*:
fixes m :: $'a \cdot ('m{::}monad, 'e)$ *errorT*
assumes $m = ErrorT \cdot (return \cdot \bot)$
assumes $return \cdot \bot \neq (\bot :: ('a \cdot 'e\ error) \cdot 'm)$
shows $bindET \cdot m \cdot unitET \neq m$
by (*simp add*: *errorT-eq-iff assms*)

Right unit is satisfied for inner monads with strict return.

lemma *bindET-unitET-right-restricted*:
fixes m :: $'a \cdot ('m{::}monad, 'e)$ *errorT*
assumes $return \cdot \bot = (\bot :: ('a \cdot 'e\ error) \cdot 'm)$
shows $bindET \cdot m \cdot unitET = m$
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* :: $'a \cdot ('m{::}monad,\ 'e)$ *errorT* $\Rightarrow$ *bool*
where *invar-bottom*: *invar* $\bot$
| *invar-lub*: $\bigwedge Y.\ [\![chain\ Y;\ \bigwedge i.\ invar\ (Y\ i)]\!] \Longrightarrow invar\ (\bigsqcup i.\ Y\ i)$
| *invar-unitET*: $\bigwedge x.\ invar\ (unitET \cdot x)$
| *invar-bindET*: $\bigwedge m\ k.\ [\![invar\ m;\ \bigwedge x.\ invar\ (k \cdot x)]\!] \Longrightarrow invar\ (bindET \cdot m \cdot k)$
| *invar-throwET*: $\bigwedge e.\ invar\ (throwET \cdot e)$
| *invar-catchET*: $\bigwedge m\ h.\ [\![invar\ m;\ \bigwedge e.\ invar\ (h \cdot e)]\!] \Longrightarrow invar\ (catchET \cdot m \cdot h)$
| *invar-liftET*: $\bigwedge m.\ invar\ (liftET \cdot m)$

Right unit is satisfied for arguments built from standard functions.

lemma *bindET-unitET-right-invar*:
assumes *invar m*
shows $bindET \cdot m \cdot unitET = m$
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 $f :: 'a \to 'b$ **and** $m :: 'a\cdot('m::monad,'e)\ errorT$
  assumes *invar m*
  shows $fmap\cdot f\cdot m = bindET\cdot m\cdot(\Lambda\ x.\ unitET\cdot(f\cdot x))$
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 $invar' :: 'a\cdot('m::monad,\ 'e)\ errorT \Rightarrow bool$
  where $invar'\ m \longleftrightarrow fmapET\cdot ID\cdot m = m$

All standard operations preserve the invariant.

lemma *invar'-unitET*: $invar'\ (unitET\cdot x)$
  unfolding *invar'-def* **by** (*simp add*: *fmapET-def*)

lemma *invar'-fmapET*: $invar'\ m \Longrightarrow invar'\ (fmapET\cdot f\cdot m)$
  unfolding *invar'-def*

by (*erule subst*, *simp add*: *fmapET-def bindET-bindET eta-cfun*)

lemma *invar′-bindET*: $[\![$*invar′ m*; $\bigwedge x$. *invar′* $(k\cdot x)]\!] \Longrightarrow$ *invar′* (*bindET*$\cdot m\cdot k$)
unfolding *invar′-def*
by (*simp add*: *fmapET-def bindET-bindET eta-cfun*)

lemma *invar′-throwET*: *invar′* (*throwET*$\cdot e$)
unfolding *invar′-def* **by** (*simp add*: *fmapET-def bindET-throwET eta-cfun*)

lemma *invar′-catchET*: $[\![$*invar′ m*; $\bigwedge e$. *invar′* $(h\cdot e)]\!] \Longrightarrow$ *invar′* (*catchET*$\cdot m\cdot h$)
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*$\cdot$*e* **in** *subst*) **back**
apply (*simp add*: *eta-cfun*)
apply *simp*
done

lemma *invar′-liftET*: *invar′* (*liftET*$\cdot m$)
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*$\cdot$(*fmapET*$\cdot f\cdot m$)$\cdot$*unitET* = *fmapET*$\cdot f\cdot m$
by (*simp add*: *fmapET-def bindET-bindET*)

lemma *invar′-right-unit*: *invar′ m* $\Longrightarrow$ *bindET*$\cdot m\cdot$*unitET* = *m*
unfolding *invar′-def* **by** (*erule subst*, *rule bindET-fmapET-unitET*)

lemma *invar′-monad-fmap*:
invar′ m $\Longrightarrow$ *fmapET*$\cdot f\cdot m$ = *bindET*$\cdot m\cdot(\Lambda\ x.$ *unitET*$\cdot(f\cdot x))$
unfolding *invar′-def* **by** (*erule subst*, *simp add*: *errorT-eq-iff*)

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

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

lemma *writerT-cases* [*case-names WriterT*]:
  obtains k **where** $y = WriterT{\cdot}k$
proof
  show $y = WriterT{\cdot}(runWriterT{\cdot}y)$
    by (*cases y*, *simp-all*)
qed

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

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

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

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

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

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

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

17.2 Functor class instance

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

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

instance *writerT* :: (*functor*, *domain*) *functor*
proof
  fix *f g* :: *udom* → *udom* **and** *xs* :: *udom*·('*a*,'*b*) *writerT*
  show $fmapU \cdot f \cdot (fmapU \cdot g \cdot xs) = fmapU \cdot (\Lambda\ x.\ f \cdot (g \cdot x)) \cdot xs$
    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*·('*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* → '*a*·('*m*::*monad*,'*w*::*monoid*) *writerT*
  where $unitWT = (\Lambda\ x.\ WriterT \cdot (return \cdot (Writer \cdot mempty \cdot x)))$

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

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

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

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

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

$runWriterT\cdot(unitWT\cdot x) = return\cdot(Writer\cdot mempty\cdot x)$
unfolding *unitWT-def* **by** *simp*

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

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

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

lemma *runWriterT-fmapWT* [*simp*]:
$runWriterT\cdot(fmapWT\cdot f\cdot m) =$
$runWriterT\cdot m \ggg= (\Lambda\ (Writer\cdot w\cdot x).\ return\cdot(Writer\cdot w\cdot(f\cdot x)))$
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*:
$liftWT\cdot(return\cdot x) = unitWT\cdot x$
by (*rule writerT-eqI*, *simp add*: *fmap-return*)

lemma *liftWT-bind*:
$liftWT\cdot(bind\cdot m\cdot k) = bindWT\cdot(liftWT\cdot m)\cdot(liftWT\ oo\ k)$
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*:
$fmapWT\cdot f\cdot(fmapWT\cdot g\cdot m) = fmapWT\cdot(\Lambda\ x.\ f\cdot(g\cdot x))\cdot m$
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 $k :: 'a \to 'b \cdot ('m::monad,'w::monoid)\ writerT$
assumes *1*: $k \cdot x = WriterT \cdot (return \cdot \bot)$
assumes *2*: $return \cdot \bot \neq (\bot :: ('b \cdot 'w\ writer) \cdot 'm::monad)$
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 $k :: 'a \to 'b \cdot ('m::monad,'w::monoid)\ writerT$
assumes $return \cdot \bot = (\bot :: ('b \cdot 'w\ writer) \cdot 'm)$
shows $bindWT \cdot (unitWT \cdot x) \cdot k = k \cdot x$
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 :: $'a\cdot('m::monad,'w::monoid)$ *writerT*
  assumes $m = WriterT\cdot(return\cdot\bot)$
  assumes $return\cdot\bot \neq (\bot :: ('a\cdot'w\ writer)\cdot'm)$
  shows $bindWT\cdot m\cdot unitWT \neq m$
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 :: $'a\cdot('m::monad,'w::monoid)$ *writerT*
  assumes $return\cdot\bot = (\bot :: ('a\cdot'w\ writer)\cdot'm)$
  shows $bindWT\cdot m\cdot unitWT = m$
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* :: $'a\cdot('m::monad,\ 'w::monoid)\ writerT \Rightarrow bool$
  where *invar-bottom*: *invar* $\bot$
  | *invar-lub*: $\bigwedge Y.\ [\![chain\ Y;\ \bigwedge i.\ invar\ (Y\ i)]\!] \Longrightarrow invar\ (\bigsqcup i.\ Y\ i)$
  | *invar-unitWT*: $\bigwedge x.\ invar\ (unitWT\cdot x)$
  | *invar-bindWT*: $\bigwedge m\ k.\ [\![invar\ m;\ \bigwedge x.\ invar\ (k\cdot x)]\!] \Longrightarrow invar\ (bindWT\cdot m\cdot k)$
  | *invar-tellWT*: $\bigwedge x\ w.\ invar\ (tellWT\cdot x\cdot w)$
  | *invar-liftWT*: $\bigwedge m.\ invar\ (liftWT\cdot m)$

Right unit is satisfied for arguments built from standard functions.

lemma *bindWT-unitWT-right-invar*:
  fixes *m* :: *'a·('m::monad,'w::monoid) 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 $runWriterT \cdot m \gg\!= (\Lambda\,(Writer \cdot w \cdot x).\ return \cdot (Writer \cdot w \cdot x)) = runWriterT \cdot m$
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* $(k \cdot x)$
shows $bindWT \cdot (unitWT \cdot x) \cdot k = k \cdot x$
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 $invar' :: 'a \cdot ('m::monad, 'w::monoid)\ writerT \Rightarrow bool$
where $invar'\ m \longleftrightarrow fmapWT \cdot ID \cdot m = m$

All standard operations preserve the invariant.

lemma *invar'-bottom*: $invar'\ \bot$
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*: $[\![invar'\ m;\ \bigwedge x.\ invar'\ (k \cdot x)]\!] \Longrightarrow invar'\ (bindWT \cdot m \cdot k)$
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*:
$bindWT \cdot (unitWT \cdot x) \cdot (\Lambda\ x.\ fmapWT \cdot f \cdot (k \cdot x))$
$= fmapWT \cdot f \cdot (k \cdot x)$
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 $bindWT \cdot (fmapWT \cdot f \cdot m) \cdot unitWT = fmapWT \cdot f \cdot m$
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*: $invar'\ m \Longrightarrow bindWT \cdot m \cdot unitWT = m$
unfolding *invar′-def* **by** (*erule subst*, *rule bindWT-fmapWT-unitWT*)

lemma *invar′-monad-fmap*:
$invar'\ m \Longrightarrow fmapWT \cdot f \cdot m = bindWT \cdot m \cdot (\Lambda\ x.\ unitWT \cdot (f \cdot x))$
unfolding *invar′-def*
by (*erule subst*, *simp add*: *writerT-eq-iff mempty-right*)

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